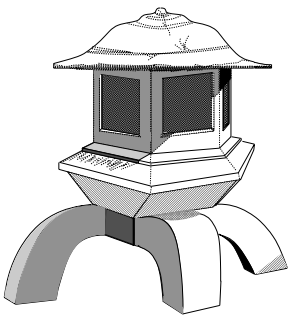
The HOL System REFERENCE



Preface

This volume is the reference manual for the HOL system. It is one of four documents making up the documentation for HOL:

1. *LOGIC*: a formal description of the higher order logic implemented by the HOL system;
2. *TUTORIAL*: a tutorial introduction to HOL, with case studies;
3. *DESCRIPTION*: a detailed user's guide for the HOL system;
4. *REFERENCE*: the reference manual for HOL.

These four documents will be referred to by the short names (in small slanted capitals) given above.

This document, *REFERENCE*, provides documentation on all the pre-defined ML variable bindings in the HOL system. These include: general-purpose functions, such as ML functions for list processing, arithmetic, input/output, and interface configuration; functions for processing the types and terms of the HOL logic, for setting up theories, and for using the subgoal package; primitive and derived forward inference rules; tactics and tacticals; and pre-proved built-in theorems.

The *REFERENCE* volume is purely for reference and browsing. It is generated from the same database that is used by the help system. For an introduction to the HOL system, see *TUTORIAL*; for a systematic presentation, see *DESCRIPTION* and *LOGIC*.

Acknowledgements

The bulk of HOL is based on code written by—in alphabetical order—Hasan Amjad, Richard Boulton, Anthony Fox, Mike Gordon, Elsa Gunter, John Harrison, Peter Homeier, Gérard Huet (and others at INRIA), Joe Hurd, Ramana Kumar, Ken Friis Larsen, Tom Melham, Robin Milner, Lockwood Morris, Magnus Myreen, Malcolm Newey, Michael Norrish, Larry Paulson, Konrad Slind, Don Syme, Chun Tian, Thomas Türk, Chris Wadsworth, and Tjark Weber. Many others have supplied parts of the system, bug fixes, etc.

Current edition

The current edition of all four volumes (*LOGIC*, *TUTORIAL*, *DESCRIPTION* and *REFERENCE*) has been prepared by Michael Norrish and Konrad Slind. Further contributions to these volumes came from: Hasan Amjad, who developed a model checking library and wrote sections describing its use; Jens Brandt, who developed and documented a library for the rational numbers; Anthony Fox, who formalized and documented new word theories and the associated libraries; Mike Gordon, who documented the libraries for BDDs and SAT; Peter Homeier, who implemented and documented the quotient library; Joe Hurd, who added material on first order proof search; Chun Tian, who documented the HOL theories for probability and measure theory; and Tjark Weber, who wrote libraries for Satisfiability Modulo Theories (SMT) and Quantified Boolean Formulae (QBF).

The material in the third edition constitutes a thorough re-working and extension of previous editions. The only essentially unaltered piece is the semantics by Andy Pitts (in *LOGIC*), reflecting the fact that, although the HOL system has undergone continual development and improvement, the HOL logic is unchanged since the first edition (1988).

Second edition

The second edition of *REFERENCE* was a joint effort by the Cambridge HOL group.

First edition

The three volumes *TUTORIAL*, *DESCRIPTION* and *REFERENCE* were produced at the Cambridge Research Center of SRI International with the support of DSTO Australia.

The HOL documentation project was managed by Mike Gordon, who also wrote parts of *DESCRIPTION* and *TUTORIAL* using material based on an early paper describing the HOL system[1] and *The ML Handbook*[2]. Other contributers to *DESCRIPTION* incude Avra Cohn, who contributed material on theorems, rules, conversions and tactics, and also composed the index (which was typeset by Juanito Camilleri); Tom Melham, who wrote the sections describing type definitions, the concrete type package and the 'resolution' tactics; and Andy Pitts, who devised the set-theoretic semantics of the HOL logic and wrote the material describing it.

The original document design used LaTeX macros supplied by Elsa Gunter, Tom Melham and Larry Paulson. The typesetting of all three volumes was managed by Tom Melham. The cover design is by Arnold Smith, who used a photograph of a 'snow watching lantern' taken by Avra Cohn (in whose garden the original object resides). John Van Tassel composed the LaTeX picture of the lantern.

Many people other than those listed above have contributed to the HOL documentation effort, either by providing material, or by sending lists of errors in the first edition. Thanks to everyone who helped, and thanks to DSTO and SRI for their generous support.

[1] M.J.C. Gordon, 'HOL: a Proof Generating System for Higher Order Logic', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, (Kluwer Academic Publishers, 1988), pp. 73–128.

[2] *The ML Handbook*, unpublished report from Inria by Guy Cousineau, Mike Gordon, Gérard Huet, Robin Milner, Larry Paulson and Chris Wadsworth.

In Memory of Mike Gordon

As documented in the academic literature, in material available from his archived webpages at the University of Cambridge Computer Laboratory, and in these manuals, Mike Gordon was HOL's primary creator and developer. Mike not only created a significant piece of software, inspiring this and many other projects since, but also built a world-leading research group in the Computer Laboratory. This research environment was wonderfully productive for many of the system's authors, and we all owe Mike an enormous debt for both the original work on HOL, and for the way he and his group supported our own development as researchers and HOL hackers.

Mike Gordon, 1948–2017

Chapter 1

Entries

This chapter provides manual entries for pre-defined ML identifiers in the HOL system. These include: general-purpose functions, such as functions for list processing, arithmetic, input/output, and interface configuration; functions for processing the types and terms of the HOL logic, for setting up theories, and for using the subgoal package; primitive and derived forward inference rules; and tactics and tacticals. The arrangement is alphabetical. If an entry's title box includes a parenthesised word to the right, this identifies the ML structure where that identifier is bound. The interactive system starts with some structures already present, others will need to be `load`-ed first.

$ (Lib)

```
Lib.$ : ('a -> 'b) * 'a -> 'b
```

Right-associated infix function application operator

Writing `f $ x` is another way of writing `f x`.

Failure

Fails if `f x` would fail.

Comments

Because `$` is right-associated, this can be a convenient way to avoid parentheses. For example,

```
first_x_assum $ qspec_then ‘m’ $ qx_choose_then ‘z’ strip_assume_tac
```

instead of

```
first_x_assum (qspec_then ‘m’ (qx_choose_then ‘z’ strip_assume_tac))
```

Note also that `$` is tighter than the various `THEN` infixes, so a tactic such as the one above can be used in a proof without needing protection by extra parentheses.

&& (BasicProvers)

```
op BasicProvers.&& : simpset * thm list -> simpset
```

Re-exported from `bossLib.&&`. See that entry for full documentation.

&& (bossLib)

```
op bossLib.&& : simpset * thm list -> simpset
```

Also exported as `BasicProvers.&&`

Infix operator for adding theorems into a simpset.

It is occasionally necessary to extend an existing simpset `ss` with a collection `rwlist` of new rewrite rules. To achieve this, one applies the `&&` function via `ss && rwlist`.

Failure

Never fails.

Example

```
> val ss = boolSimps.bool_ss && pairLib.pair_rws;
val ss =
   Included fragments (with 1 anonymous fragment [remove using name ""]):
      BOOL, CONG, NOT, PURE, UNWIND
   Rewrites (with 9 anonymous rewrites)
   Other net names/keys:
      .rewrite:COND_BOOL_CLAUSES.1, .rewrite:COND_BOOL_CLAUSES.2,
      .rewrite:COND_BOOL_CLAUSES.3, .rewrite:COND_BOOL_CLAUSES.4,
      .rewrite:EXCLUDED_MIDDLE'.1, .rewrite:EXISTS_REFL'.1,
      .rewrite:EXISTS_UNIQUE_REFL'.1, .rewrite:NOT_AND'.1,
      .rewrite:lift_disj_eq.1, .rewrite:lift_disj_eq.2,
[...Output elided...]
```

Comments

Of limited applicability since most of the tactics for rewriting already include this functionality. However, applications of `ZAP_TAC` can benefit.

See also

`simpLib.++`, `simpLib.SIMP_CONV`, `bossLib.RW_TAC`

++ (simpLib)

```
op simpLib.++ : simpset * ssfrag -> simpset
```

Infix operator for adding an `ssfrag` item into a simpset.

`bossLib.++` is identical to `simpLib.++`.

If the incoming fragment has a name and that name is currently in the simpset's *excluded* set (set up by an earlier `exclude_ssfrags` call — e.g. via the `Proof[exclude_frags = ...]` attribute), the addition is a silent no-op and `ss ++ frag` returns `ss` unchanged. Use `force_add` (or, in the rewrite-list of a simplification tactic, the `SF` marker) to override that prohibition.

See also

`bossLib.++`, `simpLib.exclude_ssfrags`, `simpLib.force_add`

--> (Type)

```
op Type.--> : hol_type * hol_type -> hol_type
```

Right associative infix operator for building a function type.

If `ty1` and `ty2` are HOL types, then `ty1 --> ty2` builds the HOL type `ty1 -> ty2`.

Failure

Never fails.

Example

```
> bool --> alpha;
val it = ":bool -> α": hol_type
```

Comments

This operator associates to the right, that is, `ty1 --> ty2 --> ty3` is identical to `ty1 --> (ty2 --> ty3)`.

See also

`Type.dom_rng`, `Type.mk_type`, `Type.mk_thy_type`

== (Parse)

```
Parse.== : hol_type quotation -> 'a -> hol_type
```

Parses a quotation into a HOL type.

An invocation ==` ... `== is identical to `Type ` ... ``.

Failure

As for `Parse.Type`.

Turns strings into types.

See also

`Parse.Term`

Abbr (BasicProvers)

```
BasicProvers.Abbr : term quotation -> thm
```

Signals to simplification tactics that an abbreviation should be used.

The `Abbr` function is used to signal to various simplification tactics that an abbreviation in the current goal should be eliminated before simplification proceeds. Each theorem

created by Abbr is removed from the tactic's theorem-list argument, and causes a call to Q.UNABBREV_TAC with that Abbr theorem's argument. Finally, the simplification tactic continues, with the rest of the theorem-list as its argument. Thus,

```
tac [..., Abbr`v`, ..., Abbr`u`, ...]
```

has the same effect as

```
Q.UNABBREV_TAC `v` THEN Q.UNABBREV_TAC `u` THEN
tac [..., ..., ...]
```

Every theorem created by Abbr in the argument list is treated in this way. The tactics that understand Abbr arguments are SIMP_TAC, ASM_SIMP_TAC, FULL_SIMP_TAC, RW_TAC and SRW_TAC.

Failure

Abbr itself never fails, but the tactic it is used in may do, particularly if the induced calls to UNABBREV_TAC fail.

Comments

This function is a notational convenience that allows the effect of multiple tactics to be packaged into just one.

See also

Q.ABBREV_TAC, simpLib.SIMP_TAC, Q.UNABBREV_TAC

ABBREV_TAC (Q)

```
Q.ABBREV_TAC : term quotation -> tactic
```

Introduces an abbreviation into a goal.

The tactic `Q.ABBREV_TAC q` parses the quotation `q` in the context of the goal to which it is applied. The result must be a term of the form `v = e` with `v` a variable. The effect of the tactic is to replace the term `e` wherever it occurs in the goal by `v` (or a primed variant of `v` if `v` already occurs in the goal), and to add the assumption `Abbrev(v = e)` to the goal's assumptions. Again, if `v` already occurs free in the goal, then the new assumption will be `Abbrev(v' = e)`, with `v'` a suitably primed version of `v`.

It is not an error if the expression `e` does not occur anywhere within the goal. In this situation, the effect of the tactic is simply to add the assumption `Abbrev(v = e)`.

The `Abbrev` constant is defined in `markerTheory` to be the identity function over boolean values. It is used solely as a tag, so that abbreviations can be found by other tools, and so that simplification tactics such as `RW_TAC` will not eliminate them. When it sees them as part of its context, the simplifier treats terms of the form `Abbrev(v = e)` as assumptions `e = v`. In this way, the simplifier can use abbreviations to create further sharing, after an abbreviation's creation.

Abbreviation assumptions of this form "protect" their variable argument; simplification tactics (e.g., `fs`) will not replace the variable `v`, even though they may have been passed rewrites to use such as `v = e'`.

Failure

Fails if the quotation is ill-typed. This may happen because variables in the quotation that also appear in the goal are given the same type in the quotation as they have in the goal. Also fails if the variable of the equation appears in the expression that it is supposed to be abbreviating.

Example

Substitution in the goal:

```
> Q.ABBREV_TAC ‘n = 10’ ([], “10 < 9 * 10”);
val it = ([([“Abbrev(n = 10)”], “n < 9 * n”)], fn) :
  (term list * term) list * (thm list -> thm)
```

and the assumptions:

```
> Q.ABBREV_TAC ‘m = n + 2’ ([“f (n + 2) < 6”], “n < 7”);
val it = ([([“Abbrev(m = n + 2)”, “f m < 6”], “n < 7”)], fn) :
  (term list * term) list * (thm list -> thm)
```

and both

```
> Q.ABBREV_TAC ‘u = x ** 32’ ([“x ** 32 = f z”],
                              “g (x ** 32 + 6) - 10 < 65”);
val it =
    ([([“Abbrev(u = x ** 32)”, “u = f z”], “g (u + 6) - 10 < 65”)],
     fn) :
    (term list * term) list * (thm list -> thm)
```

Comments

The `bossLib` library provides `qabbrev_tac` as a synonym for `Q.ABBREV_TAC`.

It is possible to abbreviate functions, using quotations such as ‘`f = \n. n + 3`’. When this is done `ABBREV_TAC` will not itself do anything more than replace exact copies of the abstraction, but the simplifier will subsequently see occurrences of the pattern and replace them.

Thus:

```
> (qabbrev_tac ‘f = \x. x + 1’ >> asm_simp_tac bool_ss [])
     ([], “3 + 1 = 4 + 1”);
val it =
   ([([“Abbrev (f = (\x. x + 1))”], “f 3 = f 4”)], fn):
   goal list * (thm list -> thm)
```

where the simplifier has seen occurrences of the `x+1` pattern and replaced it with calls to the `f`-abbreviation.

See also

`BasicProvers.Abbr`, `Q.HO_MATCH_ABBREV_TAC`, `Q.MATCH_ABBREV_TAC`, `Q.UNABBREV_TAC`

ABS (Thm)

```
Thm.ABS : term -> thm -> thm
```

Abstracts both sides of an equation.

```
        A |- t1 = t2
   ------------------------  ABS x            [Where x is not free in A]
    A |- (\x.t1) = (\x.t2)
```

Failure

If the theorem is not an equation, or if the variable `x` is free in the assumptions `A`.

Example

```
> let val m = “m:bool”
  in
     ABS m (REFL m)
  end;
val it = ⊢ (λm. m) = (λm. m): thm
```

See also

`Drule.ETA_CONV`, `Drule.EXT`, `Drule.MK_ABS`

ABS_CONV (Conv)

```
Conv.ABS_CONV : conv -> conv
```

Applies a conversion to the body of an abstraction.

If `c` is a conversion that maps a term `tm` to the theorem `|- tm = tm'`, then the conversion `ABS_CONV c` maps abstractions of the form `\x.tm` to theorems of the form:

```
|- (\x.tm) = (\x.tm')
```

That is, `ABS_CONV c (\x.t)` applies `c` to the body of the abstraction `\x.t`.

Failure

ABS_CONV c tm fails if tm is not an abstraction or if tm has the form \x.t but the conversion c fails when applied to the term t. The function returned by ABS_CONV c may also fail if the ML function c:term->thm is not, in fact, a conversion (i.e. a function that maps a term M to a theorem |- M = N).

Example

```
> ABS_CONV SYM_CONV (Term `\x. 1 = x`)
val it = ⊢ (λx. 1 = x) = (λx. x = 1): thm
```

See also

Conv.RAND_CONV, Conv.RATOR_CONV, Conv.SUB_CONV, Conv.BINDER_CONV, Conv.QUANT_CONV, Conv.STRIP_BINDER_CONV, Conv.STRIP_QUANT_CONV

ABS_TAC (Tactic)

```
Tactic.ABS_TAC : tactic
```

Strips lambda abstraction on both sides of an equation.

When applied to a goal of the form A ?- (\x. M) = (\y. N) (where M and N may or may not mention their respective bound variables), the tactic ABS_TAC strips away the lambda abstractions:

```
  A ?- (\x. f x) = (\y. g y)
============================  ABS_TAC
        A ?- f x = g x
```

Failure

Fails unless the goal has the above form, namely an equation both sides of which consist of a lamdba abstraction.

Comments

When the lambda abstractions' bound variables conflict with existing free variables in the goal, variants of those names may occur in the goal that results.

See also

`Tactic.AP_TERM_TAC`, `Tactic.AP_THM_TAC`

Absyn (Parse)

```
Parse.Absyn : term quotation -> Absyn.absyn
```

Implements the first phase of term parsing; the removal of special syntax.

`Absyn` takes a quotation and parses it into an abstract syntax tree of type `absyn`, using the current term and type grammars. This phase of parsing is unconcerned with types, and will happily parse meaningless expressions that are syntactically valid.

Example

`Absyn` will parse the expression `` `let x = e1 in e2` `` into

```
APP(APP(IDENT "LET", LAM(VIDENT "x", IDENT "e2")), IDENT "e1")
```

The record syntax `` `rec.fld1` `` is converted into something of the form

```
APP(IDENT "....fld1", IDENT "rec")
```

where the dots will actually be equal to the value of `GrammarSpecials.recsel_special` (a string).

Failure

Fails if the quotation has a syntax error.

`Absyn` is not often used, but may be handy for implementing some weird and wonderful concrete syntax that surpasses the functionality of the HOL parser.

See also

`Parse.Term`, `Parse.term_grammar`

AC (simpLib)

```
simpLib.AC : thm -> thm -> thm
```

Packages associativity and commutativity theorems for use in the simplifier.

The `AC` function combines an associativity and commutativity theorem. The resulting theorem can be passed to the simplifier as if a rewrite, but will rather be used by the simplifier as the basis for performing AC-normalisation.

The theorems can be combined in either order, can be partly generalised, and need not express associativity in any particular direction from left to right.

Failure

`AC` never fails, but if applied to theorems that are not of the required form, the simplifier will raise an exception when it attempts to use the result.

Example

```
> SIMP_CONV bool_ss [AC arithmeticTheory.ADD_COMM arithmeticTheory.ADD_ASSOC] ``3 +
val it = ⊢ 3 + x + y + 1 = x + (y + (1 + 3)): thm

> SIMP_CONV bool_ss [AC (GSYM arithmeticTheory.ADD_ASSOC) arithmeticTheory.ADD_COMM
val it = ⊢ x + 1 + y + 3 = x + (y + (1 + 3)): thm
```

See also

`simpLib.SSFRAG`

AC_CONV (Conv)

```
Conv.AC_CONV : (thm * thm) -> conv
```

Proves equality of terms using associative and commutative laws.

Suppose _ is a function, which is assumed to be infix in the following syntax, and `ath` and `cth` are theorems expressing its associativity and commutativity; they must be of the following form, except that any free variables may have arbitrary names and may be universally quantified:

```
ath = |- m _ (n _ p) = (m _ n) _ p
cth = |- m _ n = n _ m
```

Then the conversion `AC_CONV(ath,cth)` will prove equations whose left and right sides can be made identical using these associative and commutative laws.

Failure

Fails if the associative or commutative law has an invalid form, or if the term is not an equation between AC-equivalent terms.

Example

Consider the terms `x + SUC t + ((3 + y) + z)` and `3 + SUC t + x + y + z`. `AC_CONV` proves them equal.

```
- AC_CONV(ADD_ASSOC,ADD_SYM)
    (Term `x + (SUC t) + ((3 + y) + z) = 3 + (SUC t) + x + y + z`);

> val it =
  |- (x + ((SUC t) + ((3 + y) + z)) = 3 + ((SUC t) + (x + (y + z)))) = T
```

Comments

Note that the preproved associative and commutative laws for the operators +, *, /\ and \/ are already in the right form to give to `AC_CONV`.

See also

`Conv.SYM_CONV`

ACCEPT_TAC (Tactic)

`Tactic.ACCEPT_TAC : thm_tactic`

Solves a goal if supplied with the desired theorem (up to alpha-conversion).

`ACCEPT_TAC` maps a given theorem `th` to a tactic that solves any goal whose conclusion is alpha-convertible to the conclusion of `th`.

Failure

`ACCEPT_TAC th (A,g)` fails if the term `g` is not alpha-convertible to the conclusion of the supplied theorem `th`.

Example

`ACCEPT_TAC` applied to the axiom

```
BOOL_CASES_AX = |- !t. (t = T) \/ (t = F)
```

will solve the goal

```
?- !x. (x = T) \/ (x = F)
```

but will fail on the goal

```
?- !x. (x = F) \/ (x = T)
```

Used for completing proofs by supplying an existing theorem, such as an axiom, or a lemma already proved.

See also

`Tactic.MATCH_ACCEPT_TAC`

aconv (Term)

```
Term.aconv : term -> term -> bool
```

Tests for alpha-convertibility of terms.

When applied to two terms, `aconv` returns `true` if they are alpha-convertible, and `false` otherwise. Two terms are alpha-convertible if they differ only in the way that names have been given to bound variables.

Failure

Never fails.

Example

```
> aconv (Term `?x y. x /\ y`) (Term `?y x. y /\ x`)
val it = true: bool
```

See also

`Thm.ALPHA`, `Drule.ALPHA_CONV`

ADD_ASSUM (Drule)

```
Drule.ADD_ASSUM : term -> thm -> thm
```

Adds an assumption to a theorem.

When applied to a boolean term `s` and a theorem `A |- t`, the inference rule `ADD_ASSUM` returns the theorem `A u {s} |- t`.

```
      A |- t
--------------  ADD_ASSUM s
 A u {s} |- t
```

Failure

Fails unless the given term has type `bool`.

See also

`Thm.ASSUME`, `Drule.UNDISCH`

add_bare_numeral_form (Parse)

```
Parse.add_bare_numeral_form : (char * string option) -> unit
```

Adds support for annotated numerals to the parser/pretty-printer.

The function `add_bare_numeral_form` allows the user to give special meaning to strings of digits that are suffixed with single characters. A call to this function with pair argument `(c, s)` adds `c` as a possible suffix. Subsequently, if a sequence of digits is parsed, and it has the character `c` directly after the digits, then the natural number corresponding to these digits is made the argument of the "map function" corresponding to `s`.

This map function is computed as follows: if the `s` option value is `NONE`, then the function is considered to be the identity and never really appears; the digits denote a natural number. If the value of `s` is `SOME s'`, then the parser translates the string to an application of `s'` to the natural number denoted by the digits.

Failure

Fails if the suffix character is not a letter.

Example

The following function, `binary_of`, defined with equations:

```
val bthm =
  |- binary_of n = if n = 0 then 0
                   else n MOD 10 + 2 * binary_of (n DIV 10) : thm
```

can be used to convert numbers whose decimal notation is `x`, to numbers whose binary notation is `x` (as long as `x` only involves zeroes and ones).

The following call to `add_bare_numeral_form` then sets up a numeral form that could be used by users wanting to deal with binary numbers:

```
- add_bare_numeral_form(#"b", SOME "binary_of");
> val it = () : unit

- Term`1011b`;
> val it = `1011b` : term

- dest_comb it;
> val it = (`binary_of`, `1011`) : term * term
```

Comments

It is highly recommended that users avoid using suffixes that might be interpreted as hexadecimal digits A to F, in either upper or lower case. Further, HOL convention has it that suffix character should be lower case.

If one has a range of values that are usefully indexed by natural numbers, the function `add_bare_numeral_form` provides a syntactically convenient way of reading and writing these values. If there are other functions in the range type such that the mapping function is a homomorphism from the natural numbers, then `add_numeral_form` could be used, and the appropriate operators (+, * etc) overloaded.

See also

`Parse.add_numeral_form`

ADD_CONV (reduceLib)

`reduceLib.ADD_CONV : conv`

Calculates by inference the sum of two numerals.

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...) of type `:num`, then `ADD_CONV ``m + n``` returns the theorem:

```
|- m + n = s
```

where `s` is the numeral that denotes the sum of the natural numbers denoted by `m` and `n`.

Failure

`ADD_CONV tm` fails unless `tm` is of the form ``m + n``, where `m` and `n` are numerals of type `:num`.

Example

```
> reduceLib.ADD_CONV ``75 + 25``;
val it = ⊢ 75 + 25 = 100: thm
```

See also

`reduceLib.MUL_CONV`

add_implicit_rewrites (Rewrite)

```
Rewrite.add_implicit_rewrites: thm list -> unit
```

Augments the built-in database of simplifications automatically included in rewriting.

Used to build up the power of the built-in simplification set.

See also

`Rewrite.set_implicit_rewrites`

add_infix (Parse)

```
Parse.add_infix : string * int * HOLgrammars.associativity -> unit
```

Adds a string as an infix with the given precedence and associativity to the term grammar.

This function adds the given string to the global term grammar such that the string

```
<str1> s <str2>
```

will be parsed as

```
s <t1> <t2>
```

where `<str1>` and `<str2>` have been parsed to two terms `<t1>` and `<t2>`. The parsing process does not pay any attention to whether or not `s` corresponds to a constant or not. This resolution happens later in the parse, and will result in either a constant or a variable with name `s`. In fact, if this name is overloaded, the eventual term generated may have a constant of quite a different name again; the resolution of overloading comes as a separate phase (see the entry for `overload_on`).

Failure

`add_infix` fails if the precedence level chosen for the new infix is the same as a different type of grammar rule (e.g., suffix or binder), or if the specified precedence level has infixes already but of a different associativity.

It is also possible that the choice of string `s` will result in an ambiguous grammar. This will be marked with a warning. The parser may behave in strange ways if it encounters ambiguous phrases, but will work normally otherwise.

Example

Though we may not have + defined as a constant, we can still define it as an infix for the purposes of printing and parsing:

```
- add_infix ("+", 500, HOLgrammars.LEFT);
> val it = () : unit

- val t = Term`x + y`;
<<HOL message: inventing new type variable names: 'a, 'b, 'c.>>
> val t = `x + y` : term
```

We can confirm that this new infix has indeed been parsed that way by taking the resulting term apart:

```
- dest_comb t;
> val it = (`$+ x`, `y`) : term * term
```

With its new status, + has to be "quoted" with a dollar-sign if we wish to use it in a position where it is not an infix, as in the binding list of an abstraction:

```
- Term`\$+. x + y`;
<<HOL message: inventing new type variable names: 'a, 'b, 'c.>>
> val it = `\$+. x + y` : term
- dest_abs it;
> val it = (`$+`,`x + y`) : term * term
```

The generation of three new type variables in the examples above emphasises the fact that the terms in the first example and the body of the second are really no different from `f x y` (where `f` is a variable), and don't have anything to do with the constant for addition from `arithmeticTheory`. The new + infix is left associative:

```
- Term`x + y + z`;
<<HOL message: inventing new type variable names: 'a, 'b.>>
> val it = `x + y + z` : term

- dest_comb it;
> val it = (`$+ (x + y)`, `z`) : term * term
```

It is also more tightly binding than /\ (which has precedence 400 by default):

```
- Term`p /\ q + r`;
<<HOL message: inventing new type variable names: 'a, 'b.>>
> val it = `p /\ q + r` : term

- dest_comb it;
> val it = (`$/\ p`, `q + r`) : term * term
```

An attempt to define a right associative operator at the same level fails:

```
Lib.try add_infix("-", 500, HOLgrammars.RIGHT);

Exception raised at Parse.add_infix:
Grammar Error: Attempt to have differently associated infixes
               (RIGHT and LEFT) at same level
```

Similarly we can't define an infix at level 900, because this is where the (true prefix) rule for logical negation (~) is.

```
- Lib.try add_infix("-", 900, HOLgrammars.RIGHT);

Exception raised at Parse.add_infix:
Grammar Error: Attempt to have different forms at same level
```

Finally, an attempt to have a second + infix at a different precedence level causes grief when we later attempt to use the parser:

```
- add_infix("+", 400, HOLgrammars.RIGHT);
> val it = () : unit

- Term`p + q`;
<<HOL warning: Parse.Term: Grammar ambiguous on token pair + and +,
               and probably others too>>
<<HOL message: inventing new type variable names: 'a, 'b, 'c>>
> val it = ``p + q`` : term
```

In this situation, the behaviour of the parser will become quite unpredictable whenever the + token is encountered. In particular, + may parse with either fixity.

Most use of infixes will want to have them associated with a particular constant in which case the definitional principles (`new_infixl_definition` etc) are more likely to be appropriate. However, a development of a theory of abstract algebra may well want to have infix variables such as + above.

Comments

As with other functions in the `Parse` structure, there is a companion `temp_add_infix` function, which has the same effect on the global grammar, but which does not cause this effect to persist when the current theory is exported.

See also

Parse.add_rule, Parse.add_listform, Parse.Term

add_infix_type (Parse)

```
Parse.add_infix_type : {Assoc : associativity,
                        Name : string,
                        ParseName : string option,
                        Prec : int} ->
                       unit
```

Adds a type infix.

A call to add_infix_type adds an infix type symbol to the type grammar. The argument is a record of four values providing information about the infix.

The Assoc field specifies the associativity of the symbol (possible values: LEFT, RIGHT and NONASSOC). The standard HOL type infixes (+, #, -> and |->) are all right-associative. The Name field specifies the name of the binary type operator that is being mapped to. If the name of the type is not the same as the concrete syntax (as in all the standard HOL examples above), the concrete syntax can be provided in the ParseName field. The Prec field specifies the binding precedence of the infix. This should be a number less than 100, and probably greater than or equal to 50, where the function -> symbol lies. The greater the number, the more tightly the symbol attempts to "grab" its arguments.

Failure

Fails if the desired precedence level contains an existing infix with a different associativity.

Example

```
> Hol_datatype `atree = Nd of 'v => ('k # atree) list`;
val it = (): unit

> add_infix_type { Assoc = LEFT, Name = "atree",
                   ParseName = SOME ">->", Prec = 65 };
```

```
val it = (): unit

> type_of ``Nd``;
val it = ":α -> (β # (β >-> α)) list -> β >-> α": hol_type
```

add_listform (Parse)

```
Parse.add_listform :
  {separator : pp_element list, leftdelim : pp_element list,
   rightdelim : pp_element list, cons : string, nilstr : string,
   block_info : term_grammar.block_info } ->
  unit
```

Adds a "list-form" to the built-in grammar, allowing the parsing of strings such as `[a; b; c]` and `{}`.

The `add_listform` function allows the user to augment the HOL parser with rules so that it can turn a string of the form

```
<ld> str1 <sep> str2 <sep> ... strn <rd>
```

into the term

```
<cons> t1 (<cons> t2 ... (<cons> tn <nilstr>))
```

where `<ld>` is the left delimiter string, `<rd>` the right delimiter, and `<sep>` is the separator string from the fields of the record argument to the function. The various `stri` are strings representing the `ti` terms. Further, the grammar will also parse `<ld> <rd>` into `<nilstr>`.

The `pp_element` lists passed to this function for the `separator`, `leftdelim` and `rightdelim` fields are interpreted as by the `add_rule` function. These lists must have exactly one `TOK` element (this provides the string that will be printed), and other formatting elements such as `BreakSpace`.

The `block_info` field is a pair consisting of a "consistency" (`PP.CONSISTENT`, or `PP.INCONSISTENT`), and an indentation depth (an integer). The standard value for this field is `(PP.INCONSISTENT,0)`, which will cause lists too long to fit in a single line to

print with as many elements on the first line as will fit, and for subsequent elements to appear unindented on subsequent lines. Changing the "consistency" to `PP.CONSISTENT` would cause lists too long for a single line to print with one element per line. The indentation level number specifies the number of extra spaces to be inserted when a line-break occurs.

In common with the `add_rule` function, there is no requirement that the `cons` and `nilstr` fields be the names of constants; the parser/grammar combination will generate variables with these names if there are no corresponding constants.

The HOL pretty-printer is simultaneously aware of the new rule, and terms of the forms above will print appropriately.

Failure

Fails if any of the `pp_element` lists are ill-formed: if they include `TM`, `BeginFinalBlock`, or `EndInitialBlock` elements, or if do not include exactly one `TOK` element. Subsequent calls to the term parser may also fail or behave unpredictably if the strings chosen for the various fields above introduce precedence conflicts. For example, it will almost always be impossible to use left and right delimiters that are already present in the grammar, unless they are there as the left and right parts of a closefix.

Example

The definition of the "list-form" for lists in the HOL distribution is:

```
add_listform {separator = [TOK ";", BreakSpace(1,0)],
              leftdelim = [TOK "["], rightdelim = [TOK "]"],
              cons = "CONS", nilstr = "NIL",
              block_info = (PP.INCONSISTENT, 0)};
```

while the set syntax is defined similarly:

```
add_listform {leftdelim = [TOK "{"], rightdelim = TOK ["}"],
              separator = [";", BreakSpace(1,0)],
              cons = "INSERT", nilstr = "EMPTY",
              block_info = (PP.INCONSISTENT, 0)};
```

Used to make sequential term structures print and parse more pleasingly.

Comments

As with other parsing functions, there is a `temp_add_listform` version of this function, which has the same effect on the global grammar, but which does not cause this effect to persist when the current theory is exported.

See also

`Parse.add_rule`

add_numeral_form (Parse)

```
Parse.add_numeral_form : (char * string option) -> unit
```

Adds support for numerals of differing types to the parser/pretty-printer.

This function allows the user to extend HOL's parser and pretty-printer so that they recognise and print numerals. A numeral in this context is a string of digits. Each such string corresponds to a natural number (i.e., the HOL type `num`) but `add_numeral_form` allows for numerals to stand for values in other types as well.

A call to `add_numeral_form(c,s)` augments the global term grammar in two ways. Firstly, in common with the function `add_bare_numeral_form` (q.v.), it allows the user to write a single letter suffix after a numeral (the argument `c`). The presence of this character specifies `s` as the "injection function" which is to be applied to the natural number denoted by the preceding digits.

Secondly, the constant denoted by the `s` argument is overloaded to be one of the possible resolutions of an internal, overloaded operator, which is invisibly wrapped around all numerals that appear without a character suffix. After applying `add_numeral_form`, the function denoted by the argument `s` is now a possible resolution of this overloading, so numerals can now be seen as members of the range of the type of `s`.

Finally, if `s` is not `NONE`, the constant denoted by `s` is overloaded to be one of the possible resolutions of the string `&`. This operator is thus the standard way of writing the injection function from `:num` into other numeric types.

The injection function specifed by argument `s` is either the constant with name `s0`, if `s` is of the form `SOME s0`, or the identity function if `s` is `NONE`. Using `add_numeral_form` with `NONE` for this parameter is done in the development of `arithmeticTheory`, and should not be done subsequently.

Failure

Fails if `arithmeticTheory` is not loaded, as this is where the basic constants implementing natural number numerals are defined. Also fails if there is no constant with the given name, or if it doesn't have type `:num -> 'a` for some `'a`. Fails if `add_bare_numeral_form` would also fail on this input.

Example

The natural numbers are given numeral forms as follows:

```
val _ = add_numeral_form (#"n", NONE);
```

This is done in `arithmeticTheory` so that after it is loaded, one can write numerals and have them parse (and print) as natural numbers. However, later in the development, in `integerTheory`, numeral forms for integers are also introduced:

```
val _ = add_numeral_form(#"i", SOME "int_of_num");
```

Here `int_of_num` is the name of the function which injects natural numbers into integers. After this call is made, numeral strings can be treated as integers or natural numbers, depending on the context.

```
- load "integerTheory";
> val it = () : unit
- Term`3`;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = `3` : term
- type_of it;
> val it = `:int` : hol_type
```

The parser has chosen to give the string "3" integer type (it will prefer the most recently specified possibility, in common with overloading in general). However, numerals can appear with natural number type in appropriate contexts:

```
- Term`(SUC 3, 4 + ~x)`;
> val it = `(SUC 3,4 + ~x)` : term
- type_of it;
> val it = `:num # int` : hol_type
```

Moreover, one can always use the character suffixes to absolutely specify the type of the numeral form:

```
- Term`f 3 /\ p`;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = `f 3 /\ p` : term

- Term`f 3n /\ p`;
> val it = `f 3 /\ p` : term
```

Comments

Overloading on too many numeral forms is a sure recipe for confusion.

See also

`Parse.add_bare_numeral_form`, `Parse.overload_on`, `Parse.show_numeral_types`

add_rewrites (Rewrite)

```
Rewrite.add_rewrites : rewrites -> thm list -> rewrites
```

Add theorems to a collection of rewrite rules.

The function `add_rewrites` processes each element in a list of theorems and adds the resulting rewrite rules to a value of type `rewrites`.

Failure

Never fails.

Example

```
> load "pairTheory"; open pairTheory;
  add_rewrites empty_rewrites (PAIR_MAP_THM::pairLib.pair_rws);
```

```
val it = (): unit
val it =
   ⊢ (f ## g) (x,y) = (f x,g y);  ⊢ (FST x,SND x) = x;  ⊢ FST (x,y) = x;
   ⊢ SND (x,y) = y
   Number of rewrite rules = 4
   : rewrites
```

For building bespoke rewrite rule sets.

See also

Rewrite.bool_rewrites, Rewrite.empty_rewrites, Rewrite.implicit_rewrites, Rewrite.GEN_REWRITE_CONV, Rewrite.GEN_REWRITE_RULE, Rewrite.GEN_REWRITE_TAC

add_rule (Parse)

```
Parse.add_rule :
  {term_name : string, fixity : fixity,
   pp_elements: term_grammar.pp_element list,
   paren_style : term_grammar.ParenStyle,
   block_style : term_grammar.PhraseBlockStyle *
                 term_grammar.block_info}  -> unit
```

Adds a parsing/printing rule to the global grammar.

The function add_rule is a fundamental method for adding parsing (and thus printing) rules to the global term grammar that sits behind the term-parsing function Parse.Term, and the pretty-printer installed for terms. It is used for everything except the addition of list-forms, for which refer to the entry for add_listform.

There are five components in the record argument to add_rule. The term_name component is the name of the term (whether a constant or a variable) that will be generated at the head of the function application. Thus, the term_name component when specifying parsing for conditional expressions is COND.

The following values (all in structure Parse) are useful for constructing fixity values:

```
   val LEFT       : HOLgrammars.associativity
```

```
val RIGHT      : HOLgrammars.associativity
val NONASSOC   : HOLgrammars.associativity

val Binder     : fixity
val Closefix   : fixity
val Infixl     : int -> fixity
val Infixr     : int -> fixity
val Infix      : HOLgrammars.associativity * int -> fixity
val Prefix     : int -> fixity
val Suffix     : int -> fixity
```

The `Binder` fixity is for binders such as universal and existential quantifiers (`!` and `?`). Binders can actually be seen as (true) prefixes (should `` `!x. p /\ q` `` be parsed as `` `(!x. p) /\ q` `` or as `` `!x. (p /\ q)` ``?), but the `add_rule` interface only allows binders to be added at the one level (the weakest in the grammar). Further, when binders are added using this interface, all elements of the record apart from the `term_name` are ignored, so the name of the binder must be the same as the string that is parsed and printed (but see also restricted quantifiers: `associate_restriction`).

The remaining fixities all cause `add_rule` to pay due heed to the `pp_elements` ("parsing/printing elements") component of the record. As far as parsing is concerned, the only important elements are `TOK` and `TM` values, of the following types:

```
val TM  : term_grammar.pp_element
val TOK : string -> term_grammar.pp_element
```

The `TM` value corresponds to a "hole" where a sub-term is possible. The `TOK` value corresponds to a piece of concrete syntax, a string that is required when parsing, and which will appear when printing. The sequence of `pp_elements` specified in the record passed to `add_rule` specifies the "kernel" syntax of an operator in the grammar. The "kernel" of a rule is extended (or not) by additional sub-terms depending on the fixity type, thus:

```
Closefix    :      [Kernel]     (* no external arguments *)
Prefix      :      [Kernel] _   (* an argument to the right *)
Suffix      :    _ [Kernel]     (* an argument to the left *)
Infix       :    _ [Kernel] _   (* arguments on both sides *)
```

Thus simple infixes, suffixes and prefixes would have singleton `pp_element` lists, consisting of just the symbol desired. More complicated mix-fix syntax can be constructed

by identifying whether or not sub-term arguments exist beyond the kernel of concrete syntax. For example, syntax for the evaluation relation of an operational semantics (`_ |- _ --> _`) is an infix with a kernel delimited by `|-` and `-->` tokens. Syntax for denotation brackets `[| _ |]` is a closefix with one internal argument in the kernel.

The remaining sorts of possible `pp_element` values are concerned with pretty-printing. (The basic scheme is implemented on top of a standard Oppen-style pretty-printing package.) They are

```
(* where
     type term_grammar.block_info = PP.break_style * int
*)
val BreakSpace : (int * int) -> term_grammar.pp_element
val HardSpace : int -> term_grammar.pp_element

val BeginFinalBlock : term_grammar.block_info -> term_grammar.pp_element
val EndInitialBlock : term_grammar.block_info -> term_grammar.pp_element
val PPBlock : term_grammar.pp_element list * term_grammar.block_info
              -> term_grammar.pp_element

val OnlyIfNecessary : term_grammar.ParenStyle
val ParoundName : term_grammar.ParenStyle
val ParoundPrec : term_grammar.ParenStyle
val Always : term_grammar.ParenStyle
val IfNotTop : {realonly:bool} -> term_grammar.ParenStyle

val AroundEachPhrase : term_grammar.PhraseBlockStyle
val AroundSamePrec   : term_grammar.PhraseBlockStyle
val AroundSameName   : term_grammar.PhraseBlockStyle
val NoPhrasing       : term_grammar.PhraseBlockStyle
```

The two spacing values provide ways of specifying white-space should be added when terms are printed. Use of `HardSpace n` results in `n` spaces being added to the term whatever the context. On the other hand, `BreakSpace(m,n)` results in a break of width `m` spaces unless this makes the current line too wide, in which case a line-break will occur, and the next line will be indented an extra `n` spaces.

For example, the `add_infix` function (q.v.) is implemented in terms of `add_rule` in such a way that a single token infix `s`, has a `pp_element` list of

```
[HardSpace 1, TOK s, BreakSpace(1,0)]
```

This results in chains of infixes (such as those that occur with conjunctions) that break so as to leave the infix on the right hand side of the line. Under this constraint, printing can't break so as to put the infix symbol on the start of a line, because that would imply that the `HardSpace` had in fact been broken. (Consequently, if a change to this behaviour is desired, there is no global way of effecting it, but one can do it on an infix-by-infix basis by deleting the given rule (see, for example, `remove_termtok`) and then "putting it back" with different pretty-printing constraints.)

The `PPBlock` function allows the specification of nested blocks (blocks in the Oppen pretty-printing sense) within the list of `pp_elements`. Because there are sub-terms in all but the `Closefix` fixities that occur beyond the scope of the `pp_element` list, the `BeginFinalBlock` and `EndInitialBlock` functions can also be used to indicate the boundary of blocks whose outer extent is the term beyond the kernel represented by the `pp_element` list. There is an example of this below.

The possible `ParenStyle` values describe when parentheses should be added to terms. The `OnlyIfNecessary` value will cause parentheses to be added only when required to disambiguate syntax. The `ParoundName` will cause parentheses to be added if necessary, or where the head symbol has the given `term_name` and where this term is not the argument of a function with the same head name. This style of parenthesisation is used with tuples, for example. The `ParoundPrec` value is similar, but causes parentheses to be added when the term is the argument to a function with a different precedence level. The `IfNotTop` value will cause parentheses to appear whenever the term is not being printed as the "top" term. A term is considered to be "top" if it is the whole term being printed (and this is known as the "real top"), or if it occurs between two tokens that always delimit complete terms. For example, the semi-colons in a list-like form are such delimiters, as are the list-form's left and right brackets, as are the "if" and "then" tokens in an if-then-else form. Having the `realonly` parameter set to true will cause parentheses whenever the context is anything not the real top, while having it set to false will cause parentheses if in neither sort of "top". Finally, the `Always` value causes parentheses always to be added.

The `PhraseBlockStyle` values describe when pretty-printing blocks involving this term should be entered. The `AroundEachPhrase` style causes a pretty-printing block to be created around each term. This is not appropriate for operators such as conjunction however, where all of the arguments to the conjunctions in a list are more pleasingly thought of as being at the same level. This effect is gained by specifying either `AroundSamePrec` or `AroundSameName`. The former will cause the creation of a new block for the phrase if it is at a different precedence level from its parent, while the latter creates the block if the parent name is not the same. The former is appropriate for + and − which are at the same precedence level, while the latter is appropriate for `/\`. Finally, the `NoPhrasing` style causes there to be no block at all around terms controlled by this rule. The intention

in using such a style is to have block structure controlled by the level above.

Failure

This function will fail if the `pp_element` list does not have `TOK` values at the beginning and the end of the list, or if there are two adjacent `TM` values in the list. It will fail if the rule specifies a fixity with a precedence, and if that precedence level in the grammar is already taken by rules with a different sort of fixity.

Example

The traditional (now discontinued) HOL88/90 syntax for conditionals is `b => t | e`. With "dangling" terms (the `b` and the `e`) to the left and right, it is an infix (and one of very weak precedence at that).

```
val _ = add_rule{term_name = "COND",
                 fixity = Infix (HOLgrammars.RIGHT, 3),
                 pp_elements = [HardSpace 1, TOK "=>",
                                BreakSpace(1,0), TM,
                                BreakSpace(1,0), TOK "|",
                                HardSpace 1],
                 paren_style = OnlyIfNecessary,
                 block_style = (AroundEachPhrase,
                                (PP.INCONSISTENT, 0))};
```

The more familiar `if-then-else` syntax has a "dangling" term only to the right of the construction, so this rule's fixity is of type `Prefix`. (If the rule was made a `Closefix`, strings such as `` `if P then Q else R` `` would still parse, but so too would `` `if P then Q else` ``.) This example also illustrates the use of blocks within rules to improve pretty-printing.

```
val _ = add_rule{term_name = "COND", fixity = Prefix 70,
                 pp_elements = [PPBlock([TOK "if", BreakSpace(1,2),
                                         TM, BreakSpace(1,0),
                                         TOK "then"], (PP.CONSISTENT, 0)),
                                BreakSpace(1,2), TM, BreakSpace(1,0),
                                BeginFinalBlock(PP.CONSISTENT, 2),
                                TOK "else", BreakSpace(1,0)],
                 paren_style = OnlyIfNecessary,
```

```
                    block_style = (AroundEachPhrase,
                                   (PP.INCONSISTENT, 0))};
```

Note that the above form is not that actually used in the system. As written, it allows for pretty-printing some expressions as:

```
if P then
   <very long term> else Q
```

because the `block_style` is `INCONSISTENT`. The actual pretty-printer for if-then-else is a custom piece of code installed with `add_user_printer`. This handles nice printing of chained conditionals.

The pretty-printer prefers later rules over earlier rules by default (though this choice can be changed with `prefer_form_with_tok` (q.v.)), so if both of these calls were made, conditional expressions would print using the `if-then-else` syntax rather than the `_ => _ | _` syntax.

For making pretty concrete syntax possible.

Comments

Because adding new rules to the grammar may result in precedence conflicts in the operator-precedence matrix, it is as well with interactive use to test the `Term` parser immediately after adding a new rule, as it is only with this call that the precedence matrix is built.

As with other functions in the `Parse` structure, there is a companion `temp_add_rule` function, which has the same effect on the global grammar, but which does not cause this effect to persist when the current theory is exported.

An Isabelle-style concrete syntax for specifying rules would probably be desirable as it would conceal the complexity of the above from most users.

See also

`Parse.add_listform`, `Parse.add_infix`, `Parse.prefer_form_with_tok`, `Parse.remove_rules_for_term`

ADD_SGS_TAC (Tactical)

```
Tactical.ADD_SGS_TAC : term list -> tactic -> tactic
```

Adds extra sub-goals to the outcome of a tactic, as may be required to make an invalid tactic valid.

Suppose `tac` applied to the goal `(asl,w)` produces new goal list `gl`. Then `ADD_SGS_TAC tml tac (asl,w)` produces, as new goals, `gl` and, additionally, `(asl,tm)` for each `tm` in `tml`.

Then, if the justification returned by `tac`, when applied to the proved goals `gl`, gives a theorem which uses some additional assumption `tm` in `tml`, then the proved goal `(asl,tm)` is used to remove `tm` from the assumption list of that theorem.

Failure

`ADD_SGS_TAC tml tac (asl,w)` fails if `tac (asl,w)` fails.

Where a tactic `tac` requires certain assumptions `tml` to be present in the goal, which are not present but are capable of being proved, `ADD_SGS_TAC tml tac` will conveniently set up new subgoals to prove the missing assumptions.

Comments

`VALIDATE tac` is similar and will usually be easier to use, but its extra new subgoals may occur in an unpredictable order.

Example

Given a goal with an implication in the assumptions, one can split it into two subgoals, defining an auxiliary function

```
> fun split_imp_asm th =
    let val (tm, thu) = UNDISCH_TM th ;
    in ADD_SGS_TAC [tm] (ASSUME_TAC thu) end ;
val split_imp_asm = fn: thm -> Tactical.tactic
```

This can be used as follows:

```
val it =
   Proof manager status: 3 proofs.
   3. Incomplete goalstack:
```

```
        Initial goal:
        ∃R. WF R ∧ (∀rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)) ∧
            ∀rst x ord. R (ord,FILTER ($¬ ∘ ord x) rst) (ord,x::rst)

   2. Incomplete goalstack:
        Initial goal:
        1 + 2 = 2 + 1

        Current goal:
        ∀(x,y). x + y = y + x

   1. Incomplete goalstack:
        Initial goal:
         0.  p ⇒ q
        ------------------------------------
             r

> e (FIRST_X_ASSUM split_imp_asm) ;
OK..
2 subgoals:
val it =

    0.  q
   ------------------------------------
        r

   p
```

To propose a term, prove it as a subgoal and then use it to prove the goal, as is done using `SUBGOAL_THEN tm ASSUME_TAC`, can also be done by `ADD_SGS_TAC [tm] (ASSUME_TAC (ASSUME tm))`

See also

Tactic.impl_tac, `Tactical.VALIDATE`, `Tactical.GEN_VALIDATE`

add_strliteral_form (Parse)

```
Parse.add_strliteral_form : {inj:term, ldelim:string} -> unit
```

Adds interpretation for string literal syntaxes

If `ld` is a valid left delimiter, with corresponding right delimiter `rd`, then a call to `add_strliteral_form{inj=t,ldelim=ld}` causes the parser and pretty-printer to treat string literals delimited by `ld` and `rd` as occurrences of the term `inj` applied to the given HOL value (which will be of `string` type).

If the given `ld`-`rd` pair is already associated with an injector, then the parsing process will resolve the ambiguity with the standard overloading resolution method. In particular, note that the standard double quotation mark (ASCII character 34, `"`) is associated with the "null" injector, which takes string literals into the `string` type. Other injectors can be associated with this delimiter pair.

The other valid delimiter pairs are double guillemets (`«...»`, U+00AB and U+00BB) and single guillemets (`<...>`, U+2039 and U+203A).

Failure

Fails if the `ldelim` field does not correspond to a valid left delimiter, or if the HOL type of the `inj` field is not `:string->X` for some type `X`.

Example

If we have established a new type of deeply embedded terms with variables, constants and binary applications:

```
Datatype`tm = V string | Cst string | App tm tm`;
```

then we can overload the usual double-quoted string literals to also be applications of the `V` constructor:

```
> add_strliteral_form {inj=``V``, ldelim="\""};
> ``App (V "foo") (App "bar" "baz")``;
val it = “App "foo" (App "bar" "baz")”: term
```

where all the string literals in the output are actually applications of `V` to a real literal.

We can further choose to have constants printed with enclosing `«...»` by:

```
> add_strliteral_form {inj=``Cst``, ldelim="«"};
> ``App "foo" (Cst "bar")``;
val it = “App "foo" «bar»”: term
```

Note that in this situation, use of the double guillemets is unambiguous, but a bare string literal is strictly ambiguous (the default is to prefer the core string type):

```
> type_of “«foo»”;
val it = “:tm”: hol_type

> type_of “"foo"”;
<<HOL message: more than one resolution of overloading was possible>>
val it = “:string”: hol_type
```

Comments

This facility is analogous to the way in which numerals can be seen to inhabit types other than just `:num`. As with other parsing facilities there is a temporary form `temp_add_strliteral_form`, which does not cause the change to the grammar to persist to descendant theories.

The effect of adding a new string literal form can be reversed by parallel `remove_string_literal_form` and `temp_remove_string_literal_form` functions.

See also

`Parse.add_numeral_form`

add_tag (Thm)

```
Thm.add_tag : tag * thm -> thm
```

Adds oracle tags to a theorem.

A call to `add_tag(tg,th)` returns a `th'` such that calling `Thm.tag(th')` returns the tag that is the merge of the tag associated with `th` (if any) and `tg`.

Failure

Never fails.

Comments

If an oracle implementation wishes to record additional information about the oracle mechanisms that have contributed to the 'proof' of a theorem (perhaps the use of existing HOL theorems that will have their own tags), then this function can be used to add that record.

See also

`Thm.mk_oracle_thm`

add_user_printer (Parse)

```
Parse.add_user_printer : (string * term) -> unit
```

Adds a user specified pretty-printer for a specified type.

The function `add_user_printer` is used to add a special purpose term pretty-printer to the interactive system. The pretty-printer is called whenever the term to be printed matches (with `match_term`) the term provided as the second parameter. If multiple calls to `add_user_printer` are made with the same string parameter, the older functions are replaced entirely. If multiple printers match, the more specific match will be chosen. If two matches are equally specific, the match chosen is unspecified.

The function that performs the printing is not given directly, but is instead referred to by name (the first parameter to `add_user_printer`). This name must be linked to the desired code with a call to `term_grammar.userSyntaxFns.register_userPP`, which function should be called within another ML file (i.e., not the "Script'' file of the theory). The name passed to the `register_userPP` function is then the name that must also be passed to `add_user_printer`. The term argument is the desired pattern.

Alternatively, if the name specified is the empty string (`""`), the behaviour is to ensure that terms matching this pattern are not handled by the user-printing machinery. The

expectation is that a more general pattern has already been registered, but that in this specified scenario the general term-printing machinery should be used.

The user-supplied function may choose not to print anything for the given term and hand back control to the standard printer by raising the exception `term_pp_types.UserPP_Failed`. All other exceptions will propagate to the top-level. If the system printer receives the `UserPP_Failed` exception, it prints out the term using its standard algorithm, but will again attempt to call the user function on any sub-terms that match the pattern.

The type `userprinter` is an abbreviation defined in `term_grammar` to be

```
type userprinter =
  type_grammar.grammar * term_grammar.grammar ->
  PPBackend.t ->
  sysprinter ->
  term_pp_types.ppstream_funs ->
  (grav * grav * grav) -> int ->
  term -> uprinter
```

where the type `grav` (from `term_pp_types`) is

```
datatype grav = Top | RealTop | Prec of (int * string)
```

The type `uprinter` (standing for "unit printer'') is a special monadic printing type based on the `smpp` module (explained further in the example below). The type `sysprinter` is another abbreviation

```
type sysprinter =
  { gravs : (grav * grav * grav), binderp : bool,
    depth : int } -> term -> uprinter
```

Thus, when the user's printing function is called, it is passed ten parameters, including three "gravity'' values in a triple, and two grammars. The fourth parameter is the system's own printer. The fifth parameter is a record of functions to call for adding a string to the output, adding a break, adding new lines, defining some styles for printing like the color, etc. The availability of the system's printer allows the user function to use the default printer on sub-terms that it is not interested in. The user function must not call the `sysprinter` on the term that it is handed initially as the `sysprinter` will immediately call the user printing function all over again. If the user printer wants to give the whole term back to the system printer, then it must use the `UserPP_Failed` exception described above.

Though there are existing functions `add_string`, `add_break` etc. that can be used to create pretty-printing values, users should prefer instead to use the functions that are provided in the triple with the `sysprinter`. This then gives them access to functions that can prevent inadvertent symbol merges.

The `grav` type is used to let pretty-printers know a little about the context in which a term is to be printed out. The triple of gravities is given in the order "parent", "left" and "right". The left and right gravities specify the precedence of any operator that might be attempting to "grab" arguments from the left and right. For example, the term

```
(p /\ (if q then r else s)) ==> t
```

should be pretty-printed as

```
p /\ (if q then r else s) ==> t
```

The system figures this out when it comes to print the conditional expression because it knows both that the operator to the left has the appropriate precedence for conjunction but also that there is an operator with implication's precedence to the right. The issue arises because conjunction is tighter than implication in precedence, leading the printer to decide that parentheses aren't necessary around the conjunction. Similarly, considered on its own, the conjunction doesn't require parentheses around the conditional expression because there is no competition between them for arguments.

The `grav` constructors `Top` and `RealTop` indicate a context analogous to the top of the term, where there is no binding competition. The constructor `RealTop` is reserved for situations where the term really is the top of the tree; `Top` is used for analogous situations such when the term is enclosed in parentheses. (In the conditional expression above, the printing of `q` will have `Top` gravities to the left and right.)

The `Prec` constructor for gravity values takes both a number indicating precedence level and a string corresponding to the token that has this precedence level. This string parameter is of most importance in the parent gravity (the first component of the triple) where it can be useful in deciding whether or not to print parentheses and whether or not to begin fresh pretty-printing blocks. For example, tuples in the logic look better if they have parentheses around the topmost instance of the comma-operator, regardless of whether or not this is required according to precedence considerations. By examining the parent gravity, a printer can determine more about the term's context. (Note that the parent gravity will also be one or other of the left and right gravities; but it is not possible to tell which.)

The integer parameter to both the system printing function and the user printing function is the depth of the term. The system printer will stop printing a term if the depth ever

reaches exactly zero. Each time it calls itself recursively, the depth parameter is reduced by one. It starts out at the value stored in `Globals.max_print_depth`. Setting the latter to `~1` will ensure that all of a term is always printed.

The `binderp` parameter to the system-printer is true when the term to be printed should be considered as a binder. This makes a difference when the printer comes to print type annotations: annotations will occur with variables if the variable is in a binding position, and not elsewhere. This logic ensures that a term like `\x. x` prints as `\x:'a. x`. The first, binding occurrence of the variable gets an annotation; subsequent occurrences do not.

Failure

Fails if the string parameter does not correspond to a name used to register a function with `term_grammar.userSyntaxFns.register_userPP`. In addition, if the function parameter fails to print all terms of the registered type in any other way than raising the `UserPP_Failed` exception, then the pretty-printer will also fail.

Example

In the examples that follow, the companion `temp_add_userprinter` function is used: this function takes a value of type `userprinter` directly, and so is a more direct demonstration of how user-printers can behave. The boilerplate required for preserved-across-`export_theory` functionality starts with the external module. For a theory `fooScript.sml`, one might write the file `fooPP.sml`:

```
structure fooPP =
struct

  fun term_printer_code ... = ...
  val _ = term_grammar.userSyntaxFns.register_userPP {
            name = "foo.term_printer", code = term_printer_code
          }
end
```

where `term_printer_code` has type `userprinter`. Note that the code for the printer has to be written so that it can be compiled before the theory `foo` is in context. In particular, top-level calls to `mk_thy_const` and the like will fail if they attempt to bind constants declared in theory `foo`.

In `fooScript.sml`, the following is the idiom required:

```
local open fooPP in end;
val _ = add_ML_dependency "fooPP"
val _ = add_user_printer ("foo.term_printer", ``term pattern``)
```

As discussed, the remaining examples use `temp_add_user_printer`. The first example uses the system printer to print sub-terms, and concerns itself only with printing conjunctions. Note how the actions that make up the pretty-printer (combinations of `add_string` and `add_break` are combined with the infix >> operator (from the `smpp` module).

```
> fun myprint Gs B sys (ppfns:term_pp_types.ppstream_funs) gravs d t =
  let
    open Portable term_pp_types smpp
    val (str,brk) = (#add_string ppfns, #add_break ppfns);
    val (l,r) = dest_conj t
    fun syspr gravs =
      sys {gravs = gravs, depth = d - 1, binderp = false}
  in
    str "CONJ:" >>
    brk (1,0) >>
    syspr (Top, Top, Top) l >>
    brk (1,0) >> str "and then" >> brk(1,0) >>
    sys (Top, Top, Top) r >>
    str "ENDCONJ"
  end handle HOL_ERR _ => raise term_pp_types.UserPP_Failed;
val myprint = fn :
   'a -> 'b ->
   (grav * grav * grav -> int -> term ->
     (term_pp_types.printing_info,'c)smpp.t) ->
   term_pp_types.ppstream_funs -> 'd -> int -> term ->
   (term_pp_types.printing_info,unit)smpp.t

> temp_add_user_printer ("myprint", ``p /\ q``, myprint);
val it = () : unit

> ``p ==> q /\ r``;
val it = ``p ==> CONJ: q and then r ENDCONJ`` : term
```

The variables `p`, `q` and `r` as well as the implication are all of boolean type, but are handled by the system printer. The user printer handles just the special form of the conjunction. Note that this example actually falls within the scope of the `add_rule` functionality.

The next approach to printing conjunctions is not possible with `add_rule`. This example uses the styling and blocking functions to create part of its effect. These functions (`ustyle` and `ublock` respectively) are higher-order functions that take printers as arguments and cause the arguments to be printed with a particular governing style (`ustyle`), or indented to reveal block structure (`ublock`).

```
- fun myprint2 Gs B sys (ppfns:term_pp_types.ppstream_funs) (pg,lg,rg) d t =
  let
    open Portable term_pp_types PPBackEnd smpp
    val {add_string,add_break,ublock,ustyle,...} = ppfns
    val (l,r) = dest_conj t
    fun delim wrap body =
      case pg of
        Prec(_, "CONJ") => body
      | _ => wrap body
    fun syspr t =
      sys {gravs = (Prec(0,"CONJ"), Top, Top), depth = d - 1,
           binderp = false} t
  in
    delim (fn bod => ublock CONSISTENT 0
                        (ustyle [Bold] (add_string "CONJ") >>
                         add_break (1,2) >>
                         ublock INCONSISTENT 0 bod >>
                         add_break (1,0) >>
                         ustyle [Bold] (add_string "ENDCONJ")))
       (syspr l >> add_string "," >> add_break (1,0) >> syspr r)
  end handle HOL_ERR _ => raise term_pp_types.UserPP_Failed;

- temp_add_user_printer ("myprint2", ``p /\ q``, myprint2);

- ``p /\ q /\ r /\ s /\ t /\ u /\ p /\ p /\ p /\ p /\ p /\ p /\
    p /\ p /\ p /\ p/\ p /\ p /\ q /\ r /\ s /\ t /\ u /\ v /\
    (w /\ x) /\ (p \/ q) /\ r``;

> val it =
    ``CONJ
        p, q, r, s, t, u, p, p, p, p, p, p, p, p, p, p, p, q,
        r, s, t, u, v, w, x, p \/ q, r
      ENDCONJ`` : term
```

This example also demonstrates using parent gravities to print out a big term. The function passed as an argument to delim is only called when the parent gravity is not "CONJ". This ensures that the special delimiters only get printed when the first conjunction is encountered. Subsequent, internal conjunctions get passed the "CONJ" gravity in the calls to sys.

A better approach (and certainly a more direct one) would probably be to call strip_conj and print all of the conjuncts in one fell swoop. Additionally, this example demonstrates how easy it is to conceal genuine syntactic structure with a pretty-printer. Finally, it shows how styles can be used.

For extending the pretty-printer in ways not possible to encompass with the built-in grammar rules for concrete syntax.

See also

Parse.add_rule, Term.match_term, Parse.remove_user_printer

all (Lib)

```
Lib.all : ('a -> bool) -> 'a list -> bool
```

Tests whether a predicate holds throughout a list.

all P [x1,...,xn] equals P x1 andalso .... andalso P xn. all P [] yields true.

Failure

If P x0,...,P x(j-1) all evaluate to true and P xj raises an exception e, then all P [x0,...,x(j-1),xj,...,xn] raises e.

Example

```
> all (equal 3) [3,3,3];
val it = true: bool

> all (equal 3) [];
val it = true: bool
```

```
> all (fn _ => raise Fail "") [];
val it = true: bool

> all (fn _ => raise Fail "") [1];
Exception- Fail "" raised
```

See also

`Lib.all2`, `Lib.exists`, `Lib.first`

all2 (Lib)

```
Lib.all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool
```

Tests whether a predicate holds pairwise throughout two lists.

An invocation

```
all2 P [x1,...,xn] [y1,...,yn]
```

equals

```
P x1 y1 andalso .... andalso P xn yn
```

Also, `all2 P [] []` yields `true`.

Failure

If `P x0,...,P x(j-1)` all evaluate to `true` and `P xj` raises an exception `e`, then

```
all2 P [x0,...,x(j-1),xj,...,xn]
```

raises `e`. An invocation `all2 P l1 l2` will also raise an exception if the length of `l1` is not equal to the length of `l2`.

Example

```
> all2 equal [1,2,3] [1,2,3];
val it = true: bool

> all2 equal [1,2,3] [1,2,3,4] handle e => Raise e;
Exception- HOL_ERR (at Lib.all2: different length lists) raised

> all2 (fn _ => fn _ => raise Fail "") [] [];
val it = true: bool

> all2 (fn _ => fn _ => raise Fail "") [1] [1];
Exception- Fail "" raised
```

See also

`Lib.all`

all_atoms (Term)

```
Term.all_atoms : term -> term set
```

Returns variables and constants occurring in a term.

A call to `all_atoms t` will return the set of all the variables and constants that appear in a term. The variables include those that occur under binders, even if only in binding position. Multiple instances of the same (polymorphic) constant can occur in the result if those instances are present in the term.

Because bound variables are returned as part of the result, alpha-equivalent terms will not necessarily give the same results when `all_atoms` is applied to them.

Failure

Never fails.

Example

```
> HOLset.listItems (all_atoms ``!v. v /\ p``);
val it = [“p”, “v”, “$!”, “$/\”]: term list

> show_types := true;
val it = (): unit

> HOLset.listItems (all_atoms ``!v. v /\ !f. f v``);
val it =
   [“(f :bool -> bool)”, “(v :bool)”, “($! :(bool -> bool) -> bool)”,
    “($! :((bool -> bool) -> bool) -> bool)”, “$/\”]: term list

> HOLset.listItems (all_atoms ``!v:'a. T``);
val it = [“(v :α)”, “($! :(α -> bool) -> bool)”, “T”]: term list
```

Comments

There is a companion function `all_atomsl` taking an accumulator, which has type `term list -> term set -> term set`.

See also

`Term.all_vars`

all_consts (Term)

```
Term.all_consts : unit -> term list
```

All known constants in the current theory.

An invocation `all_consts` returns a list of all declared constants in the current theory, i.e., all constants in the current theory segment and in its ancestry.

Failure

Never fails.

Example

```
> all_consts();
val it =
   [“words$word_xor”, “words$word_xnor”, “words$word_to_oct_string”,
    “words$word_to_oct_list”, “words$word_to_hex_string”,
    “words$word_to_hex_list”, “words$word_to_dec_string”,
    “words$word_to_dec_list”, “words$word_to_bin_string”,
    “words$word_to_bin_list”, “words$word_sub”, “words$word_smin”,
    “words$word_smax”, “words$word_slice”, “words$word_signed_bits”,
    “words$word_sign_extend”, “words$word_rrx”, “words$word_ror_bv”,
    “words$word_ror”, “words$word_rol_bv”, “words$word_rol”,
    “words$word_reverse”, “words$word_replicate”, “words$word_rem”,
    “words$word_reduce”, “words$word_quot”, “words$word_or”,
    “words$word_nor”, “words$word_nand”, “words$word_mul”, “words$word_msb”,
    “words$word_modify”, “words$word_mod”, “words$word_min”,
    “words$word_max”, “words$word_lt”, “words$word_lsr_bv”, “words$word_lsr”,
    “words$word_lsl_bv”, “words$word_lsl”, “words$word_lsb”, “words$word_ls”,
    “words$word_log2”, “words$word_lo”, “words$word_len”, “words$word_le”,
    “words$word_join”, “words$word_hs”, “words$word_hi”, “words$word_gt”,
    “words$word_ge”, “words$word_from_oct_string”,
    “words$word_from_oct_list”, “words$word_from_hex_string”,
    “words$word_from_hex_list”, “words$word_from_dec_string”,
    “words$word_from_dec_list”, “words$word_from_bin_string”,
    “words$word_from_bin_list”, “words$word_extract”,
    “words$word_exp_tailrec”, “words$word_exp”, “words$word_div”,
    “words$word_concat”, “words$word_compare”, “words$word_bits”,
    “words$word_bit”, “words$word_asr_bv”, “words$word_asr”,
    “words$word_and”, “words$word_add”, “words$word_abs”, “words$word_T”,
    “words$word_L2”, “words$word_L”, “words$word_H”, “words$word_2comp”,
    “words$word_1comp”, “words$w2w”, “words$w2s”, “words$w2n”, “words$w2l”,
    “words$sw2sw”, “words$saturate_w2w”, “words$saturate_sub”,
    “words$saturate_n2w”, “words$saturate_mul”, “words$saturate_add”,
    “words$s2w”, “words$reduce_xor”, “words$reduce_xnor”, “words$reduce_or”,
    “words$reduce_nor”, “words$reduce_nand”, “words$reduce_and”,
    “words$nzcv”, “words$n2w_itself”, “words$n2w”, “words$l2w”,
    “words$dimword”, ...]: term list
```

See also

`Parse.term_grammar`

ALL_CONV (Conv)

```
Conv.ALL_CONV : conv
```

Conversion that always raises the `UNCHANGED` exception.

When applied to a term `t`, the conversion `ALL_CONV` raises the special `UNCHANGED` exception, which indicates to leave `t` unchanged.

Failure

Always raises the `UNCHANGED` exception.

Identity element for `THENC`.

See also

`Conv.UNCHANGED`, `Conv.NO_CONV`, `Thm.REFL`

ALL_EL_CONV (listLib)

```
listLib.ALL_EL_CONV : conv -> conv
```

Computes by inference the result of applying a predicate to elements of a list.

`ALL_EL_CONV` takes a conversion `conv` and a term `tm` in the following form:

```
ALL_EL P [x0;...xn]
```

It returns the theorem

```
|- ALL_EL P [x0;...xn] = T
```

if for every `xi` occurring in the list, `conv "P xi"` returns a theorem `|- P xi = T`, otherwise, if for at least one `xi`, evaluating `conv "P xi"` returns the theorem `|- P xi = F`, then it returns the theorem

```
|- ALL_EL P [x0;...xn] = F
```

Failure

`ALL_EL_CONV conv tm` fails if `tm` is not of the form described above, or failure occurs when evaluating `conv "P xi"` for some `xi`.

Example

Evaluating

```
ALL_EL_CONV bool_EQ_CONV "ALL_EL ($= T) [T;F;T]";
```

returns the following theorem:

```
|- ALL_EL($= T)[T;F;T] = F
```

In general, if the predicate `P` is an explicit lambda abstraction `(\x. P x)`, the conversion should be in the form

```
(BETA_CONV THENC conv')
```

See also

`listLib.SOME_EL_CONV`, `listLib.IS_EL_CONV`, `listLib.FOLDL_CONV`, `listLib.FOLDR_CONV`, `listLib.list_FOLD_CONV`

ALL_LT (Tactical)

```
Tactical.ALL_LT : list_tactic
```

Passes on a goal list unchanged.

`ALL_LT` applied to a goal list `gl` simply produces the goal list `gl`. It is the identity for the `THEN_LT` tactical.

Failure

Never fails.

Example

To apply tactic `tac1` to a goal, and then to apply `tac2` to all resulting subgoals except the first, use

```
tac1 THEN_LT SPLIT_LT 1 (ALL_LT, ALLGOALS tac2)
```

See also

`Tactical.THEN_LT`, `Tactical.SPLIT_LT`, `Tactical.ALLGOALS`

all_monads (monadsyntax)

```
monadsyntax.all_monads :
  unit ->
  (string *
   {bind : term, unit : term, ignorebind : term option,
    choice : term option, fail : term option, guard : term option}) list
```

Lists all declared monads

Returns a list of all declared monad types. These can be enabled with calls to `enable_monad`.

Failure

Never fails.

Example

```
> monadsyntax.all_monads();
val it =
   [("list",
     {bind = “LIST_BIND”, choice = SOME (“$++”), fail = SOME (“[]”), guard =
      SOME (“LIST_GUARD”), ignorebind = SOME (“LIST_IGNORE_BIND”), unit =
      “λx. [x]”}),
    ("option",
     {bind = “OPTION_BIND”, choice = SOME (“OPTION_CHOICE”), fail =
      SOME (“NONE”), guard = SOME (“OPTION_GUARD”), ignorebind =
      SOME (“OPTION_IGNORE_BIND”), unit = “SOME”})]:
   (string * monadsyntax.monadinfo) list
```

See also

monadsyntax.declare_monad, monadsyntax.enable_monad

ALL_TAC (Tactical)

```
Tactical.ALL_TAC : tactic
```

Passes on a goal unchanged.

ALL_TAC applied to a goal g simply produces the subgoal list [g]. It is the identity for the THEN tactical.

Failure

Never fails.

Example

The tactic

```
INDUCT_THEN numTheory.INDUCTION THENL [ALL_TAC, tac]
```

applied to a goal g, applies INDUCT_THEN numTheory.INDUCTION to g to give a basis and step subgoal; it then returns the basis unchanged, along with the subgoals produced by applying tac to the step.

Used to write tacticals such as REPEAT. Also, it is often used as a place-holder in building compound tactics using tacticals such as THENL.

See also

Prim_rec.INDUCT_THEN, Tactical.NO_TAC, Tactical.REPEAT, Tactical.THENL

ALL_THEN (Thm_cont)

```
Thm_cont.ALL_THEN : thm_tactical
```

Passes a theorem unchanged to a theorem-tactic.

For any theorem-tactic ttac and theorem th, the application ALL_THEN ttac th results simply in ttac th, that is, the theorem is passed unchanged to the theorem-tactic. ALL_THEN is the identity theorem-tactical.

Failure

The application of ALL_THEN to a theorem-tactic never fails. The resulting theorem-tactic fails under exactly the same conditions as the original one.

Writing compound tactics or tacticals, e.g. terminating list iterations of theorem-tacticals.

See also

Tactical.ALL_TAC, Tactical.FAIL_TAC, Tactical.NO_TAC, Thm_cont.NO_THEN, Thm_cont.THEN_TCL, Thm_cont.ORELSE_TCL

all_vars (Term)

```
Term.all_vars : term -> term list
```

Returns the set of all variables in a term.

An invocation `all_vars tm` returns a list representing the set of all bound and free term variables occurring in `tm`.

Failure

Never fails.

Example

```
> all_vars ``!x y. x /\ y /\ y ==> z``;
val it = [“z”, “y”, “x”]: term list
```

Comments

Code should not depend on how elements are arranged in the result of `all_vars`.

See also

`Term.all_atoms`, `Term.all_varsl`, `Term.free_vars`

all_varsl (Term)

```
Term.all_varsl : term list -> term list
```

Returns the set of all variables in a list of terms.

An invocation `all_varsl [t1,...,tn]` returns a list representing the set of all term variables occurring in `t1,...,tn`.

Failure

Never fails.

Example

```
> all_varsl [Term `x /\ y /\ y ==> x`,
             Term `!a. a ==> p ==> y`];
val it = [“x”, “y”, “p”, “a”]: term list
```

Comments

Code should not depend on how elements are arranged in the result of `all_varsl`.

See also

`Term.all_atoms`, `Term.all_vars`, `Term.empty_varset`, `Term.free_vars_lr`, `Term.free_vars`, `Term.free_varsl`, `Term.FVL`, `Type.type_vars`

ALLGOALS (Tactical)

```
Tactical.ALLGOALS : tactic -> list_tactic
```

Applies a tactic to every goal in a list

If `tac` is a tactic, `ALLGOALS tac` is a list-tactic which applies the tactic `tac` to all the goals in the given list.

Failure

The application of `ALLGOALS` to a tactic never fails. The resulting list-tactic fails if `tac` fails when applied to any of the goals in the given list.

Example

Where `tac1` and `tac2` are tactics, `tac1 THEN_LT ALLGOALS tac2` is equivalent to `tac1 THEN tac2`

See also

`Tactical.THEN_LT`, `Tactical.THEN`

allowed_term_constant (Lexis)

```
Lexis.allowed_term_constant : string -> bool
```

Tests if a string has a permissible name for a term constant.

When applied to a string, `allowed_term_constant` returns `true` if the string is a permissible constant name for a term, that is, if it is an identifier (see the DESCRIPTION for more details), and `false` otherwise.

Failure

Never fails.

Example

The following gives a sample of some allowed and disallowed constant names:

```
- map Lexis.allowed_term_constant ["pi", "@", "a name", "+++++", "10"];
> val it = [true, true, false, true, false] : bool list
```

Comments

Note that this function only performs a lexical test; it does not check whether there is already a constant of that name in the current theory.

See also

`Theory.constants`, `Lexis.allowed_type_constant`

allowed_type_constant (Lexis)

```
Lexis.allowed_type_constant : string -> bool
```

Tests if a string has a permissible name for a type constant.

When applied to a string, `allowed_type_constant` returns `true` if the string is a permissible constant name for a type operator, and `false` otherwise.

Failure

Never fails.

Example

The following gives a sample of some allowed and disallowed names for type operators:

```
- map Lexis.allowed_type_constant ["list", "'a", "fun", "->", "#", "fun2"];
> val it = [true, false, true, false, false, true] : bool list
```

Comments

Note that this function only performs a lexical test; it does not check whether there is already a type operator of that name in the current theory.

This function is not currently enforced by the system, as it was found that more flexibilty in naming was preferable.

See also

`Lexis.allowed_term_constant`

ALPHA (Thm)

```
Thm.ALPHA : term -> term -> thm
```

Proves equality of alpha-equivalent terms.

When applied to a pair of terms `t1` and `t1'` which are alpha-equivalent, `ALPHA` returns the theorem `|- t1 = t1'`.

```
-------------  ALPHA  t1  t1'
 |- t1 = t1'
```

Failure

Fails unless the terms provided are alpha-equivalent.

See also

`Term.aconv`, `Drule.ALPHA_CONV`, `Drule.GEN_ALPHA_CONV`

alpha (Type)

```
Type.alpha : hol_type
```

Common type variable.

The ML variable `Type.alpha` is bound to the type variable `'a`.

See also

`Type.beta`, `Type.gamma`, `Type.delta`, `Type.bool`

ALPHA_CONV (Drule)

```
Drule.ALPHA_CONV : term -> conv
```

Renames the bound variable of a lambda-abstraction.

If `x` is a variable of type `ty` and `M` is an abstraction (with bound variable `y` of type `ty` and body `t`), then `ALPHA_CONV x M` returns the theorem:

```
|- (\y.t) = (\x'. t[x'/y])
```

where the variable `x':ty` is a primed variant of `x` chosen so as not to be free in `\y.t`.

Failure

`ALPHA_CONV x tm` fails if `x` is not a variable, if `tm` is not an abstraction, or if `x` is a variable `v` and `tm` is a lambda abstraction `\y.t` but the types of `v` and `y` differ.

See also

`Thm.ALPHA`, `Drule.GEN_ALPHA_CONV`

ancestry (Theory)

```
Theory.ancestry : string -> string list
```

Returns the (proper) ancestry of a theory in a list.

A call to `ancestry thy` returns a list of all the proper ancestors (i.e. parents, parents of parents, etc.) of the theory `thy`. The shorthand `"-"` may be used to denote the name of the current theory segment.

Failure

Fails if `thy` is not an ancestor of the current theory.

Example

```
> load "bossLib";
val it = (): unit

> current_theory();
val it = "wombat": string

> ancestry "-";
val it =
   ["hrat", "hreal", "realax", "real_arith", "real", "sorting", "scratch",
    "numeral_bit", "permutes", "iterate", "fcp", "sum_num", "logroot", "bit",
    "numposrep", "ternaryComparisons", "string", "ASCIInumbers", "words",
```

```
"bitstring", "ConseqConv", "quantHeuristics", "patternMatches",
"ind_type", "divides", "While", "cv", "reduce", "one", "sum", "option",
"numeral", "basicSize", "numpair", "pred_set", "list", "rich_list",
"indexedLists", "hol", "quotient", "pair", "combin", "sat",
"normalForms", "relation", "min", "bool", "marker", "num", "prim_rec",
"arithmetic", "normalizer", "integer", "example"]: string list
```

See also

`Theory.parents`

AND_CONV (reduceLib)

```
reduceLib.AND_CONV : conv
```

Simplifies certain boolean conjunction expressions.

If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `AND_CONV tm` returns the corresponding theorem. Note that in the last case the conjuncts need only be alpha-equivalent rather than strictly identical.

```
AND_CONV "T /\ t" = |- T /\ t = t
AND_CONV "t /\ T" = |- t /\ T = t
AND_CONV "F /\ t" = |- F /\ t = F
AND_CONV "t /\ F" = |- t /\ F = F
AND_CONV "t /\ t" = |- t /\ t = t
```

Failure

`AND_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
#AND_CONV "(x = T) /\ F";;
|- (x = T) /\ F = F
```

```
#AND_CONV "T /\ (x = T)";;
|- T /\ (x = T) = (x = T)

#AND_CONV "(?x. x=T) /\ (?y. y=T)";;
|- (?x. x = T) /\ (?y. y = T) = (?x. x = T)
```

AND_EL_CONV (listLib)

```
listLib.AND_EL_CONV : conv
```

Computes by inference the result of taking the conjunction of the elements of a boolean list.

For any object language list of the form "`[x1;x2;...;xn]`", where `x1`, `x2`, ..., `xn` are boolean expressions, the result of evaluating

```
AND_EL_CONV "AND_EL [x1;x2;...;xn]"
```

is the theorem

```
|- AND_EL [x1;x2;...;xn] = b
```

where `b` is either the boolean constant that denotes the conjunction of the elements of the list, or a conjunction of those `xi` that are not boolean constants.

Example

```
> listLib.AND_EL_CONV "AND_EL [T;F;F;T]";
val it = ⊢ AND_EL [T; F; F; T] ⇔ F: thm

> listLib.AND_EL_CONV "AND_EL [T;T;T]";
val it = ⊢ AND_EL [T; T; T] ⇔ T: thm

> listLib.AND_EL_CONV "AND_EL [T;x;y]";
val it = ⊢ AND_EL [T; x; y] ⇔ x ∧ y: thm

> listLib.AND_EL_CONV "AND_EL [x;F;y]";
val it = ⊢ AND_EL [x; F; y] ⇔ F: thm
```

Failure

AND_EL_CONV tm fails if tm is not of the form described above.

AND_EXISTS_CONV (Conv)

```
Conv.AND_EXISTS_CONV : conv
```

Moves an existential quantification outwards through a conjunction.

When applied to a term of the form (?x.P) /\ (?x.Q), where x is free in neither P nor Q, AND_EXISTS_CONV returns the theorem:

```
|- (?x. P) /\ (?x. Q) = (?x. P /\ Q)
```

Failure

AND_EXISTS_CONV fails if it is applied to a term not of the form (?x.P) /\ (?x.Q), or if it is applied to a term (?x.P) /\ (?x.Q) in which the variable x is free in either P or Q.

Comments

It may be easier to use higher order rewriting with some of BOTH_EXISTS_AND_THM, LEFT_EXISTS_AND_THM, and RIGHT_EXISTS_AND_THM.

See also

Conv.EXISTS_AND_CONV, Conv.LEFT_AND_EXISTS_CONV, Conv.RIGHT_AND_EXISTS_CONV

AND_FORALL_CONV (Conv)

```
Conv.AND_FORALL_CONV : conv
```

Moves a universal quantification outwards through a conjunction.

When applied to a term of the form `(!x.P) /\ (!x.Q)`, the conversion `AND_FORALL_CONV` returns the theorem:

```
|- (!x.P) /\ (!x.Q) = (!x. P /\ Q)
```

Failure

Fails if applied to a term not of the form `(!x.P) /\ (!x.Q)`.

Comments

It may be easier to use higher order rewriting with `FORALL_AND_THM`.

See also

`Conv.FORALL_AND_CONV`, `Conv.LEFT_AND_FORALL_CONV`, `Conv.RIGHT_AND_FORALL_CONV`

AND_PEXISTS_CONV (PairRules)

```
PairRules.AND_PEXISTS_CONV : conv
```

Moves a paired existential quantification outwards through a conjunction.

When applied to a term of the form `(?p. t) /\ (?p. u)`, where no variables in `p` are free in either `t` or `u`, `AND_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) /\ (?p. u) = (?p. t /\ u)
```

Failure

`AND_PEXISTS_CONV` fails if it is applied to a term not of the form `(?p. t) /\ (?p. u)`, or if it is applied to a term `(?p. t) /\ (?p. u)` in which variables from `p` are free in either `t` or `u`.

See also

Conv.AND_EXISTS_CONV, PairRules.PEXISTS_AND_CONV, PairRules.LEFT_AND_PEXISTS_-CONV, PairRules.RIGHT_AND_PEXISTS_CONV

AND_PFORALL_CONV (PairRules)

```
PairRules.AND_PFORALL_CONV : conv
```

Moves a paired universal quantification outwards through a conjunction.

When applied to a term of the form `(!p. t) /\ (!p. t)`, the conversion AND_PFORALL_-CONV returns the theorem:

```
|- (!p. t) /\ (!p. u) = (!p. t /\ u)
```

Failure

Fails if applied to a term not of the form `(!p. t) /\ (!p. t)`.

See also

Conv.AND_FORALL_CONV, PairRules.PFORALL_AND_CONV, PairRules.LEFT_AND_PFORALL_-CONV, PairRules.RIGHT_AND_PFORALL_CONV

ANTE_CONJ_CONV (Conv)

```
Conv.ANTE_CONJ_CONV : conv
```

Eliminates a conjunctive antecedent in favour of implication.

When applied to a term of the form `(t1 /\ t2) ==> t`, the conversion ANTE_CONJ_CONV returns the theorem:

```
|- (t1 /\ t2 ==> t) = (t1 ==> t2 ==> t)
```

Failure

Fails if applied to a term not of the form `"(t1 /\ t2) ==> t"`.

Somewhat ad-hoc, but can be used (with `CONV_TAC`) to transform a goal of the form `?- (P /\ Q) ==> R` into the subgoal `?- P ==> (Q ==> R)`, so that only the antecedent `P` is moved into the assumptions by `DISCH_TAC`.

See also

`Tactic.CONV_TAC`, `Tactic.DISCH_TAC`

ANTE_RES_THEN (Thm_cont)

`Thm_cont.ANTE_RES_THEN : thm_tactical`

Resolves implicative assumptions with an antecedent.

Given a theorem-tactic `ttac` and a theorem `A |- t`, the function `ANTE_RES_THEN` produces a tactic that attempts to match `t` to the antecedent of each implication

```
Ai |- !x1...xn. ui ==> vi
```

(where `Ai` is just `!x1...xn. ui ==> vi`) that occurs among the assumptions of a goal. If the antecedent `ui` of any implication matches `t`, then an instance of `Ai u A |- vi` is obtained by specialization of the variables `x1`, ..., `xn` and type instantiation, followed by an application of modus ponens. Because all implicative assumptions are tried, this may result in several modus-ponens consequences of the supplied theorem and the assumptions. Tactics are produced using `ttac` from all these theorems, and these tactics are applied in sequence to the goal. That is,

```
ANTE_RES_THEN ttac (A |- t) g
```

has the effect of:

```
MAP_EVERY ttac [A1 u A |- v1, ..., Am u A |- vm] g
```

where the theorems `Ai u A |- vi` are all the consequences that can be drawn by a (single) matching modus-ponens inference from the implications that occur among the assumptions of the goal `g` and the supplied theorem `A |- t`. Any negation `~v` that appears among the assumptions of the goal is treated as an implication `v ==> F`. The sequence in which the theorems `Ai u A |- vi` are generated and the corresponding tactics applied is unspecified.

Failure

`ANTE_RES_THEN ttac (A |- t)` fails when applied to a goal `g` if any of the tactics produced by `ttac (Ai u A |- vi)`, where `Ai u A |- vi` is the `i`th resolvent obtained from the theorem `A |- t` and the assumptions of `g`, fails when applied in sequence to `g`.

Painfully detailed proof hacking.

See also

`Tactic.IMP_RES_TAC`, `Thm_cont.IMP_RES_THEN`, `Drule.MATCH_MP`, `Tactic.RES_TAC`, `Thm_cont.RES_THEN`

AP_TERM (Thm)

```
Thm.AP_TERM : term -> thm -> thm
```

Applies a function to both sides of an equational theorem.

When applied to a term `f` and a theorem `A |- x = y`, the inference rule `AP_TERM` returns the theorem `A |- f x = f y`.

```
    A |- x = y
----------------  AP_TERM f
 A |- f x = f y
```

Failure

Fails unless the theorem is equational and the supplied term is a function whose domain type is the same as the type of both sides of the equation.

See also

`Tactic.AP_TERM_TAC`, `Thm.AP_THM`, `Tactic.AP_THM_TAC`, `Thm.MK_COMB`

AP_TERM_TAC (Tactic)

```
Tactic.AP_TERM_TAC : tactic
```

Strips a function application from both sides of an equational goal.

`AP_TERM_TAC` reduces a goal of the form `A ?- f x = f y` by stripping away the function applications, giving the new goal `A ?- x = y`.

```
    A ?- f x = f y
  ================  AP_TERM_TAC
     A ?- x = y
```

Failure

Fails unless the goal is equational, with both sides being applications of the same function.

See also

`Thm.AP_TERM`, `Thm.AP_THM`, `Tactic.AP_THM_TAC`, `Tactic.MK_COMB_TAC`, `Tactic.ABS_TAC`

AP_THM (Thm)

```
Thm.AP_THM : thm -> term -> thm
```

Proves equality of equal functions applied to a term.

When applied to a theorem `A |- f = g` and a term `x`, the inference rule `AP_THM` returns the theorem `A |- f x = g x`.

```
      A |- f = g
  ----------------  AP_THM (A |- f = g) x
   A |- f x = g x
```

Failure

Fails unless the conclusion of the theorem is an equation, both sides of which are functions whose domain type is the same as that of the supplied term.

See also

`Tactic.AP_THM_TAC`, `Thm.AP_TERM`, `Drule.ETA_CONV`, `Drule.EXT`, `Conv.FUN_EQ_CONV`, `Thm.MK_COMB`

AP_THM_TAC (Tactic)

```
Tactic.AP_THM_TAC : tactic
```

Strips identical operands from functions on both sides of an equation.

When applied to a goal of the form `A ?- f x = g x`, the tactic `AP_THM_TAC` strips away the operands of the function application:

```
 A ?- f x = g x
================  AP_THM_TAC
   A ?- f = g
```

Failure

Fails unless the goal has the above form, namely an equation both sides of which consist of function applications to the same arguments.

See also

`Thm.AP_TERM`, `Tactic.AP_TERM_TAC`, `Thm.AP_THM`, `Tactic.MK_COMB_TAC`, `Tactic.ABS_TAC`, `Drule.EXT`

append (Lib)

```
Lib.append : 'a list -> 'a list -> 'a list
```

Curried form of list append.

The function `append` is a curried form of the standard operation for appending two ML lists.

Failure

Never fails.

Example

```
> append [1] [2,3] = [1] @ [2,3];
val it = true: bool
```

APPEND_CONV (listLib)

```
listLib.APPEND_CONV : conv
```

Computes by inference the result of appending two object-language lists.

For any pair of object language lists of the form "`[x1;...;xn]`" and "`[y1;...;ym]`", the result of evaluating

```
APPEND_CONV "APPEND [x1;...;xn] [y1;...;ym]"
```

is the theorem

```
|- APPEND [x1;...;xn] [y1;...;ym] = [x1;...;xn;y1;...;ym]
```

The length of either list (or both) may be 0.

Failure

`APPEND_CONV tm` fails if `tm` is not of the form "`APPEND l1 l2`", where `l1` and `l2` are (possibly empty) object-language lists of the forms "`[x1;...;xn]`" and "`[y1;...;ym]`".

apply (Count)

```
Count.apply : ('a -> 'b) -> 'a -> 'b
```

Counts primitive inferences performed when a function is applied.

The `apply` function provides a way of counting the primitive inferences that are performed when a function is applied to its argument. The reporting of the count is done when the function terminates (normally, or with an exception). The reporting also includes timing information about the function call.

Example

```
> Count.apply (CONJUNCTS o SPEC_ALL) AND_CLAUSES;
runtime: 0.00001s,    gctime: 0.00000s,     systime: 0.00000s.
Axioms: 0, Defs: 0, Disk: 0, Orcl: 0, Prims: 9; Total: 9
val it = [⊢ T ∧ t ⇔ t, ⊢ t ∧ T ⇔ t, ⊢ F ∧ t ⇔ F, ⊢ t ∧ F ⇔ F, ⊢ t ∧ t ⇔ t]:
   thm list
```

Failure

The call to `apply f x` will raise an exception if `f x` would. It will still report elapsed time and inference counts up to the point of the exception being raised.

See also

`Count.thm_count`

apropos (DB)

```
DB.apropos : term -> data list
```

Attempt to find matching theorems in the currently loaded theories.

An invocation `DB.apropos M` collects all theorems, definitions, and axioms of the currently loaded theories that have a subterm that matches `M`. If there are no matches, the empty list is returned.

Failure

Never fails.

Example

```
> DB.apropos (Term `(!x y. P x y) ==> Q`);
<<HOL message: inventing new type variable names: 'a, 'b>>
val it =
   [(("bool", "LCOMM_THM"),
     (⊢ ∀f. (∀x y z. f x (f y z) = f (f x y) z) ⇒
            (∀x y. f x y = f y x) ⇒
            ∀x y z. f x (f y z) = f y (f x z), Thm,
      Located
       {exact = true, linenum = 4225, scriptpath =
        "$(HOLDIR)/src/bool/boolScript.sml"})),
    (("ind_type", "INJ_INVERSE2"),
     (⊢ ∀P. (∀x1 y1 x2 y2. P x1 y1 = P x2 y2 ⇔ x1 = x2 ∧ y1 = y2) ⇒
            ∃X Y. ∀x y. X (P x y) = x ∧ Y (P x y) = y, Thm,
      Located
       {exact = true, linenum = 20, scriptpath =
        "$(HOLDIR)/src/datatype/ind_typeScript.sml"})),
[...Output elided...]
```

Comments

The notion of matching is a restricted version of higher-order matching.

For finer control over the theories searched, use `DB.match`.

See also

`DB.match`, `DB.find`, `DB.apropos_in`, `DB.matches`

apropos_in (DB)

```
DB.apropos_in : term -> data list -> data list
```

Attempt to select matching theorems among a given list.

An invocation `DB.apropos_in M data_list` selects all theorems, definitions, and axioms within `data_list` that have a subterm that matches `M`. If there are no matches, the empty list is returned.

Failure

Never fails.

Example

```
> DB.apropos (Term `(!x y. P x y) ==> Q`);
<<HOL message: inventing new type variable names: 'a, 'b>>
val it =
   [(("bool", "LCOMM_THM"),
     (⊢ ∀f. (∀x y z. f x (f y z) = f (f x y) z) ⇒
            (∀x y. f x y = f y x) ⇒
            ∀x y z. f x (f y z) = f y (f x z), Thm,
      Located
       {exact = true, linenum = 4225, scriptpath =
        "$(HOLDIR)/src/bool/boolScript.sml"})),
    (("ind_type", "INJ_INVERSE2"),
     (⊢ ∀P. (∀x1 y1 x2 y2. P x1 y1 = P x2 y2 ⇔ x1 = x2 ∧ y1 = y2) ⇒
            ∃X Y. ∀x y. X (P x y) = x ∧ Y (P x y) = y, Thm,
      Located
       {exact = true, linenum = 20, scriptpath =
        "$(HOLDIR)/src/datatype/ind_typeScript.sml"})),
[...Output elided...]

> DB.apropos_in (Term `(x, y)`) it ;
  [(("pair", "pair_induction"),
    (|- (!p_1 p_2. P (p_1,p_2)) ==> !p. P p, Thm))] :
```

```
  ((string * string) * (thm * class)) list
Exception- unknown symbol .
unknown symbol .
Fail "Static Errors" raised
```

Comments

The notion of matching is a restricted version of higher-order matching. It uses `DB.matches`.

Finding theorems in interactive proof sessions. The second argument will normally be the result of a previous call to `DB.find`, `DB.match`, `DB.apropos`, `DB.listDB`, `DB.thy` etc.

See also

`DB.apropos`, `DB.match`, `DB.matches`, `DB.find`, `DB.find_in`, `DB.listDB`, `DB.thy`, `DB.theorems`

arb (boolSyntax)

```
boolSyntax.arb : term
```

Constant denoting arbitrary items.

The ML variable `boolSyntax.arb` is bound to the term `bool$ARB`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.bool_case`

ARITH_CONV (numLib)

```
numLib.ARITH_CONV : conv
```

Partial decision procedure for a subset of linear natural number arithmetic.

ARITH_CONV is a partial decision procedure for Presburger natural arithmetic. Presburger natural arithmetic is the subset of arithmetic formulae made up from natural number constants, numeric variables, addition, multiplication by a constant, the relations <, <=, =, >=, > and the logical connectives ~, /\, \/, ==>, = (if-and-only-if), ! ('forall') and ? ('there exists'). Products of two expressions which both contain variables are not included in the subset, but the functions SUC and PRE which are not normally included in a specification of Presburger arithmetic are allowed in this HOL implementation.

ARITH_CONV further restricts the subset as follows: when the formula has been put in prenex normal form it must contain only one kind of quantifier, that is the quantifiers must either all be universal ('forall') or all existential. Variables may appear free (unquantified) provided any quantifiers that do appear in the prenex normal form are universal; free variables are taken as being implicitly universally quantified so mixing them with existential quantifiers would violate the above restriction.

Given a formula in the permitted subset, ARITH_CONV attempts to prove that it is equal to T (true). For universally quantified formulae the procedure only works if the formula would also be true of the non-negative rationals; it cannot prove formulae whose truth depends on the integral properties of the natural numbers. The procedure is also incomplete for existentially quantified formulae, but in this case there is no rule-of-thumb for determining whether the procedure will work.

The function features a number of preprocessors which extend the coverage beyond the subset specified above. In particular, natural number subtraction and conditional statements are allowed. Another permits substitution instances of universally quantified formulae to be accepted. Note that Boolean-valued variables are not allowed.

Failure

The function can fail in two ways. It fails if the argument term is not a formula in the specified subset, and it also fails if it is unable to prove the formula. The failure strings are different in each case. However, the function may announce that it is unable to prove a formula that one would expect it to reject as being outside the subset. This is due to it looking for substitution instances; it has generalised the formula so that the new formula is in the subset but is not valid.

Example

A simple example containing a free variable:

```
- ARITH_CONV ``m < SUC m``;
> val it = |- m < (SUC m) = T : thm
```

A more complex example with subtraction and universal quantifiers, and which is not initially in prenex normal form:

```
- ARITH_CONV
  ``!m p. p < m ==> !q r. (m < (p + q) + r) ==> ((m - p) < q + r)``;
> val it = |- (!m p. p < m ==> (!q r. m < ((p + q) + r) ==> (m - p) < (q + r))) = T
```

Two examples with existential quantifiers:

```
- ARITH_CONV ``?m n. m < n``;
> val it = |- (?m n. m < n) = T

- ARITH_CONV ``?m n. (2 * m) + (3 * n) = 10``;
> val it = |- (?m n. (2 * m) + (3 * n) = 10) = T
```

An instance of a universally quantified formula involving a conditional statement and subtraction:

```
- ARITH_CONV
  ``((p + 3) <= n) ==> (!m. ((m EXP 2 = 0) => (n - 1) | (n - 2)) > p)``;
> val it = |- (p + 3) <= n ==> (!m. ((m EXP 2 = 0) => n - 1 | n - 2) > p) = T
```

Failure due to mixing quantifiers:

```
- ARITH_CONV ``!m. ?n. m < n``;
evaluation failed     ARITH_CONV -- formula not in the allowed subset
```

Failure because the truth of the formula relies on the fact that the variables cannot have fractional values:

```
- ARITH_CONV ``!m n. ~(SUC (2 * m) = 2 * n)``;
evaluation failed     ARITH_CONV -- cannot prove formula
```

See also

Arith.NEGATE_CONV, Arith.EXISTS_ARITH_CONV, Arith.FORALL_ARITH_CONV, Arith.INSTANCE_T_CONV, Arith.PRENEX_CONV, Arith.SUB_AND_COND_ELIM_CONV

ARITH_FORM_NORM_CONV (Arith)

```
Arith.ARITH_FORM_NORM_CONV : conv
```

Normalises an unquantified formula of linear natural number arithmetic.

ARITH_FORM_NORM_CONV converts a formula of natural number arithmetic into a disjunction of conjunctions of less-than-or-equal-to inequalities. The arithmetic expressions are only allowed to contain natural number constants, numeric variables, addition, the SUC function, and multiplication by a constant. The formula must not contain quantifiers, but may have disjunction, conjunction, negation, implication, equality on Booleans (if-and-only-if), and the natural number relations: <, <=, =, >=, >. The formula must not contain products of two expressions which both contain variables.

The inequalities in the result are normalised so that each variable appears on only one side of the inequality, and each side is a linear sum in which any constant appears first followed by products of a constant and a variable. The variables are ordered lexicographically, and if the coefficient of the variable is 1, the product of 1 and the variable appears in the term rather than the variable on its own.

Failure

The function fails if the argument term is not a formula in the specified subset.

Example

```
#ARITH_FORM_NORM_CONV "m < n";;
|- m < n = (1 + (1 * m)) <= (1 * n)

#ARITH_FORM_NORM_CONV
# "(n < 4) ==> ((n = 0) \/ (n = 1) \/ (n = 2) \/ (n = 3))";;
|- n < 4 ==> (n = 0) \/ (n = 1) \/ (n = 2) \/ (n = 3) =
```

```
4 <= (1 * n) \/
(1 * n) <= 0 /\ 0 <= (1 * n) \/
(1 * n) <= 1 /\ 1 <= (1 * n) \/
(1 * n) <= 2 /\ 2 <= (1 * n) \/
(1 * n) <= 3 /\ 3 <= (1 * n)
```

Useful in constructing decision procedures for linear arithmetic.

arith_ss (bossLib)

`bossLib.arith_ss : simpset`

Simplification set for arithmetic.

The simplification set `arith_ss` is a version of `std_ss` enhanced for arithmetic. It includes many arithmetic rewrites, an evaluation mechanism for ground arithmetic terms, and a decision procedure for linear arithmetic. It also incorporates a cache of successfully solved conditions proved when conditional rewrite rules are successfully applied.

The following rewrites are currently used to augment those already present from `std_ss`:

```
|- !m n. (m * n = 0) = (m = 0) \/ (n = 0)
|- !m n. (0 = m * n) = (m = 0) \/ (n = 0)
|- !m n. (m + n = 0) = (m = 0) /\ (n = 0)
|- !m n. (0 = m + n) = (m = 0) /\ (n = 0)
|- !x y. (x * y = 1) = (x = 1) /\ (y = 1)
|- !x y. (1 = x * y) = (x = 1) /\ (y = 1)
|- !m. m * 0 = 0
|- !m. 0 * m = 0
|- !x y. (x * y = SUC 0) = (x = SUC 0) /\ (y = SUC 0)
|- !x y. (SUC 0 = x * y) = (x = SUC 0) /\ (y = SUC 0)
|- !m. m * 1 = m
|- !m. 1 * m = m
|- !x.((SUC x = 1) = (x = 0)) /\ ((1 = SUC x) = (x = 0))
|- !x.((SUC x = 2) = (x = 1)) /\ ((2 = SUC x) = (x = 1))
|- !m n. (m + n = m) = (n = 0)
|- !m n. (n + m = m) = (n = 0)
|- !c. c - c = 0
```

```
|- !m. SUC m - 1 = m
|- !m. (0 - m = 0) /\ (m - 0 = m)
|- !a c. a + c - c = a
|- !m n. (m - n = 0) = m <= n
|- !m n. (0 = m - n) = m <= n
|- !n m. n - m <= n
|- !n m. SUC n - SUC m = n - m
|- !m n p. m - n > p = m > n + p
|- !m n p. m - n < p = m < n + p /\ 0 < p
|- !m n p. m - n >= p = m >= n + p \/ 0 >= p
|- !m n p. m - n <= p = m <= n + p
|- !n. n <= 0 = (n = 0)
|- !m n p. m + p < n + p = m < n
|- !m n p. p + m < p + n = m < n
|- !m n p. m + n <= m + p = n <= p
|- !m n p. n + m <= p + m = n <= p
|- !m n p. (m + p = n + p) = (m = n)
|- !m n p. (p + m = p + n) = (m = n)
|- !x y w. x + y < w + x = y < w
|- !x y w. y + x < x + w = y < w
|- !m n. (SUC m = SUC n) = (m = n)
|- !m n. SUC m < SUC n = m < n
|- !n m. SUC n <= SUC m = n <= m
|- !m i n. SUC n * m < SUC n * i = m < i
|- !p m n. (n * SUC p = m * SUC p) = (n = m)
|- !m i n. (SUC n * m = SUC n * i) = (m = i)
|- !n m. ~(SUC n <= m) = m <= n
|- !p q n m. (n * SUC q ** p = m * SUC q ** p) = (n = m)
|- !m n. ~(SUC n ** m = 0)
|- !n m. ~(SUC (n + n) = m + m)
|- !m n. ~(SUC (m + n) <= m)
|- !n. ~(SUC n <= 0)
|- !n. ~(n < 0)
|- !n. (MIN n 0 = 0) /\ (MIN 0 n = 0)
|- !n. (MAX n 0 = n) /\ (MAX 0 n = n)
|- !n. MIN n n = n
|- !n. MAX n n = n
|- !n m. MIN m n <= m /\ MIN m n <= n
|- !n m. m <= MAX m n /\ n <= MAX m n
|- !n m. (MIN m n < m = ~(m = n) /\ (MIN m n = n)) /\
```

```
           (MIN m n < n = ~(m = n) /\ (MIN m n = m)) /\
           (m < MIN m n = F) /\ (n < MIN m n = F)
|- !n m. (m < MAX m n = ~(m = n) /\ (MAX m n = n)) /\
           (n < MAX m n = ~(m = n) /\ (MAX m n = m)) /\
           (MAX m n < m = F) /\ (MAX m n < n = F)
|- !m n. (MIN m n = MAX m n) = (m = n)
|- !m n. MIN m n < MAX m n = ~(m = n)
```

The decision procedure proves valid purely univeral formulas constructed using variables and the operators `SUC`,`PRE`,`+`,`-`,`<`,`>`,`<=`,`>=`. Multiplication by constants is accomodated by translation to repeated addition. An attempt is made to generalize sub-formulas of type `num` not fitting into this syntax.

Comments

The philosophy behind this simpset is fairly conservative. For example, some potential rewrite rules, e.g., the recursive clauses for addition and multiplication, are not included, since it was felt that their incorporation too often resulted in formulas becoming more complex rather than simpler. Also, transitivity theorems are avoided because they tend to make simplification diverge.

See also

`BasicProvers.RW_TAC`, `BasicProvers.SRW_TAC`, `simpLib.SIMP_TAC`, `simpLib.SIMP_CONV`, `simpLib.SIMP_RULE`, `BasicProvers.bool_ss`, `bossLib.std_ss`, `bossLib.list_ss`

asm (bossLib)

```
bossLib.asm : string -> thm_tactic -> tactic
```

Passes the named assumption to the continuation.

Given a name `n`, applies the continuation to the assumption `n :- t`.

Failure

Fails if there is no assumption with name n, i.e., no assumption n :- t, or if the continuation fails for the given assumption.

See also

bossLib.asm_x, bossLib.mk_asm

ASM_CASES_TAC (Tactic)

```
Tactic.ASM_CASES_TAC : term -> tactic
```

Given a term, produces a case split based on whether or not that term is true.

Given a term u, ASM_CASES_TAC applied to a goal produces two subgoals, one with u as an assumption and one with ~u:

```
              A ?-  t
  ================================  ASM_CASES_TAC u
   A u {u} ?- t   A u {~u} ?- t
```

ASM_CASES_TAC u is implemented by DISJ_CASES_TAC(SPEC u EXCLUDED_MIDDLE), where EXCLUDED_MIDDLE is the axiom |- !u. u \/ ~u.

Failure

By virtue of the implementation (see above), the decomposition fails if EXCLUDED_MIDDLE cannot be instantiated to u, e.g. if u does not have boolean type.

Example

The tactic ASM_CASES_TAC u can be used to produce a case analysis on u:

```
- let val u = Term `u:bool`
      val g = Term `(P:bool -> bool) u`
```

```
    in
    ASM_CASES_TAC u ([],g)
    end;

    ([([`u`],  `P u`),
      ([`~u`], `P u`)], fn) : tactic_result
```

Performing a case analysis according to whether a given term is true or false.

See also

`Tactic.BOOL_CASES_TAC`, `Tactic.COND_CASES_TAC`, `Tactic.DISJ_CASES_TAC`, `Thm.SPEC`, `Tactic.STRUCT_CASES_TAC`, `BasicProvers.Cases`, `bossLib.Cases_on`

ASM_MESON_TAC (mesonLib)

```
mesonLib.ASM_MESON_TAC : thm list -> tactic
```

Performs first order proof search to prove the goal, using the assumptions and the theorems given.

`ASM_MESON_TAC` is identical in behaviour to `MESON_TAC` except that it uses the assumptions of a goal as well as the provided theorems.

Failure

`ASM_MESON_TAC` fails if it can not find a proof of the goal with depth less than or equal to the `mesonLib.max_depth` value.

See also

`mesonLib.GEN_MESON_TAC`, `mesonLib.MESON_TAC`

ASM_QI_TAC (bossLib)

```
bossLib.ASM_QI_TAC : tactic
```

Try to instantiate quantifiers with some default heuristics using also the assumptions..

ASM_QI_TAC is short for ASM_QUANT_INSTANTIATE_TAC [std_qp].

See also

bossLib.QI_TAC, quantHeuristicsLib.ASM_QUANT_INSTANTIATE_TAC, quantHeuristicsLib.QUANT_INSTANTIATE_TAC

ASM_QUANT_INSTANTIATE_TAC(quantHeuristicsLib)

quantHeuristicsLib.ASM_QUANT_INSTANTIATE_TAC : quant_param list -> tactic

---

A tactic to instantiate quantifiers in a term using a given list of quantifier heuristic parameters.

This tactic is based on quantHeuristicsLib.QUANT_INSTANTIATE_CONV. It tries to instantiate quantifiers. Free variables of the goal are seen as universally quantified by this tactic. Therefore, it tries to instantiate these free variables.

In contrast to quantHeuristicsLib.QUANT_INSTANTIATE_TAC this tactic takes the assumptions of the goal into account.

See also

quantHeuristicsLib.QUANT_INSTANTIATE_CONV, quantHeuristicsLib.QUANT_INSTANTIATE_TAC

ASM_REWRITE_RULE (Rewrite)

Rewrite.ASM_REWRITE_RULE : thm list -> thm -> thm

Rewrites a theorem including built-in rewrites and the theorem's assumptions.

ASM_REWRITE_RULE rewrites using the tautologies in implicit_rewrites, the given list of theorems, and the set of hypotheses of the theorem. All hypotheses are used. No ordering is specified among applicable rewrites. Matching subterms are searched for recursively, starting with the entire term of the conclusion and stopping when no rewritable expressions remain. For more details about the rewriting process, see GEN_REWRITE_RULE. To avoid using the set of basic tautologies, see PURE_ASM_REWRITE_RULE.

Failure

ASM_REWRITE_RULE does not fail, but may result in divergence. To prevent divergence where it would occur, ONCE_ASM_REWRITE_RULE can be used.

See also

Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_ASM_REWRITE_RULE, Rewrite.PURE_ASM_REWRITE_RULE, Rewrite.PURE_ONCE_ASM_REWRITE_RULE, Rewrite.REWRITE_RULE

ASM_REWRITE_TAC (Rewrite)

```
Rewrite.ASM_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using built-in rewrites and the goal's assumptions.

ASM_REWRITE_TAC generates rewrites with the tautologies in implicit_rewrites, the set of assumptions, and a list of theorems supplied by the user. These are applied top-down and recursively on the goal, until no more matches are found. The order in which the set of rewrite equations is applied is an implementation matter and the user should not depend on any ordering. Rewriting strategies are described in more detail under GEN_REWRITE_TAC. For omitting the common tautologies, see the tactic PURE_ASM_REWRITE_TAC. To rewrite with only a subset of the assumptions use FILTER_ASM_REWRITE_TAC.

Failure

ASM_REWRITE_TAC does not fail, but it can diverge in certain situations. For rewriting to a limited depth, see ONCE_ASM_REWRITE_TAC. The resulting tactic may not be valid if the

applicable replacement introduces new assumptions into the theorem eventually proved.

Example

The use of assumptions in rewriting, specially when they are not in an obvious equational form, is illustrated below:

```
- let val asm = [Term `P x`]
      val goal = Term `P x = Q x`
  in
  ASM_REWRITE_TAC[] (asm, goal)
  end;

val it = ([([`P x`], `Q x`)], fn) : tactic_result

- let val asm = [Term `~P x`]
      val goal = Term `P x = Q x`
  in
  ASM_REWRITE_TAC[] (asm, goal)
  end;

val it = ([([`~P x`], `~Q x`)], fn) : tactic_result
```

See also

Rewrite.FILTER_ASM_REWRITE_TAC, Rewrite.FILTER_ONCE_ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_REWRITE_TAC, Rewrite.PURE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_ASM_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST_TAC, BasicProvers.VAR_EQ_TAC

ASM_SET_TAC (bossLib)

```
bossLib.ASM_SET_TAC : thm list -> tactic
```

Tactic to automate some routine set theory by reduction to FOL, using the assumptions and the theorems given.

ASM_SET_TAC is identical in behaviour to SET_TAC except that it uses the assumptions of a goal as well as the provided theorems.

Failure

Fails if the underlying resolution machinery (METIS_TAC) cannot prove the goal, or the supplied theorems (and the assumptions) are not enough for the FOL reduction.

See also

bossLib.SET_TAC, bossLib.SET_RULE

ASM_SIMP_RULE (simpLib)

```
simpLib.ASM_SIMP_RULE : simpset -> thm list -> thm -> thm
```

Simplifies a theorem, using the theorem's assumptions as rewrites in addition to the provided rewrite theorems and simpset.

Failure

Never fails, but may diverge.

Example

```
> simpLib.ASM_SIMP_RULE bool_ss [] (ASSUME (Term `x = 3`))
val it =  [.] ⊢ T: thm
```

The assumptions can be used to simplify the conclusion of the theorem. For example, if the conclusion of a theorem is an implication, the antecedent together with the hypotheses may help simplify the conclusion.

See also

simpLib.SIMP_CONV, simpLib.SIMP_RULE

ASM_SIMP_TAC (bossLib)

```
bossLib.ASM_SIMP_TAC : simpset -> thm list -> tactic
```

Also exported as `simpLib.ASM_SIMP_TAC`.

Simplifies a goal using the simpset, the provided theorems, and the goal's assumptions.

`ASM_SIMP_TAC` does a simplification of the goal, adding both the assumptions and the provided theorem to the given simpset as rewrites. This simpset is then applied to the goal in the manner explained in the entry for `SIMP_CONV`.

`ASM_SIMP_TAC` is to `SIMP_TAC`, as `ASM_REWRITE_TAC` is to `REWRITE_TAC`.

Failure

`ASM_SIMP_TAC` never fails, though it may diverge.

Example

The simple goal `x < y ?- x + y < y + y` can be proved by using `bossLib.arith_ss` and the assumption by

```
ASM_SIMP_TAC bossLib.arith_ss []
```

See also

`bossLib.++`, `bossLib.bool_ss`, `bossLib.FULL_SIMP_TAC`, `simpLib.mk_simpset`, `bossLib.SIMP_CONV`, `bossLib.SIMP_TAC`, `Rewrite.ASM_REWRITE_TAC`, `BasicProvers.VAR_EQ_TAC`

ASM_SIMP_TAC (simpLib)

```
simpLib.ASM_SIMP_TAC : simpset -> thm list -> tactic
```

Re-exported from `bossLib.ASM_SIMP_TAC`. See that entry for full documentation.

asm_x (bossLib)

```
bossLib.asm_x : string -> thm_tactic -> tactic
```

Passes the named assumption to the continuation.

Given a name `n`, applies the continuation to the assumption `n :- t` and removes `n :- t` from the assumption list.

Failure

Fails if there is no assumption with name `n`, i.e., no assumption `n :- t`, or if the continuation fails for the given assumption.

See also

`bossLib.asm`, `bossLib.mk_asm`

assert (Lib)

```
Lib.assert : ('a -> bool) -> 'a -> 'a
```

Checks that a value satisfies a predicate.

`assert p x` returns `x` if the application `p x` yields `true`. Otherwise, `assert p x` fails.

Failure

`assert p x` fails with exception `HOL_ERR` if the predicate `p` yields `false` when applied to the value `x`. If the application `p x` raises an exception `e`, then `assert p x` raises `e`.

Example

```
> null [];
val it = true: bool

> assert null ([]:int list);
val it = []: int list

> null [1];
val it = false: bool

> assert null [1];
Exception- HOL_ERR (at Lib.assert: predicate not true) raised
```

See also

`Lib.can`, `Lib.assert_exn`

assert_exn (Lib)

```
Lib.assert_exn : ('a -> bool) -> 'a -> exn -> 'a
```

Checks that a value satisfies a predicate.

`assert_exn p x e` returns `x` if the application `p x` evaluates to `true`. Otherwise, `assert_exn p x e` raises `e`

Failure

`assert_exn p x e` fails with exception `e` if the predicate `p` yields `false` when applied to the value `x`. If the application `p x` raises an exception `ex`, then `assert_exn p x e` raises `ex`.

Example

```
> null [];
```

```
val it = true: bool

> assert_exn null ([]:int list) (Fail "non-empty list");
val it = []: int list

> null [1];
val it = false: bool

> assert_exn null [1] (Fail "non-empty list");;
Exception- Fail "non-empty list" raised
```

See also

`Lib.can`, `Lib.assert`

assoc (Lib)

```
Lib.assoc : ''a -> (''a * 'b) list -> 'b
```

Searches a list of pairs for a pair whose first component equals a specified value, then returns the second component of the pair.

`assoc x [(x1,y1),...,(xn,yn)]` locates the first `(xi,yi)` in a left-to-right scan of the list such that `xi` equals `x`. Then `yi` is returned. The lookup is done on an eqtype, i.e., the SML implementation must be able to decide equality for the type of `x`.

Failure

Fails if no matching pair is found. This will always be the case if the list is empty.

Example

```
> assoc 2 [(1,4),(3,2),(2,5),(2,6)];
val it = 5: int
```

See also

Lib.assoc1, Lib.assoc2, Lib.rev_assoc, Lib.mem, Lib.tryfind, Lib.exists, Lib.all

assoc (hol88Lib)

```
hol88Lib.assoc : ''a -> (''a * 'b) list -> ''a * 'b
```

Searches a list of pairs for a pair whose first component equals a specified value.

`assoc x [(x1,y1),...,(xn,yn)]` returns the first `(xi,yi)` in the list such that `xi` equals `x`. The lookup is done on an eqtype, i.e., the SML implementation must be able to decide equality for the type of `x`.

Failure

Fails if no matching pair is found. This will always be the case if the list is empty.

Example

```
  - assoc 2 [(1,4),(3,2),(2,5),(2,6)];
(2, 5) : (int * int)
```

Comments

Superseded by Lib.assoc and Lib.assoc1.

See also

hol88Lib.rev_assoc, Lib.assoc, Lib.assoc1

assoc1 (Lib)

```
Lib.assoc1 : ''a -> (''a * 'b) list -> (''a * 'b)option
```

Searches a list of pairs for a pair whose first component equals a specified value.

`assoc1 x [(x1,y1),...,(xn,yn)]` returns `SOME (xi,yi)` for the first pair `(xi,yi)` in the list such that `xi` equals `x`. Otherwise, `NONE` is returned. The lookup is done on an eqtype, i.e., the SML implementation must be able to decide equality for the type of `x`.

Failure

Never fails.

Example

```
> assoc1 2 [(1,4),(3,2),(2,5),(2,6)];
val it = SOME (2, 5): (int * int) option
```

See also

`Lib.assoc`, `Lib.assoc2`, `Lib.rev_assoc`, `Lib.mem`, `Lib.tryfind`, `Lib.exists`, `Lib.all`

assoc2 (Lib)

```
Lib.assoc2 : ''a -> ('b * ''a) list -> ('b * ''a)option
```

Searches a list of pairs for a pair whose second component equals a specified value.

An invocation `assoc2 y [(x1,y1),...,(xn,yn)]` returns `SOME (xi,yi)` for the first `(xi,yi)` in the list such that `yi` equals `y`. Otherwise, `NONE` is returned. The lookup is done on an eqtype, i.e., the SML implementation must be able to decide equality for the type of `y`.

Failure

Never fails.

Example

```
> assoc2 2 [(1,4),(3,2),(2,5),(2,6)];
val it = SOME (3, 2): (int * int) option
```

See also

Lib.assoc, Lib.assoc1, Lib.rev_assoc, Lib.mem, Lib.tryfind, Lib.exists, Lib.all

associate_restriction (Parse)

```
Parse.associate_restriction : (string * string) -> unit
```

Associates a restriction semantics with a binder.

If B is a binder and RES_B a constant then

```
associate_restriction("B", "RES_B")
```

will cause the parser and pretty-printer to support:

```
            ---- parse ---->
Bv::P. B                      RES_B  P (\v. B)
          <---- print ----
```

Anything can be written between the binder and "::" that could be written between the binder and "." in the old notation. See the examples below.

The following associations are predefined:

```
\v::P. B    <---->   RES_ABSTRACT P (\v. B)
!v::P. B    <---->   RES_FORALL   P (\v. B)
?v::P. B    <---->   RES_EXISTS   P (\v. B)
@v::P. B    <---->   RES_SELECT   P (\v. B)
```

Where the constants RES_FORALL, RES_EXISTS and RES_SELECT are defined in the theory bool, such that :

```
|- RES_FORALL P B   =  !x:'a. P x ==> B x

|- RES_EXISTS P B   =  ?x:'a. P x /\ B x

|- RES_SELECT P B   =  @x:'a. P x /\ B x
```

The constant RES_ABSTRACT has the following characterisation

```
|- (!p m x. x IN p ==> (RES_ABSTRACT p m x = m x)) /\
   !p m1 m2.
     (!x. x IN p ==> (m1 x = m2 x)) ==>
     (RES_ABSTRACT p m1 = RES_ABSTRACT p m2)
```

Failure

Never fails.

Example

```
> new_binder_definition("DURING", ``DURING(p:num#num->bool) = $!p``);
val it = ⊢ ∀p. $DURING p ⇔ $! p: thm

> ``DURING x::(m,n). p x``;
Exception- HOL_ERR
  (at Absyn.Absyn: on line 1, characters 2-21:
       parse_term: No restricted quantifier associated with DURING) raised

> new_definition("RES_DURING",
                 ``RES_DURING(m,n)p = !x. m<=x /\ x<=n ==> p x``);
val it = ⊢ ∀m n p. RES_DURING (m,n) p ⇔ ∀x. m ≤ x ∧ x ≤ n ⇒ p x: thm

> associate_restriction("DURING","RES_DURING");
val it = (): unit

> ``DURING x::(m,n). p x``;
val it = “DURING x::(m,n). p x”: term

> dest_comb it;
val it = (“RES_DURING (m,n)”, “λx. p x”): term * term
```

ASSUM_LIST (Tactical)

```
Tactical.ASSUM_LIST : (thm list -> tactic) -> tactic
```

Applies a tactic generated from the goal's assumption list.

When applied to a function of type `thm list -> tactic` and a goal, `ASSUM_LIST` constructs a tactic by applying `f` to a list of `ASSUME`d assumptions of the goal, then applies that tactic to the goal.

```
ASSUM_LIST f ({A1,...,An} ?- t)
      = f [A1 |- A1, ... , An |- An] ({A1,...,An} ?- t)
```

Failure

Fails if the function fails when applied to the list of `ASSUME`d assumptions, or if the resulting tactic fails when applied to the goal.

Comments

There is nothing magical about `ASSUM_LIST`: the same effect can usually be achieved just as conveniently by using `ASSUME a` wherever the assumption `a` is needed. If `ASSUM_LIST` is used, it is extremely unwise to use a function which selects elements from its argument list by number, since the ordering of assumptions should not be relied on.

Example

The tactic:

```
ASSUM_LIST SUBST_TAC
```

makes a single parallel substitution using all the assumptions, which can be useful if the rewriting tactics are too blunt for the required task.

Making more careful use of the assumption list than simply rewriting or using resolution.

See also

`Rewrite.ASM_REWRITE_TAC`, `Tactical.EVERY_ASSUM`, `Tactic.IMP_RES_TAC`, `Tactical.POP_ASSUM`, `Tactical.POP_ASSUM_LIST`, `Rewrite.REWRITE_TAC`

ASSUME (Thm)

```
Thm.ASSUME : term -> thm
```

Introduces an assumption.

When applied to a term `t`, which must have type `bool`, the inference rule `ASSUME` returns the theorem `t |- t`.

```
--------  ASSUME t
 t |- t
```

Failure

Fails unless the term `t` has type `bool`.

See also

`Drule.ADD_ASSUM`, `Thm.REFL`

ASSUME_CONJS (Drule)

```
Drule.ASSUME_CONJS : term -> thm
```

Constructs a theorem proving a conjunction from its individual conjuncts

Takes a term which should be a conjunction, and returns a theorem whose hypotheses are the individual conjuncts, and whose conclusion is the argument term, the conjunction.

Failure

Never fails.

Example

`ASSUME_CONJS (``t1 /\ t2 /\ ... /\ tn``)` returns `[t1, t2, ..., tn] |- t1 /\ t2 /\ ... /\ tn`

To split up conjuncts in selected hypotheses `hyps` of a theorem `th`, use `Lib.itlist (PROVE_HYP o ASSUME_CONJS) hyps th`

See also

`Drule.CONJUNCTS`, `Thm.CONJ`, `Drule.CONJUNCTS_AC`, `Drule.UNDISCH_SPLIT`

ASSUME_TAC (Tactic)

```
Tactic.ASSUME_TAC : thm_tactic
```

Adds an assumption to a goal.

Given a theorem `th` of the form `A' |- u`, and a goal, `ASSUME_TAC th` adds `u` to the assumptions of the goal.

```
      A ?- t
==============  ASSUME_TAC (A' |- u)
 A u {u} ?- t
```

Note that unless `A'` is a subset of `A`, this tactic is invalid.

Failure

Never fails.

Example

Given a goal g of the form `{x = y, y = z} ?- P`, where x, y and z have type `:'a`, the theorem `x = y, y = z |- x = z` can, first, be inferred by forward proof

```
let val eq1 = Term `(x:'a) = y`
    val eq2 = Term `(y:'a) = z`
in
TRANS (ASSUME eq1) (ASSUME eq2)
end;
```

and then added to the assumptions. This process requires the explicit text of the assumptions, as well as invocation of the rule ASSUME:

```
let val eq1 = Term `(x:'a) = y`
    val eq2 = Term `(y:'a) = z`
    val goal = ([eq1,eq2],Parse.Term `P:bool`)
in
ASSUME_TAC (TRANS (ASSUME eq1) (ASSUME eq2)) goal
end;

val it = ([([`x = z`, `x = y`, `y = z`], `P`)], fn) : tactic_result
```

This is the naive way of manipulating assumptions; there are more advanced proof styles (more elegant and less transparent) that achieve the same effect, but this is a perfectly correct technique in itself.

Alternatively, the axiom EQ_TRANS could be added to the assumptions of g:

```
let val eq1 = Term `(x:'a) = y`
    val eq2 = Term `(y:'a) = z`
    val goal = ([eq1,eq2], Term `P:bool`)
in
ASSUME_TAC EQ_TRANS goal
end;

val it =
  ([([`!x y z. (x = y) /\ (y = z) ==> (x = z)`,
      `x = y`,`y = z`],`P`)],fn) : tactic_result
```

A subsequent resolution (see RES_TAC) would then be able to add the assumption x = z to the subgoal shown above. (Aside from purposes of example, it would be more usual to use IMP_RES_TAC than ASSUME_TAC followed by RES_TAC in this context.)

ASSUME_TAC is the naive way of manipulating assumptions (i.e. without recourse to advanced tacticals); and it is useful for enriching the assumption list with lemmas as a prelude to resolution (RES_TAC, IMP_RES_TAC), rewriting with assumptions (ASM_REWRITE_TAC and so on), and other operations involving assumptions.

See also

Tactic.ACCEPT_TAC, Tactic.IMP_RES_TAC, Tactic.RES_TAC, Tactic.STRIP_ASSUME_TAC

augment_srw_ss (BasicProvers)

```
BasicProvers.augment_srw_ss : ssfrag list -> unit
```

Re-exported from bossLib.augment_srw_ss. See that entry for full documentation.

augment_srw_ss (bossLib)

```
bossLib.augment_srw_ss : simpLib.ssfrag list -> unit
```

Also exported as BasicProvers.augment_srw_ss.

Augments the "stateful rewriter" with a list of simpset fragments.

A call to augment_srw_ss sslist causes each element of sslist to be merged into the simpset value that the system maintains "behind" srw_ss().

Failure

Never fails.

Comments

The change to the `srw_ss()` `simpset` brought about with `augment_srw_ss` is not exported with a theory, so it is not "permanent". But see `export_rewrites` for a simple way to achieve a sort of permanence.

See also

`BasicProvers.export_rewrites`, `bossLib.srw_ss`, `bossLib.SRW_TAC`

axioms (DB)

```
DB.axioms : string -> (string * thm) list
```

All the axioms stored in the named theory.

An invocation `axioms thy`, where `thy` is the name of a currently loaded theory segment, will return a list of the axioms stored in that theory. Each theorem is paired with its name in the result. The string `"-"` may be used to denote the current theory segment.

Failure

Never fails. If `thy` is not the name of a currently loaded theory segment, the empty list is returned.

Example

```
> axioms "bool";
val it =
   [("SELECT_AX", ⊢ ∀P x. P x ⇒ P ($@ P)),
    ("INFINITY_AX", ⊢ ∃f. ONE_ONE f ∧ ¬ONTO f),
    ("ETA_AX", ⊢ ∀t. (λx. t x) = t),
    ("BOOL_CASES_AX", ⊢ ∀t. (t ⇔ T) ∨ (t ⇔ F))]: (string * thm) list
```

See also

`DB.thy`, `DB.fetch`, `DB.thms`, `DB.theorems`, `DB.definitions`, `DB.listDB`

b (proofManagerLib)

```
proofManagerLib.b : unit -> proof
```

Restores the proof state undoing the effects of a previous expansion.

The function `b` is part of the subgoal package. It is an abbreviation for the function `backup`. For a description of the subgoal package, see `set_goal`.

Failure

As for `backup`.

Back tracking in a goal-directed proof to undo errors or try different tactics.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.rd`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.sav` `proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

backup (proofManagerLib)

```
proofManagerLib.backup : unit -> proof
```

Restores the proof state, undoing the effects of a previous expansion.

The function `backup` is part of the subgoal package. It may be abbreviated by the function `b`. It allows backing up from the last state change (caused by calls to `expand`, `rotate` and similar functions). The package maintains a backup list of previous proof states. A call to `backup` restores the state to the previous state (which was on top of the backup list). The current state and the state on top of the backup list are discarded. The maximum number of proof states saved on the backup list can be set using `set_backup`. It defaults to 15. Adding new proof states after the maximum is reached causes the earliest proof state

on the list to be discarded. The user may backup repeatedly until the list is exhausted. The state restored includes all unproven subgoals or, if a goal had been proved in the previous state, the corresponding theorem. For a description of the subgoal package, see `set_goal`.

Failure

The function `backup` will fail if the backup list is empty.

Example

```
> g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;
val it =
   Proof manager status: 5 proofs.
   5. Incomplete goalstack:
        Initial goal:
         0.  p ⇒ q
        ------------------------------------
            r

        Current goal:
         0.  p ⇒ q
        ------------------------------------
            p

   4. Incomplete goalstack:
        Initial goal:
        p ∧ q ⇒ r ∧ s

        Current goal:
         0.  p
         1.  q
        ------------------------------------
            p'

   3. Incomplete goalstack:
        Initial goal:
        ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z
```

```
        Current goal:
         0.  ∀x z y. x ≤ z ⇒ ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z
         1.  ¬(x ≤ z)
        ------------------------------------
             ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z

   2. Incomplete goalstack:
        Initial goal:
        MAX x y ≤ z ⇔ x ≤ z ∧ y ≤ z

        Current goal:
         0.  ∀x y z. x ≤ y ⇒ (MAX x y ≤ z ⇔ x ≤ z ∧ y ≤ z)
         1.  ¬(x ≤ y)
        ------------------------------------
             MAX x y ≤ z ⇔ x ≤ z ∧ y ≤ z

   1. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

> e CONJ_TAC;
OK..
2 subgoals:
val it =

   TL [1; 2; 3] = [2; 3]

   HD [1; 2; 3] = 1

> backup();
val it =
   Initial goal:

   HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]: proof

> e (REWRITE_TAC[listTheory.HD, listTheory.TL]);
OK..
val it =
   Initial goal proved.
   ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]: proof
```

Back tracking in a goal-directed proof to undo errors or try different tactics.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.b`, `proofManagerLib.backu`
`proofManagerLib.rd`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`,
`proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`,
`proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

BBLAST_CONV (blastLib)

`blastLib.BBLAST_CONV : conv`

Bit-blasting conversion for words.

This conversion expands bit-vector terms into Boolean propositions. It goes beyond the functionality of `wordsLib.WORD_BIT_EQ_CONV` by handling addition, subtraction and orderings. Consequently, this conversion can automatically handle small, but tricky, bit-vector goals that `wordsLib.WORD_DECIDE` cannot handle. Obviously bit-blasting is a brute force approach, so this conversion should be used with care. It will only work well for smallish word sizes and when there is only and handful of additions around. It is also "eager" – additions are expanded out even when not strictly necessary. For example, in

```
(a + b) <+ c /\ c <+ d ==> (a + b) <+ d:word32
```

the sum `a + b` is expanded. Users may be able to achieve speed-ups by first introducing abbreviations and then proving general forms, e.g.

```
x <+ c /\ c <+ d ==> x <+ d:word32
```

The conversion handles most operators, however, the following are not covered / interpreted:

- Type variables for word lengths, i.e. terms of type `:'a word`.
- General multiplication, i.e. `w1 * w2`. Multiplication by a literal is okay, although this may introduce many additions.

- Bit-field selections with non-literal bounds, e.g. `(expr1 -- expr2) w`.
- Shifting by non-literal amounts, e.g. `w << expr`.
- `n2w expr` and `w2n w`. Also `w2s`, `s2w`, `w2l` and `l2w`.
- `word_div`, `word_sdiv`, `word_mod` and `word_log2`.

Example

Word orderings are handled:

```
> blastLib.BBLAST_CONV
    “!a b. ~word_msb a /\ ~word_msb b ==> (a <+ b <=> a < b:word32)”;
val it = ⊢ (∀a b. ¬word_msb a ∧ ¬word_msb b ⇒ (a <+ b ⇔ a < b)) ⇔ T: thm
```

In some cases the result will be a proposition over bit values:

```
> blastLib.BBLAST_CONV “!a. (a + 1w:word8) ' 1”;
val it = ⊢ (∀a. (a + 1w) ' 1) ⇔ ∀a. a ' 1 ⇔ ¬a ' 0: thm
```

This conversion is especially useful where “logical” and “arithmetic” bit-vector operations are combined:

```
> blastLib.BBLAST_CONV
    “!a. ((((((a:word8) * 16w) + 0x10w)) && 0xF0w) >>> 4) = (3 -- 0) (a + 1w)”;
val it = ⊢ (∀a. (a * 16w + 16w && 240w) ⋙ 4 = (3 -- 0) (a + 1w)) ⇔ T: thm
```

See also

`wordsLib.WORD_ss`, `wordsLib.WORD_ARITH_CONV`, `wordsLib.WORD_LOGIC_CONV`, `wordsLib.WORD_MUL_LSL_CONV`, `wordsLib.WORD_BIT_EQ_CONV`, `wordsLib.WORD_EVAL_CONV`, `wordsLib.WORD_CONV`

BEQ_CONV (reduceLib)

`reduceLib.BEQ_CONV : conv`

Simplifies certain expressions involving boolean equality.

If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `BEQ_CONV tm` returns the corresponding theorem. Note that in the last case the left-hand and right-hand sides need only be alpha-equivalent rather than strictly identical.

```
BEQ_CONV "T = t" = |- T = t = t
BEQ_CONV "t = T" = |- t = T = t
BEQ_CONV "F = t" = |- F = t = ~t
BEQ_CONV "t = F" = |- t = F = ~t
BEQ_CONV "t = t" = |- t = t = T
```

Failure

`BEQ_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
#BEQ_CONV "T = T";;
|- (T = T) = T

#BEQ_CONV "F = T";;
|- (F = T) = F

#BEQ_CONV "(!x:*#**. x = (FST x,SND x)) = (!y:*#**. y = (FST y,SND y))";;
|- ((!x. x = FST x,SND x) = (!y. y = FST y,SND y)) = T
```

Beta (Thm)

```
Thm.Beta : thm -> thm
```

Perform one step of beta-reduction on the right hand side of an equational theorem.

`Beta` performs a single beta-reduction step on the right-hand side of an equational theorem.

```
   A |- t = ((\x.M) N)
  --------------------- Beta
   A |- t = M [N/x]
```

Failure

If the theorem is not an equation, or if the right hand side of the equation is not a beta-redex.

Example

```
> val th = REFL “(K:'a ->'b->'a) x”;
val th = ⊢ K x = K x: thm

> SUBS_OCCS [([2],combinTheory.K_DEF)] th;
val it = ⊢ K x = (λx y. x) x: thm

> Beta it;
val it = ⊢ K x = (λy. x): thm
```

Comments

`Beta` is equivalent to `RIGHT_BETA` but faster.

See also

`Drule.RIGHT_BETA`, `Drule.ETA_CONV`

beta (Type)

```
Type.beta : hol_type
```

Common type variable.

The ML variable `Type.beta` is bound to the type variable `'b`.

See also

`Type.alpha`, `Type.gamma`, `Type.delta`, `Type.bool`

beta_conv (Term)

```
Term.beta_conv : term -> term
```

Performs one step of beta-reduction.

Beta-reduction is one of the primitive operations in the lambda calculus. A step of beta-reduction may be performed by `beta_conv M`, where `M` is the application of a lambda abstraction to an argument, i.e., has the form `((\v.N) P)`. The beta-reduction occurs by systematically replacing every free occurrence of `v` in `N` by `P`.

Care is taken so that no free variable of `P` becomes captured in this process.

Failure

If `M` is not the application of an abstraction to an argument.

Example

```
> beta_conv (mk_comb (Term `\(x:'a) (y:'b). x`, Term `(P:bool -> 'a) Q`));
val it = “λy. P Q”: term

> beta_conv (mk_comb (Term `\(x:'a) (y:'b) (y':'b). x`, Term `y:'a`));
val it = “λy' y'. y”: term
```

Comments

More complex strategies for coding up full beta-reduction can be coded up in ML. The `conversions` of Larry Paulson support this activity as inference steps.

For programming derived rules of inference.

See also

Thm.BETA_CONV, Drule.RIGHT_BETA, Drule.LIST_BETA_CONV, Drule.RIGHT_LIST_BETA, Conv.DEPTH_CONV, Conv.TOP_DEPTH_CONV, Conv.REDEPTH_CONV

BETA_CONV (Thm)

```
Thm.BETA_CONV : conv
```

Performs a single step of beta-conversion.

The conversion BETA_CONV maps a beta-redex "(\x.u)v" to the theorem

```
|- (\x.u)v = u[v/x]
```

where u[v/x] denotes the result of substituting v for all free occurrences of x in u, after renaming sufficient bound variables to avoid variable capture. This conversion is one of the primitive inference rules of the HOL system.

Failure

BETA_CONV tm fails if tm is not a beta-redex.

Example

```
> BETA_CONV (Term `(\x.x+1)y`);
val it = ⊢ (λx. x + 1) y = y + 1: thm

> BETA_CONV (Term `(\x y. x+y)y`);
val it = ⊢ (λx y. x + y) y = (λy'. y + y'): thm
```

See also

Conv.BETA_RULE, Tactic.BETA_TAC, Drule.LIST_BETA_CONV, PairedLambda.PAIRED_-BETA_CONV, Drule.RIGHT_BETA, Drule.RIGHT_LIST_BETA

BETA_RULE (Conv)

```
Conv.BETA_RULE : (thm -> thm)
```

Beta-reduces all the beta-redexes in the conclusion of a theorem.

When applied to a theorem `A |- t`, the inference rule `BETA_RULE` beta-reduces all beta-redexes, at any depth, in the conclusion `t`. Variables are renamed where necessary to avoid free variable capture.

```
 A |- ....((\x. s1) s2)....
---------------------------  BETA_RULE
   A |- ....(s1[s2/x])....
```

Failure

Never fails, but will have no effect if there are no beta-redexes.

Example

The following example is a simple reduction which illustrates variable renaming:

```
> Globals.show_assums := true;
val it = (): unit

> local val tm = “f = ((\x y. x + y) y)”
  in
   val x = ASSUME tm
  end;
val x =  [f = (λx y. x + y) y] ⊢ f = (λx y. x + y) y: thm

> BETA_RULE x;
val it =  [f = (λx y. x + y) y] ⊢ f = (λy'. y + y'): thm
```

See also

`Thm.BETA_CONV`, `Tactic.BETA_TAC`, `PairedLambda.PAIRED_BETA_CONV`, `Drule.RIGHT_-BETA`

BETA_TAC (Tactic)

```
Tactic.BETA_TAC : tactic
```

Beta-reduces all the beta-redexes in the conclusion of a goal.

When applied to a goal `A ?- t`, the tactic `BETA_TAC` produces a new goal which results from beta-reducing all beta-redexes, at any depth, in `t`. Variables are renamed where necessary to avoid free variable capture.

```
 A ?- ...((\x. s1) s2)...
=========================  BETA_TAC
  A ?- ...(s1[s2/x])...
```

Failure

Never fails, but will have no effect if there are no beta-redexes.

See also

`Thm.BETA_CONV`, `Tactic.BETA_TAC`, `PairedLambda.PAIRED_BETA_CONV`

BINDER_CONV (Conv)

```
Conv.BINDER_CONV : conv -> conv
```

Applies a conversion underneath a binder.

If `conv N` returns `A |- N = P`, then `BINDER_CONV conv (M (\v.N))` returns `A |- M (\v.N) = M (\v.P)` and `BINDER_CONV conv (\v.N)` returns `A |- (\v.N) = (\v.P)`

Failure

If `conv N` fails, or if `v` is free in `A`.

Example

```
> BINDER_CONV SYM_CONV (Term `\x. x + 0 = x`);
val it = ⊢ (λx. x + 0 = x) = (λx. x = x + 0): thm
```

Comments

For deeply nested quantifiers, `STRIP_BINDER_CONV` and `STRIP_QUANT_CONV` are more efficient than iterated application of `BINDER_CONV`, `BINDER_CONV`, or `ABS_CONV`.

See also

`Conv.QUANT_CONV`, `Conv.STRIP_QUANT_CONV`, `Conv.STRIP_BINDER_CONV`, `Conv.ABS_CONV`

BINOP_CONV (Conv)

```
Conv.BINOP_CONV : conv -> conv
```

Applies a conversion to both arguments of a binary operator.

If `c` is a conversion that when applied to `t1` returns the theorem `|- t1 = t1'` and when applied to `t2` returns the theorem `|- t2 = t2'`, then `BINOP_CONV c (Term`f t1 t2`)` will return the theorem

```
|- f t1 t2 = f t1' t2'
```

Failure

`BINOP_CONV c t` will fail if `t` is not of the general form `f t1 t2`, or if `c` fails when applied to either `t1` or `t2`, or if `c` fails to return theorems of the form `|- t1 = t1'` and `|- t2 = t2'` when applied to those arguments. (The latter case would imply that `c` wasn't a conversion at all.)

Example

```
> BINOP_CONV reduceLib.REDUCE_CONV (Term`3 * 4 + 6 * 7`);
val it = ⊢ 3 * 4 + 6 * 7 = 12 + 42: thm
```

See also

Conv.FORK_CONV, Conv.LAND_CONV, Conv.RAND_CONV, Conv.RATOR_CONV, numLib.REDUCE_CONV

BIT_ss (wordsLib)

```
wordsLib.BIT_ss : ssfrag
```

Simplification fragment for words.

The fragment BIT_ss rewrites the term "BIT i n" for ground n.

Example

```
> SIMP_CONV (std_ss++wordsLib.BIT_ss) [] "BIT i 33";
val it = ⊢ BIT i 33 ⇔ i ∈ {0; 5}: thm

> SIMP_CONV (std_ss++wordsLib.BIT_ss) [] "BIT 5 33";
val it = ⊢ BIT 5 33 ⇔ T: thm
```

See also

wordsLib.WORD_CONV, fcpLib.FCP_ss, wordsLib.SIZES_ss, wordsLib.WORD_ARITH_ss, wordsLib.WORD_LOGIC_ss, wordsLib.WORD_SHIFT_ss, wordsLib.WORD_ARITH_EQ_ss, wordsLib.WORD_BIT_EQ_ss, wordsLib.WORD_EXTRACT_ss, wordsLib.WORD_MUL_LSL_ss, wordsLib.WORD_ss

BITS_INTRO_CONV (wordsLib)

```
wordsLib.BITS_INTRO_CONV : conv
```

Tries to convert terms of the form `n MOD p`, `n MOD p DIV q` and `(n DIV q) MOD p` into a terms of the form `BITS h l n`. This will succeed when `p` and `q` are powers of two.

Example

```
> wordsLib.BITS_INTRO_CONV “(n DIV 16) MOD 4”;
val it = ⊢ n DIV 16 MOD 4 = BITS 5 4 n: thm

> wordsLib.BITS_INTRO_CONV “n MOD 16 DIV 4”;
val it = ⊢ n MOD 16 DIV 4 = BITS 3 2 n: thm

> wordsLib.BITS_INTRO_CONV “n MOD 16”;
val it = ⊢ n MOD 16 = BITS 3 0 n: thm

> wordsLib.BITS_INTRO_CONV “n MOD dimword(:'a)”;
val it = ⊢ n MOD dimword (:α) = BITS (dimindex (:α) − 1) 0 n: thm
```

See also

`wordsLib.BITS_INTRO_ss`

BITS_INTRO_ss (wordsLib)

```
wordsLib.BITS_INTRO_ss : ssfrag
```

Simplification fragment that applies the conversion `BITS_INTRO_CONV`.

See also

wordsLib.BITS_INTRO_CONV

body (Term)

```
Term.body : term -> term
```

Returns the body of an abstraction.

If M is a lambda abstraction, i.e, has the form \v. t, then body M returns t.

Failure

Fails unless M is an abstraction.

See also

Term.bvar, Term.dest_abs

BODY_CONJUNCTS (Drule)

```
Drule.BODY_CONJUNCTS : (thm -> thm list)
```

Splits up conjuncts recursively, stripping away universal quantifiers.

When applied to a theorem, BODY_CONJUNCTS recursively strips off universal quantifiers by specialization, and breaks conjunctions into a list of conjuncts.

```
 A |- !x1...xn. t1 /\ (!y1...ym. t2 /\ t3) /\ ...
-------------------------------------------------  BODY_CONJUNCTS
       [A |- t1, A |- t2, A |- t3, ...]
```

Failure

Never fails, but has no effect if there are no top-level universal quantifiers or conjuncts.

Example

The following illustrates how a typical term will be split:

```
- local val tm = Parser.term_parser
                     `!x:bool. A /\ (B \/ (C /\ D)) /\ ((!y:bool. E) /\ F)`
  in
  val x = ASSUME tm
  end;

  val x = . |- !x. A /\ (B \/ C /\ D) /\ (!y. E) /\ F : thm

- BODY_CONJUNCTS x;
val it = [. |- A, . |- B \/ C /\ D, . |- E, . |- F] : thm list
```

See also

Thm.CONJ, Thm.CONJUNCT1, Thm.CONJUNCT2, Drule.CONJUNCTS, Tactic.CONJ_TAC

bool (Type)

```
Type.bool : hol_type
```

Basic type constant.

The ML variable Type.bool is bound to the type constant bool.

See also

alpha, Type.beta, Type.gamma, Type.delta

bool_case (boolSyntax)

```
boolSyntax.bool_case : term
```

Constant denoting case expressions for `bool`.

The ML variable `boolSyntax.bool_case` is bound to the term `bool$bool_case`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.let_tm`, `boolSyntax.arb`

BOOL_CASES_TAC (Tactic)

```
Tactic.BOOL_CASES_TAC : (term -> tactic)
```

Performs boolean case analysis on a (free) term in the goal.

When applied to a term `x` (which must be of type `bool` but need not be simply a variable), and a goal `A ?- t`, the tactic `BOOL_CASES_TAC` generates the two subgoals corresponding to `A ?- t` but with any free instances of `x` replaced by `F` and `T` respectively.

```
            A ?- t
============================  BOOL_CASES_TAC "x"
 A ?- t[F/x]    A ?- t[T/x]
```

The term given does not have to be free in the goal, but if it isn't, `BOOL_CASES_TAC` will merely duplicate the original goal twice.

Failure

Fails unless the term `x` has type `bool`.

Example

The goal:

```
?- (b ==> ~b) ==> (b ==> a)
```

can be completely solved by using `BOOL_CASES_TAC` on the variable `b`, then simply rewriting the two subgoals using only the inbuilt tautologies, i.e. by applying the following tactic:

```
BOOL_CASES_TAC (Parse.Term `b:bool`) THEN REWRITE_TAC[]
```

Avoiding fiddly logical proofs by brute-force case analysis, possibly only over a key term as in the above example, possibly over all free boolean variables.

See also

`Tactic.ASM_CASES_TAC`, `Tactic.COND_CASES_TAC`, `Tactic.DISJ_CASES_TAC`, `Tactic.STRUCT_CASES_TAC`

bool_compset (computeLib)

```
computeLib.bool_compset : compset
```

A simplification set for use with `CBV_CONV` for basic computations.

This compset is a simplification set for use with the compute library performing computations about operations on primitive booleans and other basic constants, such as LET, conditional, implication, conjunction, disjunction, and negation.

Example

```
> computeLib.CBV_CONV computeLib.bool_compset (Term `F ==> (T \/ F)`);
val it = ⊢ F ⇒ T ∨ F ⇔ T: thm
```

See also

computeLib.CBV_CONV

bool_EQ_CONV (Conv)

Conv.bool_EQ_CONV : conv

Simplifies expressions involving boolean equality.

The conversion bool_EQ_CONV simplifies equations of the form t1 = t2, where t1 and t2 are of type bool. When applied to a term of the form t = t, the conversion bool_EQ_CONV returns the theorem

```
|- (t = t) = T
```

When applied to a term of the form t = T, the conversion returns

```
|- (t = T) = t
```

And when applied to a term of the form T = t, it returns

```
|- (T = t) = t
```

Failure

Fails unless applied to a term of the form t1 = t2, where t1 and t2 are boolean, and either t1 and t2 are syntactically identical terms or one of t1 and t2 is the constant T.

Example

```
> bool_EQ_CONV (Parse.Term `T = F`);
val it = ⊢ (T ⇔ F) ⇔ F: thm

> bool_EQ_CONV (Parse.Term `(0 < n) = T`);
val it = ⊢ (0 < n ⇔ T) ⇔ 0 < n: thm
```

bool_rewrites (Rewrite)

```
Rewrite.bool_rewrites: rewrites
```

Contains a number of basic equalities useful in rewriting.

The variable `bool_rewrites` is a basic collection of rewrite rules useful in expression simplification. The current collection is

```
- bool_rewrites;

> val it =
    |- (x = x) = T;  |- (T = t) = t;  |- (t = T) = t;  |- (F = t) = ~t;
    |- (t = F) = ~t;  |- ~~t = t;  |- ~T = F;  |- ~F = T;  |- T /\ t = t;
    |- t /\ T = t;  |- F /\ t = F;  |- t /\ F = F;  |- t /\ t = t;
    |- T \/ t = T;  |- t \/ T = T;  |- F \/ t = t;  |- t \/ F = t;
    |- t \/ t = t;  |- T ==> t = t;  |- t ==> T = T;  |- F ==> t = T;
    |- t ==> t = T;  |- t ==> F = ~t;  |- (if T then t1 else t2) = t1;
    |- (if F then t1 else t2) = t2;  |- (!x. t) = t;  |- (?x. t) = t;
    |- (\x. t1) t2 = t1
    Number of rewrite rules = 28
     : rewrites
```

The contents of `bool_rewrites` provide a standard basis upon which to build bespoke rewrite rule sets for use by the functions in `Rewrite`.

See also

`Rewrite.GEN_REWRITE_CONV`, `Rewrite.GEN_REWRITE_RULE`, `Rewrite.GEN_REWRITE_TAC`, `Rewrite.REWRITE_RULE`, `Rewrite.REWRITE_TAC`, `Rewrite.add_rewrites`, `Rewrite.add_implicit_rewrites`, `Rewrite.empty_rewrites`, `Rewrite.implicit_rewrites`, `Rewrite.set_implicit_rewrites`

bool_ss (BasicProvers)

```
BasicProvers.bool_ss : simpset
```

Re-exported from `bossLib.bool_ss`. See that entry for full documentation.

bool_ss (boolSimps)

```
boolSimps.bool_ss : simpset
```

Re-exported from `bossLib.bool_ss`. See that entry for full documentation.

bool_ss (bossLib)

```
bossLib.bool_ss : simpset
```

Also exported as `BasicProvers.bool_ss`, `boolSimps.bool_ss`.

Basic simpset containing standard propositional and first order logic simplifications, plus beta conversion.

The `bool_ss` simpset is almost at the base of the system-provided simpset hierarchy. Though not very powerful, it does include the following ad hoc collection of rewrite rules for propositions and first order terms:

```
|- !A B. ~(A ==> B) = A /\ ~B
|- !A B. (~(A /\ B) = ~A \/ ~B) /\
         (~(A \/ B) = ~A /\ ~B)
|- !P. ~(!x. P x) = ?x. ~P x
|- !P. ~(?x. P x) = !x. ~P x
|- (~p = ~q) = (p = q)
|- !x. (x = x) = T
|- !t. ((T = t) = t) /\
       ((t = T) = t) /\
       ((F = t) = ~t) /\
       ((t = F) = ~t)
```

```
|- (!t. ~~t = t) /\ (~T = F) /\ (~F = T)
|- !t. (T /\ t = t) /\
       (t /\ T = t) /\
       (F /\ t = F) /\
       (t /\ F = F) /\
       (t /\ t = t)
|- !t. (T \/ t = T) /\
       (t \/ T = T) /\
       (F \/ t = t) /\
       (t \/ F = t) /\
       (t \/ t = t)
|- !t. (T ==> t = t) /\
       (t ==> T = T) /\
       (F ==> t = T) /\
       (t ==> t = T) /\
       (t ==> F = ~t)
|- !t1 t2. ((if T then t1 else t2) = t1) /\
           ((if F then t1 else t2) = t2)
|- !t. (!x. t) = t
|- !t. (?x. t) = t
|- !b t. (if b then t else t) = t
|- !a. ?x. x = a
|- !a. ?x. a = x
|- !a. ?!x. x = a,
|- !a. ?!x. a = x,
|- (!b e. (if b then T else e) = b \/ e) /\
   (!b t. (if b then t else T) = b ==> t) /\
   (!b e. (if b then F else e) = ~b /\ e) /\
   (!b t. (if b then t else F) = b /\ t)
|- !t. t \/ ~t
|- !t. ~t \/ t
|- !t. ~(t /\ ~t)
|- !x. (@y. y = x) = x
|- !x. (@y. x = y) = x
|- !f v. (!x. (x = v) ==> f x) = f v
|- !f v. (!x. (v = x) ==> f x) = f v
|- !P a. (?x. (x = a) /\ P x) = P a
|- !P a. (?x. (a = x) /\ P x) = P a
```

Also included in `bool_ss` is a conversion to perform beta reduction, as well as the

following congruence rules, which allow the simplifier to glean additional contextual information as it descends through implications and conditionals.

```
|- !x x' y y'.
    (x = x') ==>
    (x' ==> (y = y')) ==> (x ==> y = x' ==> y')

|- !P Q x x' y y'.
    (P = Q) ==>
    (Q ==> (x = x')) ==>
    (~Q ==> (y = y')) ==> ((if P then x else y) = (if Q then x' else y'))
```

Failure

Can't fail, as it is not a functional value.

The `bool_ss` simpset is an appropriate simpset from which to build new user-defined simpsets. It is also useful in its own right, for example when a delicate simplification is desired, where other more powerful simpsets might cause undue disruption to a goal. If even less system rewriting is desired, the `pure_ss` value can be used.

See also

`pureSimps.pure_ss`, `bossLib.std_ss`, `bossLib.arith_ss`, `bossLib.list_ss`, `bossLib.SIMP_CONV`, `bossLib.SIMP_TAC`, `bossLib.RW_TAC`

BUTFIRSTN_CONV (listLib)

`listLib.BUTFIRSTN_CONV : conv`

Computes by inference the result of dropping the initial n elements of a list.

For any object language list of the form "`[x0;...x(n-k);...;x(n-1)]`" , the result of evaluating

```
BUTFIRSTN_CONV "BUTFIRSTN k [x0;...x(n-k);...;x(n-1)]"
```

is the theorem

```
|- BUTFIRSTN k [x0;...;x(n-k);...;x(n-1)] = [x(n-k);...;x(n-1)]
```

Failure

`BUTFIRSTN_CONV tm` fails if `tm` is not of the form described above, or `k` is greater than the length of the list.

butlast (Lib)

```
Lib.butlast : 'a list -> 'a list
```

Computes the sub-list of a list consisting of all but the last element.

`butlast [x1,...,xn]` returns `[x1,...,x(n-1)]`.

Failure

Fails if the list is empty.

See also

`Lib.last`, `Lib.el`, `Lib.front_last`

BUTLAST_CONV (listLib)

```
listLib.BUTLAST_CONV : conv
```

Computes by inference the result of stripping the last element of a list.

For any object language list of the form "`[x0;...x(n-1)]`" , the result of evaluating

```
BUTLAST_CONV "BUTLAST [x0;...;x(n-1)]"
```

is the theorem

```
|- BUTLAST [x0;...;x(n-1)] = [x0;...; x(n-2)]
```

Failure

`BUTLAST_CONV tm` fails if `tm` is an empty list.

BUTLASTN_CONV (listLib)

```
listLib.BUTLASTN_CONV : conv
```

Computes by inference the result of dropping the last n elements of a list.

For any object language list of the form "`[x0;...x(n-k);...;x(n-1)]`" , the result of evaluating

```
BUTLASTN_CONV "BUTLASTN k [x0;...x(n-k);...;x(n-1)]"
```

is the theorem

```
|- BUTLASTN k [x0;...;x(n-k);...;x(n-1)] = [x0;...;x(n-k-1)]
```

Failure

`BUTLASTN_CONV tm` fails if `tm` is not of the form described above, or `k` is greater than the length of the list.

bvar (Term)

```
Term.bvar : term -> term
```

Returns the bound variable of an abstraction.

If `M` is a lambda abstraction, i.e, has the form `\v. t`, then `bvar M` returns `v`.

Failure

Fails unless `M` is an abstraction.

See also

`Term.body`, `Term.dest_abs`

bvk_find_term (HolKernel)

```
HolKernel.bvk_find_term : (term list * term -> bool) -> (term -> 'a) -> term -> 'a optic
```

Finds a sub-term satisfying predicate argument; applies a continuation.

A call to `bvk_find_term P k tm` searches `tm` for a sub-term satisfying `P` and calls the continuation `k` on the first that it finds. If `k` succeeds on this sub-term, the result is wrapped in `SOME` and returned to the caller. If `k` raises a `HOL_ERR` exception on the sub-term, control returns to `bvk_find_term`, which continues to look for a sub-term satisfying `P`. Other exceptions are returned to the caller. If there is no sub-term that both satisfies `P` and on which `k` operates successfully, the result is `NONE`.

The search order is top-down, left-to-right (i.e., rators of combs are examined before rands).

As with `find_term`, `P` should be total. In addition, `P` is given not just the sub-term of interest, but also the stack of bound variables that have scope over the sub-term, with the innermost bound variables appearing earlier in the list.

Failure

Fails if the predicate argument fails (i.e., raises an exception; returning false is acceptable) on a sub-term, or if the contination argument raises a non-`HOL_ERR` exception on a sub-term on which the predicate has returned true.

Example

The `RED_CONV` function from `reduceLib` reduces a ground arithmetic term over the natural numbers, failing if the term is not of the right shape.

```
- val find = bvk_find_term (equal ``:num`` o type_of o #2)
                           reduceLib.RED_CONV;
> val find = fn : term -> thm option
```

```
- find ``SUC n``;
> val it = NONE : thm option

- find ``2 * 3 + SUC n``;
> val it = SOME |- 2 * 3 = 6 : thm option

- find ``SUC n + 2 * 3``;
> val it = SOME |- 2 * 3 = 6 : thm option

- find ``2 + 1 + SUC n + 2 * 3``;
> val it = SOME |- 2 + 1 = 3 : thm option
```

See also

`HolKernel.find_term`, `HolKernel.find_terms`

by (bossLib)

```
op bossLib.by : term quotation * tactic -> tactic
```

Prove and place a theorem on the assumptions of the goal.

An invocation `tm by tac`, when applied to goal `A ?- g`, applies `tac` to goal `A ?- tm`. If `tm` is thereby proved, it is added to `A`, yielding the new goal `A,tm ?- g`. If `tm` is not proved by `tac`, then the application fails.

When `tm` is added to the existing assumptions `A`, it is "stripped", i.e., broken apart by eliminating existentials, conjunctions, and disjunctions. This can lead to case splitting.

Failure

Fails if `tac` fails when applied to `A ?- tm`, or if `tac` fails to prove that goal.

Example

Given the goal `{x <= y, w < x} ?- P`, suppose that the fact `?n. y = n + w` would help in eventually proving `P`. Invoking

```
`?n. y = n + w` by (EXISTS_TAC ``y-w`` THEN DECIDE_TAC)
```

yields the goal `{y = n + w, x <= y, w < x} ?- P` in which the proved fact has been added to the assumptions after its existential quantifier is eliminated. Note the parentheses around the tactic: this is needed for the example because `by` binds more tightly than `THEN`.

Comments

Use of `by` can be more convenient than `IMP_RES_TAC` and `RES_TAC` when they would generate many useless assumptions.

See also

`bossLib.subgoal`, `bossLib.suffices_by`, `Tactical.SUBGOAL_THEN`, `Tactic.IMP_RES_TAC`, `Tactic.RES_TAC`, `Tactic.STRIP_ASSUME_TAC`

C (Lib)

```
Lib.C : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
```

Permutes first two arguments to curried function: `C f x y` equals `f y x`.

Failure

`C f` never fails and `C f x` never fails, but `C f x y` fails if `f y x` fails.

Example

```
> map (C cons []) [1,2,3];
val it = [[1], [2], [3]]: int list list
```

See also

`Lib.##`, `Lib.I`, `Lib.K`, `Lib.S`, `Lib.W`

can (Lib)

```
Lib.can : ('a -> 'b) -> 'a -> bool
```

Tests for failure.

`can f x` evaluates to `true` if the application of `f` to `x` succeeds. It evaluates to `false` if the application fails.

Failure

Only fails if `f x` raises the `Interrupt` exception.

Example

```
> hd [];
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
Empty raised

> can hd [];
val it = false: bool

> can (fn _ => raise Interrupt) 3;
Exception- Interrupt raised
```

See also

`Lib.assert`, `Lib.trye`, `Lib.partial`, `Lib.total`, `Lib.assert_exn`

CASE_TAC (BasicProvers)

```
BasicProvers.CASE_TAC : tactic
```

Case splits on a term `t` that features in the goal as `case t of ...`, and then performs some simplification.

`BasicProvers.CASE_TAC` first calls `BasicProvers.PURE_CASE_TAC`, which searches the goal for an instance of `case t of ...` and performs a `BasicProvers.Cases_on `t``. If this succeeds, it then simplifies the goal using definitions of `case` constants, plus distinctness and injectivity theorems for datatypes.

Comments

When there are multiple `case` constants in the goal, it can be very convenient to execute the tactic `REPEAT CASE_TAC`. `bossLib.CASE_TAC` is the same as `BasicProvers.CASE_TAC`.

Failure

`BasicProvers.CASE_TAC` fails precisely when `BasicProvers.PURE_CASE_TAC` fails.

See also

`BasicProvers.PURE_CASE_TAC`

Cases (BasicProvers)

```
BasicProvers.Cases : tactic
```

Re-exported from `bossLib.Cases`. See that entry for full documentation.

Cases (bossLib)

```
bossLib.Cases : tactic
```

Also exported as `BasicProvers.Cases`.

Performs case analysis on the variable of the leading universally quantified variable of the goal.

When applied to a universally quantified goal `?- !u. G`, `Cases` performs a case-split, based on the cases theorem for the type of `u` stored in the global `TypeBase` database.

The cases theorem for a type `ty` will be of the form:

```
|- !v:ty. (?x11...x1n1. v = C1 x11 ... x1n1) \/ .... \/
          (?xm1...xmnm. v = Cm xm1 ... xmnm)
```

where there is no requirement for there to be more than one disjunct, nor for there to be any particular number of existentially quantified variables in any disjunct. For example, the cases theorem for natural numbers initially in the `TypeBase` is:

```
|- !n. (n = 0) \/ (?m. n = SUC m)
```

Case-splitting consists of specialising the cases theorem with the variable from the goal and then generating as many sub-goals as there are disjuncts in the cases theorem, where in each sub-goal (including the assumptions) the variable has been replaced by an expression involving the given 'constructor' (the `Ci`'s above) applied to as many fresh variables as appropriate.

Failure

Fails if the goal is not universally quantified, or if the type of the universally quantified variable does not have a case theorem in the `TypeBase`, as will happen, for example, with variable types.

Example

If we have defined the following type:

```
- Hol_datatype `foo = Bar of num | Baz of bool`;
> val it = () : unit
```

and the following function:

```
- val foofn_def = Define `(foofn (Bar n) = n + 10) /\
                          (foofn (Baz x) = 10)`;
```

```
> val foofn_def =
    |- (!n. foofn (Bar n) = n + 10) /\
        !x. foofn (Baz x) = 10 : thm
```

then it is possible to make progress with the goal `!x. foofn x >= 10` by applying the tactic `Cases`, thus:

```
                  ?- !x. foofn x >= 10
=======================================================  Cases
 ?- foofn (Bar n) >= 10        ?- foofn (Baz b) >= 10
```

producing two new goals, one for each constructor of the type.

See also

`bossLib.Cases_on`, `bossLib.Induct`, `Tactic.STRUCT_CASES_TAC`

Cases_on (BasicProvers)

```
BasicProvers.Cases_on : term quotation -> tactic
```

Re-exported from `bossLib.Cases_on`. See that entry for full documentation.

Cases_on (bossLib)

```
bossLib.Cases_on : term quotation -> tactic
```

Also exported as `BasicProvers.Cases_on`.

Performs case analysis on the type of a given term.

An application `Cases_on M` performs a case-split based on the type `ty` of `M`, using the cases theorem for `ty` from the global `TypeBase` database.

`Cases_on` can be used to specify variables that are buried in the quantifier prefix. `Cases_on` can also be used to perform case splits on non-variable terms. If `M` is a non-variable term that does not occur bound in the goal, then the cases theorem is instantiated with `M` and used to generate as many sub-goals as there are disjuncts in the cases theorem.

Failure

Fails if `ty` does not have a case theorem in the `TypeBase`.

Example

None yet.

See also

`bossLib.Cases`, `bossLib.Induct`, `bossLib.Induct_on`, `Tactic.STRUCT_CASES_TAC`

CASES_THENL (Thm_cont)

```
Thm_cont.CASES_THENL : (thm_tactic list -> thm_tactic)
```

Applies the theorem-tactics in a list to corresponding disjuncts in a theorem.

When given a list of theorem-tactics `[ttac1;...;ttacn]` and a theorem whose conclusion is a top-level disjunction of `n` terms, `CASES_THENL` splits a goal into `n` subgoals resulting from applying to the original goal the result of applying the `i`'th theorem-tactic to the `i`'th disjunct. This can be represented as follows, where the number of existentially quantified variables in a disjunct may be zero. If the theorem `th` has the form:

```
A' |- ?x11..x1m. t1 \/ ... \/ ?xn1..xnp. tn
```

where the number of existential quantifiers may be zero, and for all `i` from `1` to `n`:

```
  A ?- s
========== ttaci (|- ti[xi1'/xi1]..[xim'/xim])
 Ai ?- si
```

where the primed variables have the same type as their unprimed counterparts, then:

```
          A ?- s
========================  CASES_THENL [ttac1;...;ttacn] th
 A1 ?- s1  ...  An ?- sn
```

Unless `A'` is a subset of `A`, this is an invalid tactic.

Failure

Fails if the given theorem does not, at the top level, have the same number of (possibly multiply existentially quantified) disjuncts as the length of the theorem-tactic list (this includes the case where the theorem-tactic list is empty), or if any of the tactics generated as specified above fail when applied to the goal.

Performing very general disjunctive case splits.

See also

`Thm_cont.DISJ_CASES_THENL`, `Thm_cont.X_CASES_THENL`

CBV_CONV (computeLib)

```
computeLib.CBV_CONV : compset -> conv
```

Call by value rewriting.

The conversion `CBV_CONV` expects a simplification set and a term. Its term argument is rewritten using the equations added in the simplification set. The strategy used is somewhat similar to ML's, that is call-by-value (arguments of constants are completely reduced before the rewrites associated to the constant are applied) with weak reduction (no reduction of the function body before the function is applied). The main differences are that beta-redexes are reduced with a call-by-name strategy (the argument is not reduced), and reduction under binders is done when it occurs in a position where it cannot be substituted.

The simplification sets are mutable objects, this means they are extended by side-effect. The function `new_compset` will create a new set containing reflexivity (`REFL_CLAUSE`), plus the supplied rewrites. Theorems can be added to an existing compset with the function `add_thms`.

This function (`add_thms`) scans the supplied theorems using `BODY_CONJUNCTS`. Let `thm` be one such element. If `thm` is of the form `P1 ⇒ P2 ⇒ ... ⇒ t` for possibly-zero implications, then proccess `t`. If `t` is an equation, add it as a reduction rule. If `t` is of the form `¬t'`, then add the rule `t ⇔ F`, otherwise add the rule `t ⇔ T`. If there is at least one implication then also add `P1 ⇒ P2 ⇒ ... ⇒ t ⇔ T`.

It is also possible to add conversions to a simplification set with `add_conv`. The only restriction is that a constant (`c`) and an arity (`n`) must be provided. The conversion will be called only on terms in which `c` is applied to `n` arguments.

Two theorem "preprocessors" are provided to control the strictness of the arguments of a constant. `lazyfy_thm` has pattern variables on the left hand side turned into abstractions on the right hand side. This transformation is applied on every conjunct, and removes prenex universal quantifications. A typical example is `COND_CLAUSES`:

```
(COND T a b = a) /\ (COND F a b = b)
```

Using these equations is very inefficient because both `a` and `b` are evaluated, regardless of the value of the boolean expression. It is better to use `COND_CLAUSES` with the form above

```
(COND T = \a b. a) /\ (COND F = \a b. b)
```

The call-by-name evaluation of beta redexes avoids computing the unused branch of the conditional.

Conversely, `strictify_thm` does the reverse transformation. This is particularly relevant for `LET_DEF`:

```
LET = \f x. f x   -->   LET f x = f x
```

This forces the evaluation of the argument before reducing the beta-redex. Hence the usual behaviour of `LET`.

It is necessary to provide rules for all the constants appearing in the expression to reduce (all also for those that appear in the right hand side of a rule), unless the given constant is considered as a constructor of the representation chosen. As an example, `reduceLib.num_compset` creates a new simplification set with all the rules needed for basic boolean and arithmetical calculations built in.

Example

```
- val rws = computeLib.new_compset [computeLib.lazyfy_thm COND_CLAUSES];
> val rws = <compset> : compset

- computeLib.CBV_CONV rws ``(\x.x) ((\x.x) if T then 0+0 else 10)``;
> val it = |- (\x. x) ((\x. x) (if T then 0 + 0 else 10)) = 0 + 0 : thm
```

```
- computeLib.CBV_CONV reduceLib.num_compset
          ``if 100 - 5 * 5 < 80  then 2 EXP 16 else 3``;
> val it = |- (if 100 - 5 * 5 < 80 then 2 ** 16 else 3) = 65536 : thm
```

Failing to give enough rules may make `CBV_CONV` build a huge result, or even loop. The same may occur if the initial term to reduce contains free variables.

```
val eqn = bossLib.Define `exp n p = if p=0 then 1 else n * (exp n (p-1))`;
val _ = computeLib.add_thms [eqn] rws;

- computeLib.CBV_CONV rws ``exp 2 n``;
> Interrupted.

- computeLib.set_skip rws ``COND`` (SOME 1);
> val it = () : unit

- computeLib.CBV_CONV rws ``exp 2 n``;
> val it = |- exp 2 n = if n = 0 then 1 else 2 * exp 2 (n - 1) : thm
```

The first invocation of `CBV_CONV` loops since the exponent never reduces to 0. Below the first steps are computed:

```
exp 2 n
if n = 0 then 1 else 2 * exp 2 (n-1)
if n = 0 then 1 else 2 * if (n-1) = 0 then 1 else 2 * exp 2 (n-1-1)
...
```

The call to `set_skip` means that if the constants `COND` appears applied to one argument and does not create a redex (in the example, if the condition does not reduce to `T` or `F`), then the forthcoming arguments (the two branches of the conditional) are not reduced at all.

Failure

Should never fail. Nonetheless, using rewrites with assumptions may cause problems when rewriting under abstractions. The following example illustrates that issue.

```
- val th = ASSUME “0 = x”;
```

```
- val tm = Term`\(x:num). x = 0`;
- val rws = from_list [th];
- CBV_CONV rws tm;
```

This fails because the 0 is replaced by `x`, making the assumption `0 = x`. Then, the abstraction cannot be rebuilt since `x` appears free in the assumptions.

See also

`numLib.REDUCE_CONV`, `computeLib.bool_compset`, `bossLib.EVAL`

CCONTR (Thm)

```
Thm.CCONTR : term -> thm -> thm
```

Implements the classical contradiction rule.

When applied to a term `t` and a theorem `A |- F`, the inference rule `CCONTR` returns the theorem `A - {~t} |- t`.

```
     A |- F
---------------  CCONTR t
 A - {~t} |- t
```

Failure

Fails unless the term has type `bool` and the theorem has `F` as its conclusion.

Comments

The usual use will be when `~t` exists in the assumption list; in this case, `CCONTR` corresponds to the classical contradiction rule: if `~t` leads to a contradiction, then `t` must be true.

See also

`Drule.CONTR`, `Drule.CONTRAPOS`, `Tactic.CONTR_TAC`, `Thm.NOT_ELIM`

CCONTR_TAC (Tactic)

```
Tactic.CCONTR_TAC : tactic
```

Assumes the negation of the conclusion of a goal.

Given a goal `A ?- t`, `CCONTR_TAC` makes a new goal which is to prove `F` by assuming also the negation of the conclusion `t`.

```
  A ?- t
==========
A, -t ?- F
```

Failure

Never fails

See also

`Tactic.CHECK_ASSUME_TAC`, `Thm.CCONTR`, `Drule.CONTRAPOS`, `Thm.NOT_ELIM`

CHANGED_CONSEQ_CONV (ConseqConv)

```
ConseqConv.CHANGED_CONSEQ_CONV : (conseq_conv -> conseq_conv)
```

Makes a consequence conversion fail if applying it leaves a term unchanged.

If `c` is a consequence conversion that maps a term ``t`` to a theorem `|- t = t'`, `|- t' ==> t` or `|- t ==> t'`, where `t'` is alpha-equivalent to `t`, or if `c` raises the `UNCHANGED` exception when applied to ``t``, then `CHANGED_CONSEQ_CONV c` fails when applied to the term ``t``. Otherwise, `CHANGED_CONSEQ_CONV c` behaves like `c`.

See also

`Conv.CHANGED_CONV`, `ConseqConv.QCHANGED_CONSEQ_CONV`

CHANGED_CONV (Conv)

```
Conv.CHANGED_CONV : (conv -> conv)
```

Makes a conversion fail if applying it leaves a term unchanged.

If `c` is a conversion that maps a term ``t`` to a theorem `|- t = t'`, where `t'` is alpha-equivalent to `t`, or if `c` raises the `UNCHANGED` exception when applied to ``t``, then `CHANGED_CONV c` is a conversion that fails when applied to the term ``t``. If `c` maps ``t`` to `|- t = t'`, where `t'` is not alpha-equivalent to `t`, then `CHANGED_CONV c` also maps ``t`` to `|- t = t'`. That is, `CHANGED_CONV c` is the conversion that behaves exactly like `c`, except that it fails whenever the conversion `c` would leave its input term unchanged (up to alpha-equivalence).

When `CHANGED_CONV c t` fails, it raises an exception `HOL_ERR ...`, not `UNCHANGED`, since some enclosing functions handle the `UNCHANGED` exception as though `c` had succeeded by returning the theorem `|- t = t`.

Failure

`CHANGED_CONV c ``t`` ` fails if `c` maps ``t`` to `|- t = t'`, where `t'` is alpha-equivalent to `t`, or if `c` raises the `UNCHANGED` exception when applied to ``t``, or if `c` fails when applied to ``t``. The function returned by `CHANGED_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

`CHANGED_CONV` is used to transform a conversion that may leave terms unchanged, and therefore may cause a nonterminating computation if repeated, into one that can safely be repeated until application of it fails to substantially modify its input term.

See also

`Conv.UNCHANGED`, `Conv.QCHANGED_CONV`

CHANGED_TAC (Tactical)

```
Tactical.CHANGED_TAC : (tactic -> tactic)
```

Makes a tactic fail if it has no effect.

When applied to a tactic `T`, the tactical `CHANGED_TAC` gives a new tactic which is the same as `T` if that has any effect, and otherwise fails.

Failure

The application of `CHANGED_TAC` to a tactic never fails. The resulting tactic fails if the basic tactic either fails or has no effect.

See also

`Tactical.TRY`, `Tactical.VALID`

cheat (bossLib)

```
bossLib.cheat : tactic
```

Discharge a goal without proving it.

The `cheat` tactic solves the current goal immediately without proving it, using `mk_oracle_thm` and adding the `"cheat"` tag to any theorem thereby obtained.

Failure

Never fails.

Comments

The intended use of `cheat` is to temporarily plug gaps in large theory developments in order to sketch the bigger picture before filling in the details. It can be useful as a kind of high-level `SUFF_TAC`: cheat on a difficult lemma, see whether it works as intended for the main theorem, then go back and prove the lemma properly.

The usual caveats associated with `mk_oracle_thm` apply: cheating exposes you to the possibility of false theorems and contradictions. To be sure a theorem was proved without cheating, check its tags.

See also

Thm.mk_oracle_thm, Thm.tag, Globals.show_tags, Tactic.SUFF_TAC

CHECK_ASSUME_TAC (Tactic)

```
Tactic.CHECK_ASSUME_TAC : thm_tactic
```

Adds a theorem to the assumption list of goal, unless it solves the goal.

When applied to a theorem `A' |- s` and a goal `A ?- t`, the tactic `CHECK_ASSUME_TAC` checks whether the theorem will solve the goal (this includes the possibility that the theorem is just `A' |- F`). If so, the goal is duly solved. If not, the theorem is added to the assumptions of the goal, unless it is already there.

```
    A ?- t
==============  CHECK_ASSUME_TAC (A' |- F)   [special case 1]


    A ?- t
==============  CHECK_ASSUME_TAC (A' |- t)   [special case 2]


    A ?- t
==============  CHECK_ASSUME_TAC (A' |- s)   [general case]
 A u {s} ?- t
```

Unless `A'` is a subset of `A`, the tactic will be invalid, although it will not fail.

Failure

Never fails.

See also

Tactic.ACCEPT_TAC, Tactic.ASSUME_TAC, Tactic.CONTR_TAC, Tactic.DISCARD_TAC, Tactic.MATCH_ACCEPT_TAC

CHOOSE (Thm)

```
Thm.CHOOSE : term * thm -> thm -> thm
```

Eliminates existential quantification using deduction from a particular witness.

When applied to a term-theorem pair `(v,A1 |- ?x. s)` and a second theorem of the form `A2 u {s[v/x]} |- t`, the inference rule `CHOOSE` produces the theorem `A1 u A2 |- t`.

```
 A1 |- ?x. s        A2 u {s[v/x]} |- t
-------------------------------------  CHOOSE (v,(A1 |- ?x. s))
            A1 u A2 |- t
```

Where `v` is not free in `A1`, `A2` or `t`.

Failure

Fails unless the terms and theorems correspond as indicated above; in particular `v` must have the same type as the variable existentially quantified over, and must not be free in `A1`, `A2` or `t`.

See also

`Tactic.CHOOSE_TAC`, `Thm.EXISTS`, `Tactic.EXISTS_TAC`, `Drule.SELECT_ELIM`

CHOOSE_TAC (Tactic)

```
Tactic.CHOOSE_TAC : thm_tactic
```

Adds the body of an existentially quantified theorem to the assumptions of a goal.

When applied to a theorem `A' |- ?x. t` and a goal, `CHOOSE_TAC` adds `t[x'/x]` to the assumptions of the goal, where `x'` is a variant of `x` which is not free in the goal or assumption list; normally `x'` is just `x`.

```
       A ?- u
====================  CHOOSE_TAC (A' |- ?x. t)
 A u {t[x'/x]} ?- u
```

Unless `A'` is a subset of `A`, this is not a valid tactic.

Failure

Fails unless the given theorem is existentially quantified.

Example

Suppose we have a goal asserting that the output of an electrical circuit (represented as a boolean-valued function) will become high at some time:

```
?- ?t. output(t)
```

and we have the following theorems available:

```
t1 = |- ?t. input(t)
t2 = !t. input(t) ==> output(t+1)
```

Then the goal can be solved by the application of:

```
CHOOSE_TAC th1
  THEN EXISTS_TAC (Term `t+1`)
  THEN UNDISCH_TAC (Term `input (t:num) :bool`)
  THEN MATCH_ACCEPT_TAC th2
```

Comments

To do similarly with several existentially quantified variables, use `REPEAT_TCL CHOOSE_THEN ASSUME_TAC` in place of `CHOOSE_TAC`

See also

`Thm_cont.CHOOSE_THEN`, `Tactic.X_CHOOSE_TAC`, `Thm_cont.REPEAT_TCL`

CHOOSE_THEN (Thm_cont)

```
Thm_cont.CHOOSE_THEN : thm_tactical
```

Applies a tactic generated from the body of existentially quantified theorem.

When applied to a theorem-tactic `ttac`, an existentially quantified theorem `A' |- ?x. t`, and a goal, `CHOOSE_THEN` applies the tactic `ttac (t[x'/x] |- t[x'/x])` to the goal, where `x'` is a variant of `x` chosen not to be free in the assumption list of the goal. Thus if:

```
 A ?- s1
=========  ttac (t[x'/x] |- t[x'/x])
 B ?- s2
```

then

```
 A ?- s1
==========  CHOOSE_THEN ttac (A' |- ?x. t)
 B ?- s2
```

This is invalid unless `A'` is a subset of `A`.

Failure

Fails unless the given theorem is existentially quantified, or if the resulting tactic fails when applied to the goal.

Example

This theorem-tactical and its relatives are very useful for using existentially quantified theorems. For example one might use the inbuilt theorem

```
LESS_ADD_1 = |- !m n. n < m ==> (?p. m = n + (p + 1))
```

to help solve the goal

```
?- x < y ==> 0 < y * y
```

by starting with the following tactic

```
DISCH_THEN (CHOOSE_THEN SUBST1_TAC o MATCH_MP LESS_ADD_1)
```

which reduces the goal to

```
?- 0 < ((x + (p + 1)) * (x + (p + 1)))
```

which can then be finished off quite easily, by, for example:

```
REWRITE_TAC[ADD_ASSOC, SYM (SPEC_ALL ADD1),
            MULT_CLAUSES, ADD_CLAUSES, LESS_0]
```

Comments

As CHOOSE_THEN scans a theorem's hypotheses as it generates a fresh name to replace the existential, and then internally produces a fresh theorem with an additional hypothesis, this function can be inefficient if used repeatedly within code; in such cases, code should probably use X_CHOOSE_THEN instead.

See also

Tactic.CHOOSE_TAC, Thm_cont.X_CHOOSE_THEN

cj (Drule)

```
Drule.cj : int -> thm -> thm
```

Returns the i'th conjunct of a "guarded" theorem

A call to `cj i th`, where `th` has the form

```
|- !x1 .. xn. p1 /\ .. /\ pm ==> !y... q1 /\ .. ==> ... ==>
                                    c1 /\ c2 /\ ... ck
```

returns the theorem

```
|- !x1 .. xn. p1 /\ .. /\ pm ==> !y... q1 /\ .. ==> ... ==> ci
```

Note that the indexing starts at 1. The conjuncts are stripped apart without regard to the way in which they are associated, as per the behaviour of CONJUNCTS.

Failure

Fails if the conclusion of the guarded theorem does not contain at least `i` conjuncts. A bare term is always considered to be 1 conjunct.

See also

`Drule.BODY_CONJUNCTS`, `Drule.CONJUNCTS`, `Drule.underAIs`

class (DB)

```
DB.datatype class
```

Datatype for classifying theory elements.

Many of the functions in the `DB` structure return answers that involve the `class` type, which is declared as

```
datatype class = Thm | Axm | Def
```

When occurring with `th`, an ML value of type `thm`, `Axm` means that `th` has been asserted as an axiom; `Def` means that `th` is a constant definition; and `Thm` means that `th` is a plain old theorem, i.e,. not an axiom or a definition.

See also

`DB.data`

clear_overloads (term_grammar)

```
term_grammar.clear_overloads : term_grammar.grammar -> term_grammar.grammar
```

Remove non-trivial overloading from a term grammar

For a term grammar `tmG`, `clear_overloads tmG` is the similar grammar, changed to remove non-trivial overloading. (Each constant remains overloaded with itself, which avoids the printing of the theory name for every constant).

Sometimes overloading can be too helpful, when we would like to see the structure of a term (eg, in finding which theorems could simplify it).

Example

In this example we obtain the current type and term grammars `tyG` and `tmG`, then reset the current grammars to be these, except with overloading cleared from the term grammar. We print some theorems (eg, to view their internal structure), and finally we reset the current grammars to the original ones.

```
> ratTheory.RATND_RAT_OF_NUM;
val it =
   ⊢ rat$RATN (rat$rat_of_num n) = integer$int_of_num n ∧
     rat$RATD (rat$rat_of_num n) = 1: thm
> rich_listTheory.MEM_TAKE;
val it = ⊢ ∀l m x. MEM x (TAKE m l) ⇒ MEM x l: thm

> ratTheory.RATND_RAT_OF_NUM;
val it =
   ⊢ rat$RATN (rat$rat_of_num n) = integer$int_of_num n ∧
     rat$RATD (rat$rat_of_num n) = 1: thm
> rich_listTheory.MEM_TAKE;
val it = ⊢ ∀l m x. MEM x (TAKE m l) ⇒ MEM x l: thm
```

Comments

To print just a few terms without overloading, `print_without_macros` may be easier.

See also

`Parse.print_without_macros`, `Parse.current_grammars`, `Parse.temp_set_grammars`

clear_overloads_on (Parse)

```
Parse.clear_overloads_on : string -> unit
```

Clears all overloading on the specified operator.

This function removes all overloading associated with the given string, except those "overloads" that map the string to constants of the same name. These additional overloads (there may be more than one constant of the same name, as long as each such is part of a different theory) may be removed with `remove_ovl_mapping`, or by using `hide`.

Failure

Never fails. If a string is not overloaded, this function simply has no effect.

Example

```
> realTheory.REAL_INV_LT1;
val it =
   ⊢ ∀x. realax$real_lt (realax$real_of_num 0) x ∧
         realax$real_lt x (realax$real_of_num 1) ⇒
         realax$real_lt (realax$real_of_num 1) (realax$inv x): thm
> clear_overloads_on "<";
val it = (): unit
> realTheory.REAL_INV_LT1;
val it =
   ⊢ ∀x. realax$real_lt (realax$real_of_num 0) x ∧
         realax$real_lt x (realax$real_of_num 1) ⇒
         realax$real_lt (realax$real_of_num 1) (realax$inv x): thm
> clear_overloads_on "&";
val it = (): unit
> realTheory.REAL_INV_LT1;
val it =
   ⊢ ∀x. realax$real_lt (realax$real_of_num 0) x ∧
         realax$real_lt x (realax$real_of_num 1) ⇒
         realax$real_lt (realax$real_of_num 1) (realax$inv x): thm
```

If overloading gets too confusing, this function should help to clear away one layer of supposedly helpful obfuscation.

Comments

As with other parsing functions, there is a sister function, `temp_clear_overloads_on` that does the same thing, but whose effect is not saved to a theory file.

See also

`Parse.overload_on`, `Parse.remove_ovl_mapping`

CNF_CONV (normalForms)

`normalForms.CNF_CONV : conv`

Converts a formula into Conjunctive Normal Form (CNF).

Given a formula consisting of truths, falsities, conjunctions, disjunctions, negations, equivalences, conditionals, and universal and existential quantifiers, `CNF_CONV` will convert it to the canonical form:

```
?a_1 ... a_k.
  (!v_1 ... v_m1. P_1 \/ ... \/ P_n1) /\
  ...                                 /\
  (!v_1 ... v_mp. P_1 \/ ... \/ P_np)
```

The `P_ij` are literals: possibly-negated atoms. In first-order logic an atom is a formula consisting of a top-level relation symbol applied to first-order terms: function symbols and variables. In higher-order logic there is no distinction between formulas and terms, so the concept of atom is not well-formed. Note also that the `a_i` existentially bound variables may be functions, as a result of Skolemization.

Failure

`CNF_CONV` should never fail.

Example

```
> normalForms.CNF_CONV ``!x. P x ==> ?y z. Q y \/ ~?z. P z /\ Q z``;
val it =
   ⊢ (∀x. P x ⇒ ∃y z. Q y ∨ ¬∃z. P z ∧ Q z) ⇔
     ∃y. ∀x z. ¬Q z ∨ ¬P z ∨ Q (y x) ∨ ¬P x: thm
```

Example

```
> normalForms.CNF_CONV ``~(~(x = y) = z) = ~(x = ~(y = z))``;
val it = ⊢ (((x ⇎ y) ⇎ z) ⇔ (x ⇎ (y ⇎ z))) ⇔ T: thm
```

COMB2_CONV (Conv)

```
Conv.COMB2_CONV : conv * conv -> conv
```

Applies two conversions to an application's subterms.

A call to `COMB2_CONV(c1,c2) t`, when `t` is an application term of the form `f x`, causes conversion `c1` to be applied to term `f`, and conversion `c2` to be applied to term `x`. If the results of these calls are theorems of the form `|- f = f'` and `|- x = x'`, then the result of the call to `COMB2_CONV` is the theorem `|- f x = f' x'`.

If one of the two sub-calls raises the `UNCHANGED` exception, then the result of that call is taken to be the reflexive theorem (`|- x = x` if `c2` raises the exception, for example). If both conversions raise the `UNCHANGED` exception, then so too does `COMB2_CONV(c1,c2) t`.

Failure

Fails if the term is not a combination term, or if either conversion fails when applied to the respective sub-terms.

Example

```
> COMB2_CONV (ALL_CONV, numLib.REDUCE_CONV) ``f (10 * 3)``;
```

```
<<HOL message: inventing new type variable names: 'a>>
val it = ⊢ f (10 * 3) = f 30: thm
```

See also

Conv.ABS_CONV, Conv.COMB_CONV, Conv.FORK_CONV, Conv.RAND_CONV, Conv.RATOR_CONV

COMB_CONV (Conv)

```
Conv.COMB_CONV : conv -> conv
```

Applies a conversion to both immediate sub-terms of an application.

If `t` is an application term of the form `f x`, and `c` is a conversion, such that `c` maps `f` to `|- f = f'` and `x` to `|- x = x'`, then `COMB_CONV c` maps `t` to `|- f x = f' x'`.

If one of the two sub-calls raises the `UNCHANGED` exception, then the result of that call is taken to be the reflexive theorem (`|- x = x` if `c x` raises the exception, for example). If both conversions raise the `UNCHANGED` exception, then so too does `COMB_CONV c t`.

Failure

`COMB_CONV c t` fails if `t` is not an application term, or if `c` fails when applied to the rator and rand of `t`, or if `c` is not in fact a conversion (i.e., a function which maps terms `t` to a theorem `|- t = t'`).

See also

Conv.ABS_CONV, Conv.COMB2_CONV, Conv.SUB_CONV

combine (Lib)

```
Lib.combine : 'a list * 'b list -> ('a * 'b) list
```

Transforms a pair of lists into a list of pairs.

`combine ([x1,...,xn],[y1,...,yn])` returns `[(x1,y1),...,(xn,yn)]`.

Failure

Fails if the two lists are of different lengths.

Comments

Has much the same effect as the SML Basis function `ListPair.zip` except that it fails if the arguments are not of equal length. Also note that `zip` is a curried version of `combine`

See also

`Lib.zip`, `Lib.unzip`, `Lib.split`

commafy (Lib)

```
Lib.commafy : string list -> string list
```

Add commas into a list of strings.

An application `commafy [s1,...,sn]` yields `[s1, ",", ..., ",", sn]`.

Failure

Never fails.

Example

```
> commafy ["donkey", "mule", "horse", "camel", "llama"];
val it =
   ["donkey", ", ", "mule", ", ", "horse", ", ", "camel", ", ", "llama"]:
   string list
```

```
> String.concat it;
val it = "donkey, mule, horse, camel, llama": string

> commafy ["foo"];
val it = ["foo"]: string list
```

compare (Term)

```
Term.compare : term * term -> order
```

Ordering on terms.

An invocation `compare (M,N)` will return one of {`LESS`, `EQUAL`, `GREATER`}, according to an ordering on terms. The ordering is transitive and total, and equates alpha-convertible terms.

Failure

Never fails.

Example

```
> compare (T,F);
val it = GREATER: order

> compare (Term `\x y. x /\ y`, Term `\y z. y /\ z`);
val it = EQUAL: order
```

Comments

Used to build high performance datastructures for dealing with sets having many terms.

See also

`Term.empty_tmset`, `Term.var_compare`

compare (Type)

```
Type.compare : hol_type * hol_type -> order
```

An ordering on HOL types.

An invocation `compare (ty1,ty2)` returns one of `{LESS, EQUAL, GREATER}`. This is a total and transitive order.

Failure

Never fails.

Example

```
> Type.compare (bool, alpha --> alpha);
val it = LESS: order
```

Comments

One use of `compare` is to build efficient set or dictionary datastructures involving HOL types in the keys.

There is also a `Term.compare`.

See also

`Term.compare`

completeInduct_on (bossLib)

```
bossLib.completeInduct_on : term quotation -> tactic
```

Perform complete induction.

If `q` parses into a well-typed term `M`, an invocation `completeInduct_on q` begins a proof by complete (also known as 'course-of-values') induction on `M`. The term `M` should occur free in the current goal.

Failure

If `M` does not parse into a term or does not occur free in the current goal.

Example

Suppose we wish to prove that every number not equal to one has a prime factor:

```
!n. ~(n = 1) ==> ?p. prime p /\ p divides n
```

A natural way to prove this is by complete induction. Invoking `completeInduct_on `n`` yields the goal

```
{ !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m }
?-
~(n = 1) ==> ?p. prime p /\ p divides n
```

See also

`bossLib.measureInduct_on`, `bossLib.Induct`, `bossLib.Induct_on`

concl (Thm)

```
Thm.concl : thm -> term
```

Returns the conclusion of a theorem.

When applied to a theorem `A |- t`, the function `concl` returns `t`.

Failure

Never fails.

See also

`Thm.dest_thm`, `Thm.hyp`

COND_CASES_TAC (Tactic)

```
Tactic.COND_CASES_TAC : tactic
```

Induces a case split on a conditional expression in the goal.

`COND_CASES_TAC` searches for a conditional sub-term in the term of a goal, i.e. a sub-term of the form `p=>u|v`, choosing one by its own criteria if there is more than one. It then induces a case split over `p` as follows:

```
                          A ?- t
 ============================================================  COND_CASES_TAC
  A u {p} ?- t[u/(p=>u|v),T/p]   A u {~p} ?- t[v/(p=>u|v),F/p]
```

where `p` is not a constant, and the term `p=>u|v` is free in `t`. Note that it both enriches the assumptions and inserts the assumed value into the conditional.

Failure

`COND_CASES_TAC` fails if there is no conditional sub-term as described above.

Example

For `"x"`, `"y"`, `"z1"` and `"z2"` of type `":*"`, and `"P:*->bool"`,

```
COND_CASES_TAC ([], "x = (P y => z1 | z2)");;
([(["P y"], "x = z1"); (["~P y"], "x = z2")], -) : subgoals
```

but it fails, for example, if `"y"` is not free in the term part of the goal:

```
COND_CASES_TAC ([], "!y. x = (P y => z1 | z2)");;
evaluation failed     COND_CASES_TAC
```

In contrast, ASM_CASES_TAC does not perform the replacement:

```
ASM_CASES_TAC "P y" ([], "x = (P y => z1 | z2)");;
([(["P y"], "x = (P y => z1 | z2)"); (["~P y"], "x = (P y => z1 | z2)")],
 -)
: subgoals
```

Useful for case analysis and replacement in one step, when there is a conditional sub-term in the term part of the goal. When there is more than one such sub-term and one in particular is to be analyzed, COND_CASES_TAC cannot be depended on to choose the 'desired' one. It can, however, be used repeatedly to analyze all conditional sub-terms of a goal.

See also

Tactic.ASM_CASES_TAC, Tactic.DISJ_CASES_TAC, Tactic.STRUCT_CASES_TAC

COND_CONV (Conv)

Conv.COND_CONV : conv

Simplifies conditional terms.

The conversion COND_CONV simplifies a conditional term "c => u | v" if the condition c is either the constant T or the constant F or if the two terms u and v are equivalent up to alpha-conversion. The theorems returned in these three cases have the forms:

```
|- (T => u | v) = u

|- (F => u | v) = u

|- (c => u | u) = u
```

Failure

COND_CONV tm fails if tm is not a conditional "c => u | v", where c is T or F, or u and v are alpha-equivalent.

COND_CONV (reduceLib)

```
reduceLib.COND_CONV : conv
```

Simplifies certain conditional expressions.

If `tm` corresponds to one of the forms given below, where `b` has type `bool` and `t1` and `t2` have the same type, then `COND_CONV tm` returns the corresponding theorem. Note that in the last case the arms need only be alpha-equivalent rather than strictly identical.

```
COND_CONV "F => t1 | t2" = |- (T => t1 | t2) = t2
COND_CONV "T => t1 | t2" = |- (T => t1 | t2) = t1
COND_CONV "b => t | t    = |- (b => t | t) = t
```

Failure

`COND_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
#COND_CONV "F => F | T";;
|- (F => F | T) = T

#COND_CONV "T => F | T";;
|- (T => F | T) = F

#COND_CONV "b => (\x. SUC x) | (\p. SUC p)";;
|- (b => (\x. SUC x) | (\p. SUC p)) = (\x. SUC x)
```

COND_ELIM_CONV (Arith)

```
Arith.COND_ELIM_CONV : conv
```

Eliminates conditional statements from a formula.

This function moves conditional statements up through a term and if at any point the branches of the conditional become Boolean-valued the conditional is eliminated. If the term is a formula, only an abstraction can prevent a conditional being moved up far enough to be eliminated.

Failure

Never fails.

Example

```
#COND_ELIM_CONV "!f n. f ((SUC n = 0) => 0 | (SUC n - 1)) < (f n) + 1";;
|- (!f n. (f((SUC n = 0) => 0 | (SUC n) - 1)) < ((f n) + 1)) =
   (!f n.
     (~(SUC n = 0) \/ (f 0) < ((f n) + 1)) /\
     ((SUC n = 0) \/ (f((SUC n) - 1)) < ((f n) + 1)))

#COND_ELIM_CONV "!f n. (\m. f ((m = 0) => 0 | (m - 1))) (SUC n) < (f n) + 1";;
|- (!f n. ((\m. f((m = 0) => 0 | m - 1))(SUC n)) < ((f n) + 1)) =
   (!f n. ((\m. ((m = 0) => f 0 | f(m - 1)))(SUC n)) < ((f n) + 1))
```

Useful as a preprocessor to decision procedures which do not allow conditional statements in their argument formula.

See also

Arith.SUB_AND_COND_ELIM_CONV

COND_REWR_CANON (Cond_rewrite)

```
Cond_rewrite.COND_REWR_CANON : thm -> thm
```

Transform a theorem into a form accepted by COND_REWR_TAC.

COND_REWR_CANON transforms a theorem into a form accepted by COND_REWR_TAC. The input theorem should be an implication of the following form

```
!x1 ... xn. P1[xi] ==> ... ==> !y1 ... ym. Pr[xi,yi] ==>
  (!z1 ... zk. u[xi,yi,zi] = v[xi,yi,zi])
```

where each antecedent `Pi` itself may be a conjunction or disjunction. The output theorem will have all universal quantifications moved to the outer most level with possible renaming to prevent variable capture, and have all antecedents which are a conjunction transformed to implications. The output theorem will be in the following form

```
!x1 ... xn y1 ... ym z1 ... zk.
 P11[xi] ==> ... ==> P1p[xi] ==> ... ==>
  Pr1[xi,yi] ==> ... ==> Prq[x1,yi] ==> (u[xi,yi,zi] = v[xi,yi,zi])
```

Failure

This function fails if the input theorem is not in the correct form.

Example

`COND_REWR_CANON` transforms the built-in theorem `CANCL_SUB` into the form for conditional rewriting:

```
#COND_REWR_CANON CANCEL_SUB;;
Theorem CANCEL_SUB autoloading from theory `arithmetic` ...
CANCEL_SUB = |- !p n m. p <= n /\ p <= m ==> ((n - p = m - p) = (n = m))

|- !p n m. p <= n ==> p <= m ==> ((n - p = m - p) = (n = m))
```

See also

`Cond_rewrite.COND_REWRITE1_TAC`, `Cond_rewrite.COND_REWR_TAC`, `Cond_rewrite.COND_REWRITE1_CONV`, `Cond_rewrite.COND_REWR_CONV`, `Cond_rewrite.search_top_down`

COND_REWR_CONV (Cond_rewrite)

```
Cond_rewrite.COND_REWR_CONV : ((term -> term ->
 ((term # term) list # (type # type) list) list) -> thm -> conv)
```

A lower level conversion implementing simple conditional rewriting.

COND_REWR_CONV is one of the basic building blocks for the implementation of the simple conditional rewriting conversions in the HOL system. In particular, the conditional term replacement or rewriting done by all the conditional rewriting conversions in this library is ultimately done by applications of COND_REWR_CONV. The description given here for COND_REWR_CONV may therefore be taken as a specification of the atomic action of replacing equals by equals in a term under certain conditions that are used in all these higher level conditional rewriting conversions.

The first argument to COND_REWR_CONV is expected to be a function which returns a list of matches. Each of these matches is in the form of the value returned by the built-in function match. It is used to search the input term for instances which may be rewritten.

The second argument to COND_REWR_CONV is expected to be an implicative theorem in the following form:

```
A |- !x1 ... xn. P1 ==> ... Pm ==> (Q[x1,...,xn] = R[x1,...,xn])
```

where x1, ..., xn are all the variables that occur free in the left hand side of the conclusion of the theorem but do not occur free in the assumptions.

The last argument to COND_REWR_CONV is the term to be rewritten.

If fn is a function and th is an implicative theorem of the kind shown above, then COND_REWR_CONV fn th will be a conversion. When applying to a term tm, it will return a theorem

```
P1', ..., Pm' |- tm = tm[R'/Q']
```

if evaluating fn Q[x1,...,xn] tm returns a non-empty list of matches. The assumptions of the resulting theorem are instances of the antecedents of the input theorem th. The right hand side of the equation is obtained by rewriting the input term tm with instances of the conclusion of the input theorem.

Failure

COND_REWR_CONV fn th fails if th is not an implication of the form described above. If th is such an equation, but the function fn returns a null list of matches, or the function fn returns a non-empty list of matches, but the term or type instantiation fails.

Example

The following example illustrates a straightforward use of COND_REWR_CONV. We use the built-in theorem LESS_MOD as the input theorem, and the function search_top_down as the search function.

```
#LESS_MOD;;
Theorem LESS_MOD autoloading from theory `arithmetic` ...
LESS_MOD = |- !n k. k < n ==> (k MOD n = k)

|- !n k. k < n ==> (k MOD n = k)

#search_top_down;;
- : (term -> term -> ((term # term) list # (type # type) list) list)

#COND_REWR_CONV search_top_down LESS_MOD "2 MOD 3";;
2 < 3 |- 2 MOD 3 = 2
```

See also

Cond_rewrite.COND_REWR_TAC, Cond_rewrite.COND_REWRITE1_TAC, Cond_rewrite.COND_REWRITE1_CONV, Cond_rewrite.COND_REWR_CANON, Cond_rewrite.search_top_down

COND_REWR_TAC (Cond_rewrite)

```
Cond_rewrite.COND_REWR_TAC :
 (term -> term -> ((term * term) list * (type * type) list) list) ->
 thm_tactic
```

A lower level tactic used to implement simple conditional rewriting tactic.

COND_REWR_TAC is one of the basic building blocks for the implementation of conditional rewriting in the HOL system. In particular, the conditional term replacement or rewriting done by all the built-in conditional rewriting tactics is ultimately done by applications of COND_REWR_TAC. The description given here for COND_REWR_TAC may therefore be taken as a specification of the atomic action of replacing equals by equals in the goal under certain conditions that aare used in all these higher level conditional rewriting tactics.

The first argument to COND_REWR_TAC is expected to be a function which returns a list of matches. Each of these matches is in the form of the value returned by the built-in function match. It is used to search the goal for instances which may be rewritten.

The second argument to COND_REWR_TAC is expected to be an implicative theorem in the following form:

```
A |- !x1 ... xn. P1 ==> ... Pm ==> (Q[x1,...,xn] = R[x1,...,xn])
```

where x1, ..., xn are all the variables that occur free in the left-hand side of the conclusion of the theorem but do not occur free in the assumptions.

If fn is a function and th is an implicative theorem of the kind shown above, then COND_REWR_TAC fn th will be a tactic which returns a list of subgoals if evaluating

```
fn Q[x1,...,xn] gl
```

returns a non-empty list of matches when applied to a goal (asm,gl).

Let ml be the match list returned by evaluating fn Q[x1,...,xn] gl. Each element in this list is in the form of

```
([(e1,x1);...;(ep,xp)], [(ty1,vty1);...;(tyq,vtyq)])
```

which specifies the term and type instantiations of the input theorem th. Either the term pair list or the type pair list may be empty. In the case that both lists are empty, an exact match is found, i.e., no instantiation is required. If ml is an empty list, no match has been found and the tactic will fail.

For each match in ml, COND_REWR_TAC will perform the following: 1) instantiate the input theorem th to get

```
th' = A |- P1' ==> ... ==> Pm' ==> (Q' = R')
```

where the primed subterms are instances of the corresponding unprimed subterms obtained by applying INST_TYPE with [(ty1,vty1);...;(tyq,vtyq)] and then INST with [(e1,x1);...;(ep,xp)]; 2) search the assumption list asm for occurrences of any antecedents P1', ..., Pm'; 3) if all antecedents appear in asm, the goal gl is reduced to gl' by substituting R' for each free occurrence of Q', otherwise, in addition to the substitution, all antecedents which do not appear in asm are added to it and new subgoals corresponding to these antecedents are created. For example, if Pk', ..., Pm' do not appear in asm, the following subgoals are returned:

```
asm ?- Pk'  ...  asm ?- Pm'   {{asm,Pk',...,Pm'}} ?- gl'
```

If COND_REWR_TAC is given a theorem th:

```
A |- !x1 ... xn y1 ... yk. P1 ==> ... ==> Pm ==> (Q = R)
```

where the variables y1, ..., ym do not occur free in the left-hand side of the conclusion Q but they do occur free in the antecedents, then, when carrying out Step 2 described above, COND_REWR_TAC will attempt to find instantiations for these variables from the assumption asm. For example, if x1 and y1 occur free in P1, and a match is found in which e1 is an instantiation of x1, then P1' will become P1[e1/x1, y1]. If a term P1'' = P1[e1,e1'/x1,y1] appears in asm, th' is instantiated with (e1', y1) to get

```
th'' = A |- P1'' ==> ... ==> Pm'' ==> (Q' = R'')
```

then R'' is substituted into gl for all free occurrences of Q'. If no consistent instantiation is found, then P1' which contains the uninstantiated variable y1 will become one of the new subgoals. In such a case, the user has no control over the choice of the variable yi.

Failure

COND_REWR_TAC fn th fails if th is not an implication of the form described above. If th is such an equation, but the function fn returns a null list of matches, or the function fn returns a non-empty list of matches, but the term or type instantiation fails.

Example

The following example illustrates a straightforward use of COND_REWR_TAC. We use the built-in theorem LESS_MOD as the input theorem, and the function search_top_down as the search function.

```
#LESS_MOD;;
Theorem LESS_MOD autoloading from theory `arithmetic` ...
LESS_MOD = |- !n k. k < n ==> (k MOD n = k)

|- !n k. k < n ==> (k MOD n = k)

#search_top_down;;
- : (term -> term -> ((term # term) list # (type # type) list) list)
```

We set up a goal

```
#g"2 MOD 3 = 2";;
"2 MOD 3 = 2"

() : void
```

and then apply the tactic

```
#e(COND_REWR_TAC search_top_down LESS_MOD);;
OK..
2 subgoals
"2 = 2"
    [ "2 < 3" ]

"2 < 3"

 () : void
```

See also

Cond_rewrite.COND_REWRITE1_TAC, Cond_rewrite.COND_REWRITE1_CONV, Cond_rewrite.COND_REWR_CONV, Cond_rewrite.COND_REWR_CANON, Cond_rewrite.search_top_down

COND_REWRITE1_CONV (Cond_rewrite)

```
Cond_rewrite.COND_REWRITE1_CONV : thm list -> thm -> conv
```

A simple conditional rewriting conversion.

COND_REWRITE1_CONV is a front end of the conditional rewriting conversion COND_REWR_CONV. The input theorem should be in the following form

```
A |- !x11 ... . P1 ==> ... !xm1 ... . Pm ==> (!x ... . Q = R)
```

where each antecedent Pi itself may be a conjunction or disjunction. This theorem is transformed to a standard form expected by COND_REWR_CONV which carries out the actual

rewriting. The transformation is performed by COND_REWR_CANON. The search function passed to COND_REWR_CONV is search_top_down. The effect of applying the conversion COND_REWRITE1_CONV ths th to a term tm is to derive a theorem

```
A' |- tm = tm[R'/Q']
```

where the right hand side of the equation is obtained by rewriting the input term tm with an instance of the conclusion of the input theorem. The theorems in the list ths are used to discharge the assumptions generated from the antecedents of the input theorem.

Failure

COND_REWRITE1_CONV ths th fails if th cannot be transformed into the required form by COND_REWR_CANON. Otherwise, it fails if no match is found or the theorem cannot be instantiated.

Example

The following example illustrates a straightforward use of COND_REWRITE1_CONV. We use the built-in theorem LESS_MOD as the input theorem.

```
#LESS_MOD;;
Theorem LESS_MOD autoloading from theory `arithmetic` ...
LESS_MOD = |- !n k. k < n ==> (k MOD n = k)

|- !n k. k < n ==> (k MOD n = k)

#COND_REWRITE1_CONV [] LESS_MOD "2 MOD 3";;
2 < 3 |- 2 MOD 3 = 2

#let less_2_3 = REWRITE_RULE[LESS_MONO_EQ;LESS_0]
#(REDEPTH_CONV num_CONV "2 < 3");;
less_2_3 = |- 2 < 3

#COND_REWRITE1_CONV [less_2_3] LESS_MOD "2 MOD 3";;
|- 2 MOD 3 = 2
```

In the first example, an empty theorem list is supplied to COND_REWRITE1_CONV so the resulting theorem has an assumption 2 < 3. In the second example, a list containing a theorem |- 2 < 3 is supplied, the resulting theorem has no assumptions.

See also

Cond_rewrite.COND_REWR_TAC, Cond_rewrite.COND_REWRITE1_TAC, Cond_rewrite.COND_REWR_CONV, Cond_rewrite.COND_REWR_CANON, Cond_rewrite.search_top_down

COND_REWRITE1_TAC (Cond_rewrite)

Cond_rewrite.COND_REWRITE1_TAC : thm_tactic

A simple conditional rewriting tactic.

COND_REWRITE1_TAC is a front end of the conditional rewriting tactic COND_REWR_TAC. The input theorem should be in the following form

```
A |- !x11 ... . P1 ==> ... !xm1 ... . Pm ==> (!x ... . Q = R)
```

where each antecedent Pi itself may be a conjunction or disjunction. This theorem is transformed to a standard form expected by COND_REWR_TAC which carries out the actual rewriting. The transformation is performed by COND_REWR_CANON. The search function passed to COND_REWR_TAC is search_top_down. The effect of applying this tactic is to substitute into the goal instances of the right hand side of the conclusion of the input theorem Ri' for the corresponding instances of the left hand side. The search is top-down left-to-right. All matches found by the search function are substituted. New subgoals corresponding to the instances of the antecedents which do not appear in the assumption of the original goal are created. See manual page of COND_REWR_TAC for details of how the instantiation and substitution are done.

Failure

COND_REWRITE1_TAC th fails if th cannot be transformed into the required form by the function COND_REWR_CANON. Otherwise, it fails if no match is found or the theorem cannot be instantiated.

Example

The following example illustrates a straightforward use of COND_REWRITE1_TAC. We use the built-in theorem LESS_MOD as the input theorem.

```
#LESS_MOD;;
Theorem LESS_MOD autoloading from theory `arithmetic` ...
LESS_MOD = |- !n k. k < n ==> (k MOD n = k)

|- !n k. k < n ==> (k MOD n = k)
```

We set up a goal

```
#g"2 MOD 3 = 2";;
"2 MOD 3 = 2"

() : void
```

and then apply the tactic

```
#e(COND_REWRITE1_TAC LESS_MOD);;
OK..
2 subgoals
"2 = 2"
    [ "2 < 3" ]

"2 < 3"

() : void
```

See also

`Cond_rewrite.COND_REWR_TAC`, `Cond_rewrite.COND_REWRITE1_CONV`, `Cond_rewrite.COND_REWR_CONV`, `Cond_rewrite.COND_REWR_CANON`, `Cond_rewrite.search_top_down`

conditional (boolSyntax)

`boolSyntax.conditional : term`

Constant denoting conditional expressions.

The ML variable `boolSyntax.conditional` is bound to the term `bool$COND`.

See also

boolSyntax.equality, boolSyntax.implication, boolSyntax.select, boolSyntax.T, boolSyntax.F, boolSyntax.universal, boolSyntax.existential, boolSyntax.exists1, boolSyntax.conjunction, boolSyntax.disjunction, boolSyntax.bool_case, boolSyntax.let_tm, boolSyntax.arb

Cong (simpLib)

```
simpLib.Cong : thm -> thm
```

Marks a theorem as a congruence rule for the simplifier.

The Cong function marks (or "tags") a theorem so that when passed to the simplifier, it is not used as a rewrite, but rather as a congruence rule. This is a simpler way of adding a congruence rule to the simplifier than using the underlying SSFRAG function.

Failure

Never fails. On the other hand, Cong does not check that the theorem passed as an argument is a valid congruence rule, and invalid congruence rules may have unpredictable effects on the behaviour of the simplifier.

Example

```
- SIMP_CONV pure_ss [] ``!x::P. x IN P /\ Q x``;
<<HOL message: inventing new type variable names: 'a>>
! Uncaught exception:
! UNCHANGED
- RES_FORALL_CONG;
> val it =
    |- (P = Q) ==>
       (!x. x IN Q ==> (f x = g x)) ==>
       (RES_FORALL P f = RES_FORALL Q g) : thm
- SIMP_CONV pure_ss [Cong RES_FORALL_CONG] ``!x::P. x IN P ``;
<<HOL message: inventing new type variable names: 'a>>
> val it = |- (!x::P. x IN P /\ Q x) = !x::P. T /\ Q x : thm
```

(Note that RES_FORALL_CONG is already included in bool_ss and all simpsets built on it.)

See also

simpLib.SSFRAG

CONG_TAC (bossLib)

```
bossLib.CONG_TAC : int option -> tactic
```

Applies congruence rules backwards repeatedly to attack an equality

Applying CONG_TAC dopt to a goal ?- x = y attempts to apply congruence rules backwards repeatedly, generating a number of further equality subgoals. The dopt parameter limits the number of times this will be done: when dopt is NONE, there is no limit: the base transformation will be tried on the initial and all subsequent goals. If dopt is SOME i, then i is the total number of transformations that will be made. (If dopt is SOME 0 then CONG_TAC dopt is equivalent to ALL_TAC.)

If at least one of the equality's arguments is an abstraction (possibly paired), then the transformation rewrites with function extensionality, strips the universally quantified variables, and beta-reduces where necessary.

If at least one of the equality's arguments is a set comprehension, then the transformation rewrites with set-extensionality and applies the conversion that calculates what it is for a term to be a member of a comprehension to one or both sides of the equality. Unless both sides are set comprehensions, this is likely to be the last transformation possible.

If the goal matches the conclusion of any of the theorems stored as congruences for the definition package (with attribute name defncong or cong), then this theorem is applied backwards to generate new sub-goals. If the new sub-goals include preconditions and universally quantified variables, these are stripped into the assumptions.

Finally, the "base transformation" depends on the shape of the equality's arguments. If both sides are combinations (M e1 and N e2, say), then the base transformation will be similar to an application of MK_COMB_TAC, generating at least the goals ?- M = N and ?- e1 = e2. When the head terms of both applications are equal, then one step of the base transformation is taken to be the iteration of MK_COMB_TAC that strips all arguments, so that ?- f e1 .. en = f e1' .. en' will turn into n subgoals, each of the form ?- ei = ei'. (The ?- f = f subgoal will be eliminated immediately, as below.)

In all cases, new subgoals that are instances of reflexivity, or which occur in the goal's assumptions (with either orientation) are immediately eliminated.

Failure

Fails if the provided depth is either `NONE` or `SOME i` with `i` greater than zero, and the goal is not an equality at all, or cannot be changed by any of the transformations described above. For example, with `f : 'a -> num` and `g : num -> num`, the goal `?- f a = g n` cannot be reduced.

Example

The following involves a handling of abstractions:

```
> CONG_TAC NONE ([], “(∀x. f (z:'a) < x) ⇔ (∀y. c < y)”);
val it = ([([], “f z = c”)], fn): goal list * validation
```

Slightly altering the goal, and keeping the depth as `NONE` turns something true into something unprovable:

```
> CONG_TAC NONE ([], “(∀x. f (z:'a) < x) ⇔ (∀y. y < 6)”);
val it = ([([], “f z = x”), ([], “x = 6”)], fn): goal list * validation
```

where the `x` in each sub-goal is completely fresh.

Finally, user-congruences can give richer contexts when proving functions equal:

```
> CONG_TAC NONE
    ([], “MAP (λa. f a + 1) (l1:'a list) = MAP g (l2:'a list)”);
val it = ([([“MEM x l2”], “f x + 1 = g x”), ([], “l1 = l2”)], fn):
   goal list * validation
```

Comments

This is a powerful tool for taking apart two terms that share a skeleton and need only have their leaves shown to be equal. Equally, it is quite possible for this tactic to turn a solvable goal into an unsolvable one.

An application of `CONG_TAC` will never break apart the function applications that lie within the representation of natural number numerals.

The name `cong_tac` can be used as an alias for `CONG_TAC`.

See also

`Tactic.AP_TERM_TAC`, `Tactic.AP_THM_TAC`, `Tactic.MK_COMB_TAC`

CONJ (Thm)

```
Thm.CONJ : thm -> thm -> thm
```

Introduces a conjunction.

```
    A1 |- t1      A2 |- t2
   ------------------------  CONJ
     A1 u A2 |- t1 /\ t2
```

Failure

Never fails.

Comments

The theorem `AND_INTRO_THM` can be instantiated to similar effect.

See also

`Drule.BODY_CONJUNCTS`, `Thm.CONJUNCT1`, `Thm.CONJUNCT2`, `Drule.CONJ_PAIR`, `Drule.LIST_CONJ`, `Drule.CONJ_LIST`, `Drule.CONJUNCTS`

CONJ_ASM1_TAC (Tactic)

```
Tactic.CONJ_ASM1_TAC : tactic
```

Reduces a conjunctive goal to two subgoals: prove the first conjunct, then the second assuming the first.

When applied to a goal `A ?- t1 /\ t2`, the tactic `CONJ_ASM1_TAC` reduces it to two subgoals corresponding to the first conjunct then the second conjunct.

```
       A ?- t1 /\ t2
==========================  CONJ_ASM1_TAC
 A ?- t1   A u {t1} ?- t2
```

Failure

Fails unless the conclusion of the goal is a conjunction.

See also

`Tactic.CONJ_ASM2_TAC`, `Tactic.CONJ_TAC`, `Tactical.USE_SG_THEN`

CONJ_ASM2_TAC (Tactic)

```
Tactic.CONJ_ASM2_TAC : tactic
```

Reduces a conjunctive goal to two subgoals: prove the first conjunct assuming the second, then prove the second conjunct.

When applied to a goal `A ?- t1 /\ t2`, the tactic `CONJ_ASM2_TAC` reduces it to two subgoals corresponding to the two conjuncts, assuming the first to prove the second.

```
       A ?- t1 /\ t2
==========================  CONJ_ASM2_TAC
 A u {t2} ?- t1   A ?- t2
```

Failure

Fails unless the conclusion of the goal is a conjunction.

See also

`Tactic.CONJ_ASM1_TAC`, `Tactic.CONJ_TAC`, `Tactical.USE_SG_THEN`

CONJ_DISCH (Drule)

```
Drule.CONJ_DISCH : (term -> thm -> thm)
```

Discharges an assumption and conjoins it to both sides of an equation.

Given an term `t` and a theorem `A |- t1 = t2`, which is an equation between boolean terms, `CONJ_DISCH` returns `A - {t} |- (t /\ t1) = (t /\ t2)`, i.e. conjoins `t` to both sides of the equation, removing `t` from the assumptions if it was there.

```
          A |- t1 = t2
------------------------------  CONJ_DISCH "t"
 A - {t} |- t /\ t1 = t /\ t2
```

Failure

Fails unless the theorem is an equation, both sides of which, and the term provided are of type `bool`.

See also

`Drule.CONJ_DISCHL`

CONJ_DISCHL (Drule)

```
Drule.CONJ_DISCHL : (term list -> thm -> thm)
```

Conjoins multiple assumptions to both sides of an equation.

Given a term list `[t1;...;tn]` and a theorem whose conclusion is an equation between boolean terms, `CONJ_DISCHL` conjoins all the terms in the list to both sides of the equation, and removes any of the terms which were in the assumption list.

```
                        A |- s = t
  ------------------------------------------------------  CONJ_DISCHL
   A - {t1,...,tn} |- (t1/\.../\tn/\s) = (t1/\.../\tn/\t)     [t1,...,tn]
```

Failure

Fails unless the theorem is an equation, both sides of which, and all the terms provided, are of type `bool`.

See also

`Drule.CONJ_DISCH`

CONJ_FORALL_CONV (unwindLib)

```
unwindLib.CONJ_FORALL_CONV : conv
```

Moves universal quantifiers up through a tree of conjunctions.

`CONJ_FORALL_CONV "(!x1 ... xm. t1) /\ ... /\ (!x1 ... xm. tn)"` returns the following theorem:

```
|- (!x1 ... xm. t1) /\ ... /\ (!x1 ... xm. tn) =
   !x1 ... xm. t1 /\ ... /\ tn
```

where the original term can be an arbitrary tree of conjunctions. The structure of the tree is retained in both sides of the equation.

Failure

Never fails.

Example

```
#CONJ_FORALL_CONV "((!(x:*) (y:*) (z:*). a) /\ (!(x:*) (y:*) (z:*). b)) /\
#                  (!(x:*) (y:*) (z:*). c)";;
```

```
|- ((!x y z. a) /\ (!x y z. b)) /\ (!x y z. c) = (!x y z. (a /\ b) /\ c)

#CONJ_FORALL_CONV "T";;
|- T = T

#CONJ_FORALL_CONV "((!(x:*) (y:*) (z:*). a) /\ (!(x:*) (w:*) (z:*). b)) /\
#                  (!(x:*) (y:*) (z:*). c)";;
|- ((!x y z. a) /\ (!x w z. b)) /\ (!x y z. c) =
   (!x. ((!y z. a) /\ (!w z. b)) /\ (!y z. c))
```

See also

`unwindLib.FORALL_CONJ_CONV`, `unwindLib.CONJ_FORALL_ONCE_CONV`, `unwindLib.FORALL_CONJ_ONCE_CONV`, `unwindLib.CONJ_FORALL_RIGHT_RULE`, `unwindLib.FORALL_CONJ_RIGHT_RULE`

CONJ_FORALL_ONCE_CONV (unwindLib)

```
unwindLib.CONJ_FORALL_ONCE_CONV : conv
```

Moves a single universal quantifier up through a tree of conjunctions.

`CONJ_FORALL_ONCE_CONV "(!x. t1) /\ ... /\ (!x. tn)"` returns the theorem:

```
   |- (!x. t1) /\ ... /\ (!x. tn) = !x. t1 /\ ... /\ tn
```

where the original term can be an arbitrary tree of conjunctions. The structure of the tree is retained in both sides of the equation.

Failure

Fails if the argument term is not of the required form. The term need not be a conjunction, but if it is every conjunct must be universally quantified with the same variable.

Example

```
#CONJ_FORALL_ONCE_CONV "((!x. x \/ a) /\ (!x. x \/ b)) /\ (!x. x \/ c)";;
|- ((!x. x \/ a) /\ (!x. x \/ b)) /\ (!x. x \/ c) =
   (!x. ((x \/ a) /\ (x \/ b)) /\ (x \/ c))

#CONJ_FORALL_ONCE_CONV "!x. x \/ a";;
|- (!x. x \/ a) = (!x. x \/ a)

#CONJ_FORALL_ONCE_CONV "((!x. x \/ a) /\ (!y. y \/ b)) /\ (!x. x \/ c)";;
evaluation failed     CONJ_FORALL_ONCE_CONV
```

See also

unwindLib.FORALL_CONJ_ONCE_CONV, unwindLib.CONJ_FORALL_CONV, unwindLib.FORALL_CONJ_CONV, unwindLib.CONJ_FORALL_RIGHT_RULE, unwindLib.FORALL_CONJ_RIGHT_RULE

CONJ_FORALL_RIGHT_RULE (unwindLib)

```
unwindLib.CONJ_FORALL_RIGHT_RULE : (thm -> thm)
```

Moves universal quantifiers up through a tree of conjunctions.

```
    A |- !z1 ... zr.
          t = ?y1 ... yp. (!x1 ... xm. t1) /\ ... /\ (!x1 ... xm. tn)
   ------------------------------------------------------------------
     A |- !z1 ... zr. t = ?y1 ... yp. !x1 ... xm. t1 /\ ... /\ tn
```

Failure

Fails if the argument theorem is not of the required form, though either or both of `r` and `p` may be zero.

See also

unwindLib.FORALL_CONJ_RIGHT_RULE, unwindLib.CONJ_FORALL_CONV, unwindLib.FORALL_-CONJ_CONV, unwindLib.CONJ_FORALL_ONCE_CONV, unwindLib.FORALL_CONJ_ONCE_CONV

CONJ_LIST (Drule)

```
Drule.CONJ_LIST : (int -> thm -> thm list)
```

Extracts a list of conjuncts from a theorem (non-flattening version).

CONJ_LIST is the proper inverse of LIST_CONJ. Unlike CONJUNCTS which recursively splits as many conjunctions as possible both to the left and to the right, CONJ_LIST splits the top-level conjunction and then splits (recursively) only the right conjunct. The integer argument is required because the term tn may itself be a conjunction. A list of n theorems is returned.

```
 A |- t1 /\ (t2 /\ ( ... /\ tn)...)
-----------------------------------  CONJ_LIST n (A |- t1 /\ ... /\ tn)
 A |- t1   A |- t2   ...   A |- tn
```

Failure

Fails if the integer argument (n) is less than one, or if the input theorem has less than n conjuncts.

Example

Suppose the identifier th is bound to the theorem:

```
A |- (x /\ y) /\ z /\ w
```

Here are some applications of CONJ_LIST to th:

```
- CONJ_LIST 0 th;
! Uncaught exception:
```

```
! HOL_ERR

- CONJ_LIST 1 th;
> val it = [[A] |- (x /\ y) /\ z /\ w] : thm list

- CONJ_LIST 2 th;
> val it = [ [A] |- x /\ y,  [A] |- z /\ w] : thm list

- CONJ_LIST 3 th;
> val it = [ [A] |- x /\ y,  [A] |- z,  [A] |- w] : thm list

- CONJ_LIST 4 th;
! Uncaught exception:
! HOL_ERR
```

See also

Drule.BODY_CONJUNCTS, Drule.LIST_CONJ, Drule.CONJUNCTS, Thm.CONJ, Thm.CONJUNCT1, Thm.CONJUNCT2, Drule.CONJ_PAIR

CONJ_PAIR (Drule)

```
Drule.CONJ_PAIR : thm -> thm * thm
```

Extracts both conjuncts of a conjunction.

```
     A |- t1 /\ t2
----------------------  CONJ_PAIR
 A |- t1       A |- t2
```

The two resultant theorems are returned as a pair.

Failure

Fails if the input theorem is not a conjunction.

See also

`Drule.BODY_CONJUNCTS`, `Thm.CONJUNCT1`, `Thm.CONJUNCT2`, `Thm.CONJ`, `Drule.LIST_CONJ`, `Drule.CONJ_LIST`, `Drule.CONJUNCTS`

CONJ_TAC (Tactic)

```
Tactic.CONJ_TAC : tactic
```

Reduces a conjunctive goal to two separate subgoals.

When applied to a goal `A ?- t1 /\ t2`, the tactic `CONJ_TAC` reduces it to the two subgoals corresponding to each conjunct separately.

```
     A ?- t1 /\ t2
====================  CONJ_TAC
 A ?- t1      A ?- t2
```

Failure

Fails unless the conclusion of the goal is a conjunction.

See also

`Tactic.STRIP_TAC`

CONJ_VALIDATE (Tactical)

```
Tactical.CONJ_VALIDATE : tactic -> tactic
```

Adds requirement to prove assumptions that would make a tactic invalid.

A call to `CONJ_VALIDATE tac g` behaves the same as `tac g` if this tactic application is valid (meaning that when the tactic's validation function is applied to the resulting sub-goals, the generated theorem has no more assumptions than the original goal).

If the tactic's application is not valid because the resulting theorem has the wrong conclusion, CONJ_VALIDATE tac g will fail. If the resulting theorem has hypotheses not present in the goal state, then these hypotheses are added to the new goals as new proof obligations. The way in which this addition is done depends on how many subgoals tac generated when applied to g.

If tac completely solves g (though in an invalid way), the excess hypotheses h1...hn are bundled into a big conjunction and this becomes the one remaining goal. If tac g results in more than one new goal, then CONJ_VALIDATE tac g will produce one extra goal, consisting of a conjunction of hypotheses, as above. Finally, if tac g produces exactly one new goal to prove, and if the new goal's assumptions are a subset of the original goal's, CONJ_VALIDATE tac g will do the same, but with the conjunction of extra hypotheses to prove conjoined to that one goal's conclusion. If there is one new goal, but its assumptions are not a subset, then a separate sub-goal is generated.

Failure

A call CONJ_VALIDATE tac g will fail if tac g fails, or if tac g's validation function produces a theorem whose conclusion is not the same as g's conclusion.

Comments

CONJ_VALIDATE performs the same role as VALIDATE but tries to preserve the property of having a single successor goal in its argument tactics; when it does generate extra sub-goals, it also generates only one such.

Because CONJ_VALIDATE prefers to generate just one sub-goal, the result may be a state where one ends up having to prove a hypothesis in a weaker context (with fewer assumptions) to hand than if VALIDATE had been used.

See also

Tactical.VALIDATE

CONJUNCT1 (Thm)

```
Thm.CONJUNCT1 : thm -> thm
```

Extracts left conjunct of theorem.

```
 A |- t1 /\ t2
---------------  CONJUNCT1
    A |- t1
```

Failure

Fails unless the input theorem is a conjunction.

Comments

The theorem `AND1_THM` can be instantiated to similar effect.

See also

`Drule.BODY_CONJUNCTS`, `Thm.CONJUNCT2`, `Drule.CONJ_PAIR`, `Thm.CONJ`, `Drule.LIST_CONJ`, `Drule.CONJ_LIST`, `Drule.CONJUNCTS`

CONJUNCT2 (Thm)

```
Thm.CONJUNCT2 : thm -> thm
```

Extracts right conjunct of theorem.

```
 A |- t1 /\ t2
---------------  CONJUNCT2
    A |- t2
```

Failure

Fails unless the input theorem is a conjunction.

Comments

The theorem `AND2_THM` can be instantiated to similar effect.

See also

Drule.BODY_CONJUNCTS, Thm.CONJUNCT1, Drule.CONJ_PAIR, Thm.CONJ, Drule.LIST_CONJ, Drule.CONJ_LIST, Drule.CONJUNCTS

conjunction (boolSyntax)

boolSyntax.conjunction : term

Constant denoting logical conjunction.

The ML variable boolSyntax.conjunction is bound to the term bool$/\.

See also

boolSyntax.equality, boolSyntax.implication, boolSyntax.select, boolSyntax.T, boolSyntax.F, boolSyntax.universal, boolSyntax.existential, boolSyntax.exists1, boolSyntax.disjunction, boolSyntax.negation, boolSyntax.conditional, boolSyntax.bool_case, boolSyntax.let_tm, boolSyntax.arb

CONJUNCTS (Drule)

Drule.CONJUNCTS : (thm -> thm list)

Recursively splits conjunctions into a list of conjuncts.

Flattens out all conjuncts, regardless of grouping. Returns a singleton list if the input theorem is not a conjunction.

```
    A |- t1 /\ t2 /\ ... /\ tn
-----------------------------------  CONJUNCTS
 A |- t1   A |- t2   ...   A |- tn
```

Failure

Never fails.

Example

Suppose the identifier `th` is bound to the theorem:

```
A |- (x /\ y) /\ z /\ w
```

Application of `CONJUNCTS` to `th` returns the following list of theorems:

```
[A |- x, A |- y, A |- z, A |- w] : thm list
```

See also

`Drule.BODY_CONJUNCTS`, `Drule.CONJ_LIST`, `Drule.LIST_CONJ`, `Thm.CONJ`, `Thm.CONJUNCT1`, `Thm.CONJUNCT2`, `Drule.CONJ_PAIR`

CONJUNCTS_AC (Drule)

```
Drule.CONJUNCTS_AC : term * term -> thm
```

Prove equivalence under idempotence, symmetry and associativity of conjunction.

`CONJUNCTS_AC` takes a pair of terms `(t1, t2)` and proves `|- t1 = t2` if `t1` and `t2` are equivalent up to idempotence, symmetry and associativity of conjunction. That is, if `t1` and `t2` are two (different) arbitrarily-nested conjunctions of the same set of terms, then `CONJUNCTS_AC (t1,t2)` returns `|- t1 = t2`. Otherwise, it fails.

Failure

Fails if `t1` and `t2` are not equivalent, as described above.

Example

```
> CONJUNCTS_AC (Term `(P /\ Q) /\ R`, Term `R /\ (Q /\ R) /\ P`);
val it = ⊢ (P ∧ Q) ∧ R ⇔ R ∧ (Q ∧ R) ∧ P: thm
```

Used to reorder a conjunction. First sort the conjuncts in a term `t1` into the desired order (e.g., lexicographic order, for normalization) to get a new term `t2`, then call `CONJUNCTS_AC(t1,t2)`.

See also

`Drule.DISJUNCTS_AC`

CONJUNCTS_THEN (Thm_cont)

```
Thm_cont.CONJUNCTS_THEN : thm_tactical
```

Applies a theorem-tactic to each conjunct of a theorem.

`CONJUNCTS_THEN` takes a theorem-tactic `f`, and a theorem `t` whose conclusion must be a conjunction. `CONJUNCTS_THEN` breaks `t` into two new theorems, `t1` and `t2` which are `CONJUNCT1` and `CONJUNCT2` of `t` respectively, and then returns a new tactic: `f t1 THEN f t2`. That is,

```
CONJUNCTS_THEN f (A |- l /\ r) =  f (A |- l) THEN f (A |- r)
```

so if

```
 A1 ?- t1                    A2 ?- t2
==========  f (A |- l)      ==========  f (A |- r)
 A2 ?- t2                    A3 ?- t3
```

then

```
 A1 ?- t1
==========  CONJUNCTS_THEN f (A |- l /\ r)
 A3 ?- t3
```

Failure

`CONJUNCTS_THEN f` will fail if applied to a theorem whose conclusion is not a conjunction.

Comments

`CONJUNCTS_THEN f (A |- u1 /\ ... /\ un)` results in the tactic:

```
f (A |- u1) THEN f (A |- u2 /\ ... /\ un)
```

Unfortunately, it is more likely that the user had wanted the tactic:

```
f (A |- u1) THEN ... THEN f(A |- un)
```

Such a tactic could be defined as follows:

```
fun CONJUNCTS_THENL (f:thm_tactic) thm =
  List.foldl (op THEN) ALL_TAC (map f (CONJUNCTS thm));
```

or by using `REPEAT_TCL`.

See also

`Thm.CONJUNCT1`, `Thm.CONJUNCT2`, `Drule.CONJUNCTS`, `Tactic.CONJ_TAC`, `Thm_cont.CONJUNCTS_THEN2`, `Thm_cont.STRIP_THM_THEN`

CONJUNCTS_THEN2 (Thm_cont)

```
Thm_cont.CONJUNCTS_THEN2 : (thm_tactic -> thm_tactical)
```

Applies two theorem-tactics to the corresponding conjuncts of a theorem.

`CONJUNCTS_THEN2` takes two theorem-tactics, `f1` and `f2`, and a theorem `t` whose conclusion must be a conjunction. `CONJUNCTS_THEN2` breaks `t` into two new theorems, `t1` and `t2` which are `CONJUNCT1` and `CONJUNCT2` of `t` respectively, and then returns the tactic `f1 t1 THEN f2 t2`. Thus

```
CONJUNCTS_THEN2 f1 f2 (A |- l /\ r) =  f1 (A |- l) THEN f2 (A |- r)
```

so if

```
 A1 ?- t1                    A2 ?- t2
==========  f1 (A |- l)     ==========  f2 (A |- r)
 A2 ?- t2                    A3 ?- t3
```

then

```
 A1 ?- t1
==========  CONJUNCTS_THEN2 f1 f2 (A |- l /\ r)
 A3 ?- t3
```

Failure

`CONJUNCTS_THEN f` will fail if applied to a theorem whose conclusion is not a conjunction.

Comments

The system shows the type as `(thm_tactic -> thm_tactical)`.

The construction of complex `tacticals` like `CONJUNCTS_THEN`.

See also

`Thm.CONJUNCT1`, `Thm.CONJUNCT2`, `Drule.CONJUNCTS`, `Tactic.CONJ_TAC`, `Thm_cont.CONJUNCTS_THEN2`, `Thm_cont.STRIP_THM_THEN`

cons (Lib)

```
Lib.cons : 'a -> 'a list -> 'a list
```

Curried form of list cons operation.

In some programming situations it is handy to use the “cons” operation in a curried form. Although it is easy to code up on demand, the `cons` function is provided for convenience.

Failure

Never fails.

Example

```
> map (cons 1) [[],[2],[2,3]];
val it = [[1], [1, 2], [1, 2, 3]]: int list list
```

conseq_conv (ConseqConv)

```
ConseqConv.type conseq_conv
```

A type for functions that given a term produce a theorem with an implication at the top level.

Classical conversions (see `Conv`) convert a given term `t` to a term `eqt` that is equal to `t`. For a boolean term `t`, it is however sometimes useful not to preserve equivalence, but to either strengthen `t` to `st` or to weaken it to `wt`. The type `conseq_conv` is used for ML functions that perform these operations. These ML Functions are called `consequence conversions` in the following.

Given a consequence conversion `CONSEQ_CONV` and a term `t`, then `CONSEQ_CONV` can either fail with an `HOL_ERR`-exception, raise an `UNCHANGED`-exception or produce a theorem of one of the following forms:

1. `st ==> t`
2. `t ==> wt`
3. `t = eqt`

Example

Examples of simple consequence conversion are `TRUE_CONSEQ_CONV` and `FALSE_CONSEQ_CONV`.

See also

`ConseqConv.directed_conseq_conv`, `ConseqConv.TRUE_FALSE_REFL_CONSEQ_CONV`

CONSEQ_CONV_direction (ConseqConv)

```
ConseqConv.type CONSEQ_CONV_direction
```

A type used to tell directed consequence conversions what the desired result should look like.

This type is used to instruct a directed consequence conversion how to behave. Given a direction `dir` and a boolean term `t` the result of a directed consequence conversion `DCONSEQ_CONV` should be of the form

```
st ==> t for dir = CONSEQ_CONV_STRENGTHEN_direction
t ==> wt for dir = CONSEQ_CONV_WEAKEN_direction
st ==> t, t ==> wt or t = eqt for dir = CONSEQ_CONV_UNKNOWN_direction
```

See also

`ConseqConv.directed_conseq_conv`, `ConseqConv.TRUE_FALSE_REFL_CONSEQ_CONV`

CONSEQ_CONV_TAC (ConseqConv)

```
ConseqConv.CONSEQ_CONV_TAC : directed_conseq_conv -> tactic
```

Reduces the goal using a consequence conversion.

`CONSEQ_CONV_TAC c` tries to strengthen a goal `P` using `c` to a new goal `P'`. It then remains to show that `P'` holds.

See also

`Tactic.MATCH_MP_TAC`

CONSEQ_REWRITE_CONV (ConseqConv)

```
ConseqConv.CONSEQ_REWRITE_CONV : (thm list * thm list * thm list) -> directed_conseq_con
```

Applies `CONSEQ_TOP_REWRITE_CONV` repeatedly at subterms.

This directed consequence conversion is a combination of `CONSEQ_TOP_REWRITE_CONV` and `DEPTH_CONSEQ_CONV`. Given lists of theorems, these theorems are preprocessed to extract implications. Then these implications are used to either weaken or strengthen an input term.

Example

Reconsider the example for `DEPTH_CONSEQ_CONV`. Let `rewrite_every_thm` be the following theorem:

```
val rewrite_every_thm =
    |- FEVERY P FEMPTY /\
       (FEVERY P f /\ P (x,y) ==> FEVERY P (f |+ (x,y)));
```

Then the following call of `CONSEQ_REWRITE_CONV`

```
CONSEQ_REWRITE_CONV ([], [rewrite_every_thm], []) CONSEQ_CONV_STRENGTHEN_direction
  ``!y2. FEVERY P (f |+ (x1, y1) |+ (x2,y2)) /\ Q z``
```

results in

```
 |- (!y2. ((FEVERY P f /\ P (x1, y1)) /\ P (x2,y2)) /\ Q z) ==>
    (!y2. FEVERY P (f |+ (x1, y1) |+ (x2,y2)) /\ Q z)
```

More examples can be found at the end of `ConseqConv.sml`.

See also

`Drule.MATCH_MP`, `ConseqConv.CONSEQ_TOP_REWRITE_CONV`, `ConseqConv.DEPTH_CONSEQ_CONV`, `ConseqConv.EXT_CONSEQ_REWRITE_CONV`

CONSEQ_TOP_REWRITE_CONV (ConseqConv)

```
ConseqConv.CONSEQ_TOP_REWRITE_CONV : (thm list * thm list * thm list) -> directed_c
```

An extended version of `MATCH_MP`.

This consequence conversion gets 3 lists of theorems as parameters: `both_thmL`, `strengthen_thmL` and `weaken_thmL`. The theorems in these lists are used to strengthen or weaken a given boolean term at toplevel. If using them for strengthening this consequence conversion behaves similar to MATCH_MP. As the names suggest, the theorems in `strengthen_thmL` are used for strengthening, the ones in `weaken_thmL` for weakening and the ones in `both_thmL` for both.

Before trying to apply the conversion, the theorem lists are preprocessed. The theorems are split along conjunctions and allquantification is removed. Then theorems with toplevel negation `|- ~P` are rewritten to `|- P = F`. Afterwards every theorem `|- P` that is not an implication or an boolean equation is replaced by `|- P = T`. Finally, boolean equations `|- P = Q` are splitted into two theorems `|- P ==> Q` and `|- Q ==> P`. One ends up with a list of implications.

Given a term `t` the conversion tries to find a theorem `|- P ==> Q` and - depending on to the direction - strengthen `t` by matching it with `Q` or weaken it by matching it with `P`.

Example

This directed consequence conversion is intended to be used together with `DEPTH_CONSEQ_CONV`. The combination of both is called `CONSEQ_REWRITE_CONV`. Please have a look there for an example.

See also

`Drule.MATCH_MP`, `ConseqConv.CONSEQ_REWRITE_CONV`, `ConseqConv.DEPTH_CONSEQ_CONV`

constants (Theory)

```
Theory.constants : string -> term list
```

Returns a list of the constants defined in a named theory.

The call

```
constants thy
```

where `thy` is an ancestor theory (the special string `"-"` means the current theory), returns a list of all the constants in that theory.

Failure

Fails if the named theory does not exist, or is not an ancestor of the current theory.

Example

```
> load "combinTheory";
val it = (): unit

> constants "combin";
val it =
   [“$o”, “W”, “UPDATE”, “S”, “RIGHT_ID”, “RESTRICTION”, “MONOID”, “LEFT_ID”,
    “K”, “I”, “FCOMM”, “FAIL”, “EXTENSIONAL”, “COMM”, “flip”, “ASSOC”, “$:>”]:
   term list
```

See also

`Theory.types`, `Theory.current_axioms`, `Theory.current_definitions`, `Theory.current_theorems`

CONTR (Drule)

```
Drule.CONTR : term -> thm -> thm
```

Implements the intuitionistic contradiction rule.

When applied to a term `t` and a theorem `A |- F`, the inference rule `CONTR` returns the theorem `A |- t`.

```
 A |- F
--------  CONTR t
 A |- t
```

Failure

Fails unless the term has type `bool` and the theorem has `F` as its conclusion.

See also

`Thm.CCONTR`, `Drule.CONTRAPOS`, `Tactic.CONTR_TAC`, `Thm.NOT_ELIM`

CONTR_TAC (Tactic)

```
Tactic.CONTR_TAC : thm_tactic
```

Solves any goal from contradictory theorem.

When applied to a contradictory theorem `A' |- F`, and a goal `A ?- t`, the tactic `CONTR_TAC` completely solves the goal. This is an invalid tactic unless `A'` is a subset of `A`.

```
 A ?- t
========  CONTR_TAC (A' |- F)
```

Failure

Fails unless the theorem is contradictory, i.e. has `F` as its conclusion.

See also

`Tactic.CHECK_ASSUME_TAC`, `Drule.CONTR`, `Thm.CCONTR`, `Drule.CONTRAPOS`, `Thm.NOT_ELIM`

CONTRAPOS (Drule)

```
Drule.CONTRAPOS : (thm -> thm)
```

Deduces the contrapositive of an implication.

When applied to a theorem `A |- s ==> t`, the inference rule `CONTRAPOS` returns its contrapositive, `A |- ~t ==> ~s`.

```
   A |- s ==> t
----------------  CONTRAPOS
 A |- ~t ==> ~s
```

Failure

Fails unless the theorem is an implication.

See also

`Thm.CCONTR`, `Drule.CONTR`, `Conv.CONTRAPOS_CONV`, `Thm.NOT_ELIM`

CONTRAPOS_CONV (Conv)

```
Conv.CONTRAPOS_CONV : conv
```

Proves the equivalence of an implication and its contrapositive.

When applied to an implication `P ==> Q`, the conversion `CONTRAPOS_CONV` returns the theorem:

```
|- (P ==> Q) = (~Q ==> ~P)
```

Failure

Fails if applied to a term that is not an implication.

See also

`Drule.CONTRAPOS`

conv (Abbrev)

```
Abbrev.conv = term -> thm
```

The type of a function which finds an equal term and returns a theorem stating the equality

If the function `c` is not only of the right type, but is actually what we call a conversion, then `(c : conv) (t : term)` returns a theorem of the form

```
|- t = t'
```

If (roughly speaking) `t` is of the right form to be passed to `c`, but `c` doesn't find a desired equal term `t'` then `c t` may raise the exception `UNCHANGED` in preference to returning `|- t = t`, as it can be dealt with more efficiently.

See also

`Conv.UNCHANGED`

CONV_RULE (Conv)

```
Conv.CONV_RULE : (conv -> thm -> thm)
```

Makes an inference rule from a conversion.

If `c` is a conversion, then `CONV_RULE c` is an inference rule that applies `c` to the conclusion of a theorem. That is, if `c` maps a term `"t"` to the theorem `|- t = t'`, then the rule `CONV_RULE c` infers `|- t'` from the theorem `|- t`. More precisely, if `c "t"` returns `A' |- t = t'`, then:

```
      A |- t
--------------  CONV_RULE c
 A u A' |- t'
```

Note that if the conversion `c` returns a theorem with assumptions, then the resulting inference rule adds these to the assumptions of the theorem it returns.

If `c` raises `UNCHANGED` then `CONV_RULE c th` returns `th`.

Failure

`CONV_RULE c th` fails if `c` fails (other than by raising `UNCHANGED`) when applied to the conclusion of `th`. The function returned by `CONV_RULE c` will also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

See also

`Abbrev.conv`, `Conv.UNCHANGED`, `Tactic.CONV_TAC`, `Conv.HYP_CONV_RULE`, `Conv.RIGHT_-CONV_RULE`

CONV_TAC (Tactic)

```
Tactic.CONV_TAC : (conv -> tactic)
```

Makes a tactic from a conversion.

If `c` is a conversion, then `CONV_TAC c` is a tactic that applies `c` to the goal. That is, if `c` maps a term `"g"` to the theorem `|- g = g'`, then the tactic `CONV_TAC c` reduces a goal `g` to the subgoal `g'`. More precisely, if `c "g"` returns `A' |- g = g'`, then:

```
      A ?- g
==============  CONV_TAC c
      A ?- g'
```

If `c` raises `UNCHANGED` then `CONV_TAC c` reduces a goal to itself.

Note that the conversion c should return a theorem whose assumptions are also among the assumptions of the goal (normally, the conversion will returns a theorem with no assumptions). CONV_TAC does not fail if this is not the case, but the resulting tactic will be invalid, so the theorem ultimately proved using this tactic will have more assumptions than those of the original goal.

Failure

CONV_TAC c applied to a goal A ?- g fails if c fails (other than by raising UNCHANGED) when applied to the term g. The function returned by CONV_TAC c will also fail if the ML function c:term->thm is not, in fact, a conversion (i.e. a function that maps a term t to a theorem |- t = t').

CONV_TAC is used to apply simplifications that can't be expressed as equations (rewrite rules). For example, a goal can be simplified by beta-reduction, which is not expressible as a single equation, using the tactic

```
CONV_TAC(DEPTH_CONV BETA_CONV)
```

The conversion BETA_CONV maps a beta-redex "(\x.u)v" to the theorem

```
|- (\x.u)v = u[v/x]
```

and the ML expression (DEPTH_CONV BETA_CONV) evaluates to a conversion that maps a term "t" to the theorem |- t=t' where t' is obtained from t by beta-reducing all beta-redexes in t. Thus CONV_TAC(DEPTH_CONV BETA_CONV) is a tactic which reduces beta-redexes anywhere in a goal.

See also

Abbrev.conv, Conv.UNCHANGED, Conv.CONV_RULE

current_axioms (Theory)

```
Theory.current_axioms : unit -> (string * thm) list
```

Return the axioms in the current theory segment.

An invocation current_axioms() returns a list of the axioms asserted in the current theory segment.

Failure

Never fails. If no axioms have been asserted, the empty list is returned.

See also

`Theory.current_theory`, `Theory.new_theory`, `Theory.current_definitions`, `Theory.current_theorems`, `Theory.constants`, `Theory.types`, `Theory.parents`

current_definitions (Theory)

```
Theory.current_definitions : unit -> (string * thm) list
```

Return the definitions in the current theory segment.

An invocation `current_definitions()` returns the list of definitions stored in the current theory segment. Every definition is automatically stored in the current segment by the primitive definition principles.

Advanced definition principles are built in terms of the primitives, so they also store their results in the cuurent segment. However, the definitions may be quite far removed from the user input, and they may also store some consequences of the definition as theorems.

Failure

Never fails. If no definitions have been made, the empty list is returned.

See also

`Theory.current_theory`, `Theory.new_theory`, `Theory.current_axioms`, `Theory.current_theorems`, `Theory.constants`, `Theory.types`, `Theory.parents`, `Definition.new_definition`, `Definition.new_specification`, `Definition.new_type_definition`, `TotalDefn.Define`, `IndDefLib.Hol_reln`

current_grammars (Parse)

```
Parse.current_grammars : unit -> type_grammar.grammar * term_grammar.grammar
```

Obtains the global type and term grammars.

HOL uses two global grammars to control the parsing and printing of term and type values. These can be adjusted in a controlled way with functions such as `add_rule` and `overload_on`.
`Parse.current_grammars ()` returns the current values of these grammars.

Failure

Never fails.

See also

`Parse.temp_set_grammars`, `Parse.term_grammar`, `Parse.type_grammar`

current_theorems (Theory)

```
Theory.current_theorems : unit -> (string * thm) list
```

Return the theorems stored in the current theory segment.

An invocation `current_theorems ()` returns the list of theorems stored in the current theory segment.

Failure

Never fails. If no theorems have been stored, the empty list is returned.

See also

`Theory.current_theory`, `Theory.new_theory`, `Theory.current_definitions`, `Theory.current_theorems`, `Theory.constants`, `Theory.types`, `Theory.parents`

current_theory (Theory)

```
Theory.current_theory : unit -> string
```

Returns the name of the current theory segment.

A HOL session has a notion of 'current theory'. There are two senses to this phrase. First, the current theory denotes the totality of all loaded theories plus whatever definitions, axioms, and theorems have been stored in the current session. In this sense, the current theory is the full logical context being used at the moment. This logical context can be extended in two ways: (a) by loading in prebuilt theories residing on disk; and (b) by making a definition, asserting an axiom, or storing a theorem. Therefore, the current theory consists of a body of prebuilt theories that have been loaded from disk (a collection of static components) plus whatever has been stored in the current session.

This latter component — what has been stored in the current session — embodies the second sense of 'current theory'. It is more properly known as the 'current theory segment'. The current segment is dynamic in nature, for its contents can be augmented and overwritten. It functions as a kind of scratchpad used to help build a static theory segment.

In a HOL session, there is always a single current theory segment. Its name is given by calling `current_theory()`. On startup, the current theory segment is called `"scratch"`, which is just a default name. If one is just experimenting, or hacking about, then this segment can be used.

On the other hand, if one intends to build a static theory segment, one usually creates a new theory segment named `thy` by calling `new_theory thy`. This changes the value of `current_theory` to `thy`. Once such a theory segment has been built (which may take many sessions), one calls `export_theory`, which exports the stored elements to disk.

Example

```
> current_theory();
val it = "wombat": string

> new_theory "foo";
val it = (): unit

> current_theory();
```

```
val it = "foo": string
```

Failure

Never fails.

See also

`Theory.new_theory`, `Theory.export_theory`

current_trace (Feedback)

```
Feedback.current_trace : string -> int
```

Returns the current value of the tracing variable specified.

Failure

Fails if the name given is not associated with a registered tracing variable.

See also

`Feedback.register_trace`, `Feedback.reset_trace`, `Feedback.reset_traces`, `Feedback.trace`, `Feedback.traces`

curry (Lib)

```
Lib.curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Converts a function on a pair to a corresponding curried function.

The application `curry f` returns `fn x => fn y => f(x,y)`, so that

```
curry f x y = f(x,y)
```

Failure

A call `curry f` never fails; however, `curry f x y` fails if `f (x,y)` fails.

Example

```
> val increment = curry op+ 1;
val increment = fn: int -> int

> increment 6;
val it = 7: int
```

See also

`Lib`, `Lib.uncurry`

CURRY_CONV (PairRules)

`PairRules.CURRY_CONV : conv`

Currys an application of a paired abstraction.

Example

```
> PairRules.CURRY_CONV (Term `(\(x,y). x + y) (1,2)`);
val it = ⊢ (λ(x,y). x + y) (1,2) = (λx y. x + y) 1 2: thm

> PairRules.CURRY_CONV (Term `(\(x,y). x + y) z`);
val it = ⊢ (λ(x,y). x + y) z = (λx y. x + y) (FST z) (SND z): thm
```

Failure

`CURRY_CONV tm` fails if `tm` is not an application of a paired abstraction.

See also

PairRules.UNCURRY_CONV

CURRY_EXISTS_CONV (PairRules)

PairRules.CURRY_EXISTS_CONV : conv

Currys paired existential quantifications into consecutive existential quantifications.

Example

```
> PairRules.CURRY_EXISTS_CONV (Term `?(x,y). x + y = y + x`);
val it = ⊢ (∃(x,y). x + y = y + x) ⇔ ∃x y. x + y = y + x: thm

> PairRules.CURRY_EXISTS_CONV (Term `?((w,x),(y,z)). w+x+y+z = z+y+x+w`);
val it =
   ⊢ (∃((w,x),y,z). w + x + y + z = z + y + x + w) ⇔
     ∃(w,x) (y,z). w + x + y + z = z + y + x + w: thm
```

Failure

CURRY_EXISTS_CONV tm fails if tm is not a paired existential quantification.

See also

PairRules.CURRY_CONV, PairRules.UNCURRY_CONV, PairRules.UNCURRY_EXISTS_CONV, PairRules.CURRY_FORALL_CONV, PairRules.UNCURRY_FORALL_CONV

CURRY_FORALL_CONV (PairRules)

PairRules.CURRY_FORALL_CONV : conv

Currys paired universal quantifications into consecutive universal quantifications.

Example

```
> PairRules.CURRY_FORALL_CONV (Term `!(x,y). x + y = y + x`);
val it = ⊢ (∀(x,y). x + y = y + x) ⇔ ∀x y. x + y = y + x: thm

> PairRules.CURRY_FORALL_CONV (Term `!((w,x),(y,z)). w+x+y+z = z+y+x+w`);
val it =
   ⊢ (∀((w,x),y,z). w + x + y + z = z + y + x + w) ⇔
     ∀(w,x) (y,z). w + x + y + z = z + y + x + w: thm
```

Failure

`CURRY_FORALL_CONV tm` fails if `tm` is not a paired universal quantification.

See also

`PairRules.CURRY_CONV`, `PairRules.UNCURRY_CONV`, `PairRules.UNCURRY_FORALL_CONV`, `PairRules.CURRY_EXISTS_CONV`, `PairRules.UNCURRY_EXISTS_CONV`

cv_auto_trans (cv_transLib)

```
cv_transLib.cv_auto_trans : thm -> unit
```

Translates functional definitions to the `cv_compute` subset of HOL.

This is a recursive version of `cv_transLib.cv_trans`. During translation of the given HOL function, `cv_transLib.cv_auto_trans` will call itself recursively on the definitions of any not-yet-translated constants it encounters.

As with all `auto` variants, `cv_transLib.cv_auto_trans` can sometimes translate uses of higher-order functions, such as `MAP`.

Failure

When the translation produces a precondition that `cv_transLib.cv_auto_trans` cannot prove automatically, or `cv_transLib.cv_termination_tac` fails to prove the termination goal of either a recursively translated function or the top-level translator-defined `:cv` function. For a failure on the top-level function, the termination goal is pushed onto the goal stack.

Example

```
> Definition list_add1_def:
   list_add1 xs = MAP SUC xs
  End
val list_add1_def = ⊢ ∀xs. list_add1 xs = MAP SUC xs: thm
> cv_transLib.cv_auto_trans list_add1_def;
Starting translation of list_add1 from fooTheory.
Starting translation of MAP_SUC from fooTheory.
Finished translating MAP_SUC, stored in cv_MAP_SUC_thm
Starting translation of list_add1 from fooTheory.
Finished translating list_add1, stored in cv_list_add1_thm
val it = (): unit
> cv_transLib.cv_eval “list_add1 [5; 6; 7]”;
val it = ⊢ list_add1 [5; 6; 7] = [6; 7; 8]: thm
```

Comments

Designed to produce definitions suitable for evaluation by `cv_transLib.cv_eval`.

See also

`cv_transLib.cv_trans`, `cv_transLib.cv_trans_pre`, `cv_transLib.cv_trans_pre_rec`, `cv_transLib.cv_trans_rec`, `cv_transLib.cv_auto_trans_pre`, `cv_transLib.cv_auto_trans_pre_rec`, `cv_transLib.cv_auto_trans_rec`, `cv_transLib.cv_eval`, `cv_transLib.cv_termination_tac`

cv_auto_trans_pre (cv_transLib)

```
cv_transLib.cv_auto_trans_pre : string -> thm -> thm
```

Translates functional definitions to the `cv_compute` subset of HOL.

This is a recursive version of `cv_transLib.cv_trans_pre`. During translation of the given HOL function, `cv_transLib.cv_auto_trans_pre` will call itself recursively on the definitions of any not-yet-translated constants it encounters.

As with all `auto` variants, `cv_transLib.cv_auto_trans_pre` can sometimes translate uses of higher-order functions, such as `MAP`.

Failure

When the translation produces a precondition that `cv_transLib.cv_auto_trans_pre` cannot prove automatically, or `cv_transLib.cv_termination_tac` fails to prove the termination goal of either a recursively translated function or the top-level translator-defined `:cv` function. For a failure on the top-level function, the termination goal is pushed onto the goal stack.

Example

See `cv_transLib.cv_auto_trans` and `cv_transLib.cv_trans_pre` for relevant examples.

Comments

Designed to produce definitions suitable for evaluation by `cv_transLib.cv_eval`.

See also

`cv_transLib.cv_trans`, `cv_transLib.cv_trans_pre`, `cv_transLib.cv_trans_pre_rec`, `cv_transLib.cv_trans_rec`, `cv_transLib.cv_auto_trans`, `cv_transLib.cv_auto_trans_pre_rec`, `cv_transLib.cv_auto_trans_rec`, `cv_transLib.cv_eval`, `cv_transLib.cv_termination_tac`

cv_auto_trans_pre_rec (cv_transLib)

```
cv_transLib.cv_auto_trans_pre_rec : string -> thm -> tactic -> thm
```

Translates functional definitions to the `cv_compute` subset of HOL.

This is a recursive version of `cv_transLib.cv_trans_pre_rec`. During translation of the given HOL function, `cv_transLib.cv_auto_trans_pre_rec` will call itself recursively on the definitions of any not-yet-translated constants it encounters.

As with all `auto` variants, `cv_transLib.cv_auto_trans_pre_rec` can sometimes translate uses of higher-order functions, such as `MAP`.

Failure

When the translation produces a precondition that `cv_transLib.cv_auto_trans_pre_rec` cannot prove automatically, or `cv_transLib.cv_termination_tac` fails to prove the termination goal of any recursively translated function, or the provided tactic fails to prove the termination goal of the top-level translator-defined `:cv` function.

Example

See `cv_transLib.cv_auto_trans` and `cv_transLib.cv_trans_pre_rec` for relevant examples.

Comments

Designed to produce definitions suitable for evaluation by `cv_transLib.cv_eval`.

See also

`cv_transLib.cv_trans`, `cv_transLib.cv_trans_pre`, `cv_transLib.cv_trans_pre_rec`, `cv_transLib.cv_trans_rec`, `cv_transLib.cv_auto_trans`, `cv_transLib.cv_auto_trans_pre`, `cv_transLib.cv_auto_trans_rec`, `cv_transLib.cv_eval`, `cv_transLib.cv_termination_tac`

cv_auto_trans_rec (cv_transLib)

```
cv_transLib.cv_auto_trans_rec : thm -> tactic -> unit
```

Translates functional definitions to the `cv_compute` subset of HOL.

This is a recursive version of `cv_transLib.cv_trans_rec`. During translation of the given HOL function, `cv_transLib.cv_auto_trans_pre_rec` will call itself recursively on the definitions of any not-yet-translated constants it encounters.

As with all `auto` variants, `cv_transLib.cv_auto_trans_rec` can sometimes translate uses of higher-order functions, such as `MAP`.

Failure

When the translation produces a precondition that `cv_transLib.cv_auto_trans_rec` cannot prove automatically, or `cv_transLib.cv_termination_tac` fails to prove the termination goal of any recursively translated function, or the provided tactic fails to prove the termination goal of the top-level translator-defined `:cv` function.

Example

See `cv_transLib.cv_auto_trans` and `cv_transLib.cv_trans_rec` for relevant examples.

Comments

Designed to produce definitions suitable for evaluation by `cv_transLib.cv_eval`.

See also

`cv_transLib.cv_trans`, `cv_transLib.cv_trans_pre`, `cv_transLib.cv_trans_pre_rec`, `cv_transLib.cv_trans_rec`, `cv_transLib.cv_auto_trans`, `cv_transLib.cv_auto_trans_pre`, `cv_transLib.cv_auto_trans_pre_rec`, `cv_transLib.cv_eval`, `cv_transLib.cv_termination_tac`

cv_eval (cv_transLib)

```
cv_transLib.cv_eval : term -> thm
```

Uses cv_computeLib to evaluate closed terms, equipped with translations from cv_transLib.

Provides a user-friendly interface to cv_computeLib.cv_compute, as long as cv_transLib has been used to translate all constants in the given input term.

Failure

When the input term contains either free variables or constants which have not yet been translated.

Example

```
> cv_transLib.cv_trans arithmeticTheory.FACT;
Finished translating FACT, stored in cv_FACT_thm
val it = (): unit
> cv_transLib.cv_eval “FACT 50”;
val it =
   ⊢ FACT 50 =
     30414093201713378043612608166064768844377641568960512000000000000: thm
```

See also

cv_transLib.cv_trans, cv_transLib.cv_trans_pre, cv_transLib.cv_trans_pre_rec, cv_transLib.cv_trans_rec, cv_transLib.cv_auto_trans, cv_transLib.cv_auto_trans_pre, cv_transLib.cv_auto_trans_pre_rec, cv_transLib.cv_auto_trans_rec, cv_transLib.cv_eval_raw, cv_transLib.cv_termination_tac

cv_eval_raw (cv_transLib)

```
cv_transLib.cv_eval_raw : term -> thm
```

Uses `cv_computeLib` to evaluate closed terms, equipped with translations from `cv_transLib`.

Like `cv_transLib.cv_eval`, except it omits the potentially expensive evaluation out of the `:cv` type.

Failure

Fails in the same ways as `cv_transLib.cv_eval`.

Example

```
> cv_transLib.cv_trans rich_listTheory.REPLICATE;
Finished translating REPLICATE, stored in cv_REPLICATE_thm
val it = (): unit
> cv_transLib.cv_eval “REPLICATE 3 (3:num)”;
val it = ⊢ REPLICATE 3 3 = [3; 3; 3]: thm
> cv_transLib.cv_eval_raw “REPLICATE 3 (3:num)”;
val it =
   ⊢ REPLICATE 3 3 =
     cv_type$to_list cv$c2n
       (cv$Pair (cv$Num 3)
          (cv$Pair (cv$Num 3) (cv$Pair (cv$Num 3) (cv$Num 0)))): thm
```

See also

`cv_transLib.cv_eval`

cv_termination_tac (cv_transLib)

```
cv_transLib.cv_termination_tac : tactic
```

A tactic for simplifying goals concerning `cv_depth_sum`.

This tactic is used by cv_transLib.cv_trans (and siblings) when attempting to prove termination goals of translated :cv functions.

See also

cv_transLib.cv_trans, cv_transLib.cv_trans_pre, cv_transLib.cv_trans_pre_rec, cv_transLib.cv_trans_rec, cv_transLib.cv_auto_trans, cv_transLib.cv_auto_trans_pre, cv_transLib.cv_auto_trans_pre_rec, cv_transLib.cv_auto_trans_rec, cv_transLib.cv_eval

cv_trans (cv_transLib)

```
cv_transLib.cv_trans : thm -> unit
```

Translates functional definitions to the cv_compute subset of HOL.

This function is the same as cv_transLib.cv_trans_pre, except that it tries to discharge any preconditions automatically.

Failure

When translation produces a precondition that cv_transLib.cv_trans cannot prove automatically, or encounters a sub-term containing a constant that has not already been translated, or cv_transLib.cv_termination_tac fails to prove the termination goal of the translator-defined :cv function. In the latter case, the termination goal is pushed onto the goal stack.

Example

```
> cv_transLib.cv_trans arithmeticTheory.FACT;
Finished translating FACT, stored in cv_FACT'_thm
val it = (): unit
> cv_transLib.cv_eval “FACT 50”;
val it =
   ⊢ FACT 50 =
     30414093201713378043612608166064768844377641568960512000000000000: thm
```

Comments

Designed to produce definitions suitable for evaluation by `cv_transLib.cv_eval`.

See also

`cv_transLib.cv_trans_pre`, `cv_transLib.cv_trans_pre_rec`, `cv_transLib.cv_trans_rec`, `cv_transLib.cv_auto_trans`, `cv_transLib.cv_auto_trans_pre`, `cv_transLib.cv_auto_trans_pre_rec`, `cv_transLib.cv_auto_trans_rec`, `cv_transLib.cv_eval`, `cv_transLib.cv_termination_tac`

cv_trans_deep_embedding (cv_transLib)

```
cv_transLib.cv_trans_deep_embedding : conv -> thm -> unit
```

Translates equations defining a deeply embedded AST to the `cv_compute` subset of HOL.

This function is similar to `cv_transLib.cv_trans`, but can only translate constants. It is designed for the translation of large deep embeddings to `:cv` functions. It takes as an argument a conversion which must evaluate terms such as `from <deep_embedding>` (e.g. `computeLib.EVAL`).

Failure

When the input term is not a constant defining a suitable deep embedding.

Example

```
> Datatype:
   exp = Const num | Add exp exp
  End
> Definition sem_def:
   sem (Const n) = n ∧
   sem (Add e1 e2) = sem e1 + sem e2
  End
```

```
val sem_def =
   ⊢ (∀n. sem (Const n) = n) ∧ ∀e1 e2. sem (Add e1 e2) = sem e1 + sem e2: thm
> Definition deep_def:
   deep = Add (Const 5) (Add (Const 2) (Const 3))
  End
val deep_def = ⊢ deep = Add (Const 5) (Add (Const 2) (Const 3)): thm
> cv_transLib.cv_trans sem_def;
Finished translating sem, stored in cv_sem_thm
val it = (): unit
> cv_transLib.cv_trans_deep_embedding EVAL deep_def;
val it = (): unit
> cv_transLib.cv_eval “sem deep”;
val it = ⊢ sem deep = 10: thm
```

Comments

Designed to produce definitions suitable for evaluation by `cv_transLib.cv_eval`.

See also

`cv_transLib.cv_trans`, `cv_transLib.cv_eval`

cv_trans_pre (cv_transLib)

```
cv_transLib.cv_trans_pre : string -> thm -> thm
```

Translates functional definitions to the `cv_compute` subset of HOL.

Accepts a theorem describing a functional definition. Attempts to translate this to a function operating over the `:cv` type, used by `cv_computeLib.cv_compute`. Returns a precondition representing the proof obligation which must be discharged before this translated function can be evaluated with `cv_transLib.cv_eval`.

Failure

When the translation does not produce a precondition, or encounters a sub-term containing a constant that has not already been translated, or `cv_transLib.cv_termination_-`

`tac` fails to prove the termination goal of the translator-defined `:cv` function.

Example

```
> cv_transLib.cv_trans_pre "HD_pre" listTheory.HD;
Finished translating HD, stored in cv_HD_thm

WARNING: definition of cv_HD has a precondition.
You can set up the precondition proof as follows:

Theorem HD_pre[cv_pre]:
  ∀v. HD_pre v
Proof
  ho_match_mp_tac listTheory.HD_ind (* for example *)
  ...
QED

val it = ⊢ ∀v. HD_pre v ⇔ (∃t h. v = h::t) ∧ v ≠ []: thm
```

Comments

Designed to produce definitions suitable for evaluation by `cv_transLib.cv_eval`.

See also

`cv_transLib.cv_trans`, `cv_transLib.cv_trans_pre_rec`, `cv_transLib.cv_trans_rec`, `cv_transLib.cv_auto_trans`, `cv_transLib.cv_auto_trans_pre`, `cv_transLib.cv_auto_trans_pre_rec`, `cv_transLib.cv_auto_trans_rec`, `cv_transLib.cv_eval`, `cv_transLib.cv_termination_tac`

cv_trans_pre_rec (cv_transLib)

```
cv_transLib.cv_trans_pre_rec : string -> thm -> tactic -> thm
```

Translates functional definitions to the `cv_compute` subset of HOL.

This function is the same as `cv_transLib.cv_trans_pre`, except that it also takes a user-provided tactic for proving termination of the translator-defined `:cv` function.

Failure

When the translation encounters a sub-term containing a constant that has not already been translated, or the provided tactic fails to prove the termination goal of the translator-defined `:cv` function.

Example

```
> Definition count_up_def:
   count_up m k = if m < k:num then 1 + count_up (m+1) k else 0:num
  Termination
   WF_REL_TAC ‘measure $ λ(m,k). k - m:num’
  End;
val count_up_def =
   ⊢ ∀m k. count_up m k = if m < k then 1 + count_up (m + 1) k else 0: thm
> val cv_count_up_pre = cv_transLib.cv_trans_pre_rec "count_pre" count_up_def
   (WF_REL_TAC ‘measure $ λ(m,k). cv$c2n k - cv$c2n m’
    \\ Cases \\ Cases \\ gvs [] \\ rw [] \\ gvs []);
Finished translating count_up, stored in cv_count_up_thm

WARNING: definition of cv_count_up has a precondition.
You can set up the precondition proof as follows:

Theorem count_pre[cv_pre]:
  ∀m k. count_pre m k
Proof
  ho_match_mp_tac count_up_ind (* for example *)
  ...
QED

val cv_count_up_pre = ⊢ ∀m k. count_pre m k ⇔ m < k ⇒ count_pre (m + 1) k:
   thm
> Theorem count_up_pre[cv_pre]:
  ∀m k. count_up_pre m k
  Proof
```

```
  ho_match_mp_tac count_up_ind \\ rw []
  \\ simp [Once cv_count_up_pre]
  QED
Exception- HOL_ERR
  (at Q.store_thm:
     at boolLib.store_thm_at:
     at Tactic.HO_MATCH_MP_TAC:
     at HolKernel.ho_match_term:
     at ??.failwith:
       Failed to prove theorem "count_up_pre":
match: safe_insert) raised
> cv_transLib.cv_eval “count_up 5 100”;
val it = ⊢ count_pre 5 100 ⇒ count_up 5 100 = 95: thm
```

Comments

Designed to produce definitions suitable for evaluation by `cv_transLib.cv_eval`.

See also

`cv_transLib.cv_trans`, `cv_transLib.cv_trans_pre`, `cv_transLib.cv_trans_rec`, `cv_transLib.cv_auto_trans`, `cv_transLib.cv_auto_trans_pre`, `cv_transLib.cv_auto_trans_pre_rec`, `cv_transLib.cv_auto_trans_rec`, `cv_transLib.cv_eval`, `cv_transLib.cv_termination_tac`

cv_trans_rec (cv_transLib)

```
cv_transLib.cv_trans_rec : thm -> tactic -> unit
```

Translates functional definitions to the `cv_compute` subset of HOL.

This function is the same as `cv_transLib.cv_trans`, except that it also takes a user-provided tactic for proving termination of the translator-defined `:cv` function.

Failure

When translation produces a precondition that cv_transLib.cv_trans cannot prove automatically, or encounters a sub-term containing a constant that has not already been translated, or the provided tactic fails to prove the termination goal of the translator-defined :cv function.

Example

```
> Definition count_up_def:
   count_up m k = if m < k:num then 1 + count_up (m+1) k else 0:num
  Termination
   WF_REL_TAC ‘measure $ λ(m,k). k - m:num’
  End;
val count_up_def =
   ⊢ ∀m k. count_up m k = if m < k then 1 + count_up (m + 1) k else 0: thm
> cv_transLib.cv_trans_rec count_up_def
   (WF_REL_TAC ‘measure $ λ(m,k). cv$c2n k - cv$c2n m’
    \\ Cases \\ Cases \\ gvs [] \\ rw [] \\ gvs []);
Finished translating count_up, stored in cv_count_up'_thm
val it = (): unit
> cv_transLib.cv_eval “count_up 5 100”;
val it = ⊢ count_up 5 100 = 95: thm
```

Comments

Designed to produce definitions suitable for evaluation by cv_transLib.cv_eval.

See also

cv_transLib.cv_trans, cv_transLib.cv_trans_pre, cv_transLib.cv_trans_pre_rec, cv_transLib.cv_auto_trans, cv_transLib.cv_auto_trans_pre, cv_transLib.cv_auto_trans_pre_rec, cv_transLib.cv_auto_trans_rec, cv_transLib.cv_eval, cv_transLib.cv_termination_tac

data (DB)

```
DB.type data
```

Type abbreviation used in `DB` structure.

When functions from the `DB` structure are used to query the current theory, answer are often phrased in terms of the `data` type, which is a type abbreviation declared as

```
type data = (string * string) * (thm * class)
```

An element `((thy,name), (th,cl))` means that `th` is a theorem with classification `class`, stored in theory segment `thy` under `name`.

Example

```
> DB.find "BOOL_CASES_AX";
val it =
   [(("bool", "BOOL_CASES_AX"),
     (⊢ ∀t. (t ⇔ T) ∨ (t ⇔ F), Axm,
      Located
       {exact = true, linenum = 228, scriptpath =
        "$(HOLDIR)/src/bool/boolScript.sml"}))]: public_data list
```

See also

`DB.class`, `DB.thy`, `DB.find`, `DB.match`, `DB.apropos`, `DB.listDB`

Datatype (bossLib)

```
bossLib.Datatype : hol_type quotation -> unit
```

Define a concrete datatype.

Many formalizations require the definition of new types. For example, ML-style datatypes are commonly used to model the abstract syntax of programming languages and the state-space of elaborate transition systems. In HOL, such datatypes (at least, those that are inductive, or, alternatively, have a model in an initial algebra) may be specified using the invocation `Datatype `<spec>``, where `<spec>` should conform to the following grammar:

```
spec    ::= [ <binding> ; ]* <binding>

binding ::= <ident> = [ <clause> | ]* <clause>
          |  <ident> = <| [ <ident> : <type> ; ]* <ident> : <type> |>

clause  ::= <ident> <tyspec>*

tyspec  ::= ( <type> )
          |  <atomic-type>
```

where `<atomic-type>` is a single token denoting a type. For example, `num`, `bool` and `'a`.

When a datatype is successfully defined, a number of standard theorems are automatically proved about the new type: the constructors of the type are proved to be injective and disjoint, induction and case analysis theorems are proved, and each type also has a 'size' function defined for it. All these theorems are stored in the current theory and added to a database accessed via the functions in `TypeBase`.

The notation used to declare datatypes is, unfortunately, not the same as that of ML. If anything, the syntax is rather more like Haskell's. For example, an ML declaration

```
datatype ('a,'b) btree = Leaf of 'a
                       | Node of ('a,'b) btree * 'b * ('a,'b) btree
```

would most likely be declared in HOL as

```
Datatype `btree = Leaf 'a
                | Node btree 'b btree`
```

Note that any type parameters for the new type are not allowed; they are inferred from the right hand side of the binding. The type variables in the specification become arguments to the new type operator in alphabetic order.

When a record type is defined, the parser is adjusted to allow new syntax (appropriate for records), and a number of useful simplification theorems are also proved. The most useful of the latter are automatically stored in the `TypeBase` and can be inspected using the `simpls_of` function. For further details on record types, see the DESCRIPTION.

Example

In the following, we shall give an overview of the kinds of types that may be defined by `Datatype`.

To start, enumerated types can be defined as in the following example:

```
Datatype `enum = A1  | A2  | A3  | A4  | A5
               | A6  | A7  | A8  | A9  | A10
               | A11 | A12 | A13 | A14 | A15
               | A16 | A17 | A18 | A19 | A20
               | A21 | A22 | A23 | A24 | A25
               | A26 | A27 | A28 | A29 | A30`
```

Other non-recursive types may be defined as well:

```
Datatype `foo = N num
              | B bool
              | Fn ('a -> 'b)
              | Pr ('a # 'b`)
```

Turning to recursive types, we can define a type of binary trees where the leaves are numbers.

```
Datatype `tree = Leaf num | Node tree tree`
```

We have already seen a type of binary trees having polymorphic values at internal nodes. This time, we will declare it in "paired" format.

```
Datatype `tree = Leaf 'a
               | Node (tree # 'b # tree)`
```

This specification seems closer to the declaration that one might make in ML, but is more difficult to deal with in proof than the curried format used above.

The basic syntax of the named lambda calculus is easy to describe:

```
- load "stringTheory";
> val it = () : unit

- Datatype `lambda = Var string
                   | Const 'a
                   | Comb lambda lambda
                   | Abs lambda lambda`
```

The syntax for 'de Bruijn' terms is roughly similar:

```
Datatype `dB = Var string
             | Const 'a
             | Bound num
             | Comb dB dB
             | Abs dB`
```

Arbitrarily branching trees may be defined by allowing a node to hold the list of its subtrees. In such a case, leaf nodes do not need to be explicitly declared.

```
Datatype `ntree = Node of 'a (ntree list)`
```

A (tupled) type of 'first order terms' can be declared as follows:

```
Datatype `term = Var string
               | Fnapp (string # term list)`
```

Mutally recursive types may also be defined. The following, extracted by Elsa Gunter from the Definition of Standard ML, captures a subset of Core ML.

```
Datatype
 `atexp = var_exp string
        | let_exp dec exp ;

    exp = aexp    atexp
        | app_exp exp atexp
        | fn_exp  match ;

  match = match  rule
        | matchl rule match ;

   rule = rule pat exp ;

    dec = val_dec   valbind
        | local_dec dec dec
        | seq_dec   dec dec ;

valbind = bind  pat exp
```

```
          | bindl pat exp valbind
          | rec_bind valbind ;

      pat = wild_pat
          | var_pat string`
```

Simple record types may be introduced using the `<| ... |>` notation.

```
Datatype `state = <| Reg1 : num; Reg2 : num; Waiting : bool |>`
```

The use of record types may be recursive. For example, the following declaration could be used to formalize a simple file system.

```
Datatype
  `file = Text string
        | Dir directory
   ;
   directory = <| owner : string ;
                  files : (string # file) list |>`
```

Failure

Now we address some types that cannot be declared with `Datatype`. In some cases they cannot exist in HOL at all; in others, the type can be built in the HOL logic, but `Datatype` is not able to make the definition.

First, an empty type is not allowed in HOL, so the following attempt is doomed to fail.

```
Datatype `foo = A foo`
```

So called 'nested types', which are occasionally quite useful, cannot at present be built with `Datatype`:

```
Datatype `btree = Leaf 'a
                | Node (('a # 'a) btree)`
```

Co-algebraic types may not currently be built with `Datatype`, not even by attempting to encode the remainder of the list as a function:

```
Datatype `lazylist = Nil
                   | Cons ('a # (one -> lazylist))`
```

Indeed, this specification corresponds to an algebraic type isomorphic to "standard" lists, but Datatype rejects it because it cannot handle recursion to the right of a function arrow. The type of co-algebraic lists can be built in HOL: see llistTheory.

Finally, for cardinality reasons, HOL does not allow the following attempt to model the untyped lambda calculus as a set (note the -> in the clause for the Abs constructor):

```
Datatype `lambda = Var string
                 | Const 'a
                 | Comb lambda lambda
                 | Abs (lambda -> lambda)`
```

Instead, one would have to build a theory of complete partial orders (or something similar) with which to model the untyped lambda calculus.

Comments

The consequences of an invocation of Datatype are stored in the current theory segment and in TypeBase. The principal consequences of a datatype definition are the primitive recursion and induction theorems. These provide the ability to define simple functions over the type, and an induction principle for the type. For a type named ty, the primitive recursion theorem is stored under ty_Axiom and the induction theorem is put under ty_induction. Other consequences include the distinctness of constructors (ty_distinct), and the injectivity of constructors (ty_11). A 'degenerate' version of ty_induction is also stored under ty_nchotomy: it provides for reasoning by cases on the construction of elements of ty. Finally, some special-purpose theorems are stored: ty_case_cong gives a congruence theorem for "case" statements on elements of ty. These case statements are introduced by ty_case_def. Also, a definition of the "size" of the type is added to the current theory, under the name ty_size_def.

For example, invoking

```
Datatype `tree = Leaf num | Node tree tree`;
```

results in the definitions

```
tree_case_def =
  |- (!a f f1. tree_CASE (Leaf a) f f1 = f a) /\
     !a0 a1 f f1. tree_CASE (Node a0 a1) f f1 = f1 a0 a1

tree_size_def
```

```
|- (!a. tree_size (Leaf a) = 1 + a) /\
    !a0 a1. tree_size (Node a0 a1) = 1 + (tree_size a0 + tree_size a1)
```

being added to the current theory. The following theorems about the datatype are also stored in the current theory.

```
tree_Axiom
  |- !f0 f1.
       ?fn. (!a. fn (Leaf a) = f0 a) /\
            !a0 a1. fn (Node a0 a1) = f1 a0 a1 (fn a0) (fn a1)

tree_induction
  |- !P. (!n. P (Leaf n)) /\
         (!t t0. P t /\ P t0 ==> P (Node t t0))
         ==>
         !t. P t

tree_nchotomy  |- !t. (?n. t = Leaf n) \/ ?t' t0. t = Node t' t0

tree_11
  |- (!a a'. (Leaf a = Leaf a') = (a = a')) /\
      !a0 a1 a0' a1'. (Node a0 a1 = Node a0' a1') = (a0=a0') /\ (a1=a1')

tree_distinct  |- !a1 a0 a. Leaf a <> Node a0 a1

tree_case_cong
  |- !M M' f f1.
       (M = M') /\
       (!a. (M' = Leaf a) ==> (f a = f' a)) /\
       (!a0 a1. (M' = Node a0 a1) ==> (f1 a0 a1 = f1' a0 a1))
       ==>
       (tree_CASE M f f1 = tree_CASE M' f' f1')
```

When a type involving records is defined, many more definitions are made and added to the current theory.

A definition of mutually recursives types results in the above theorems and definitions being added for each of the defined types.

See also

`Definition.new_type_definition`, `TotalDefn.Define`, `IndDefLib.Hol_reln`, `TypeBase`

datatype_theorems (EmitTeX)

```
EmitTeX.datatype_theorems : string -> (string * thm) list
```

All the datatype theorems stored in the named theory.

An invocation `datatype_theorems thy`, where `thy` is the name of a currently loaded theory segment, will return a list of the datatype theorems stored in that theory. Each theorem is paired with the name of the datatype in the result. The string "-" may be used to denote the current theory segment.

Failure

Never fails. If `thy` is not the name of a currently loaded theory segment, the empty list is returned.

Example

```
> new_theory "example";
Exporting theory "scratch" ... done.
Theory "scratch" took 0.01042s to build
<<HOL message: Created theory "example">>
val it = (): unit
> val _ = Hol_datatype `example = First | Second`;
<<HOL message: Defined type: "example">>
> EmitTeX.datatype_theorems "example";
val it = [("example", ⊢ DATATYPE (example First Second))]:
   (string * thm) list
```

See also

`DB.theorems`, `bossLib.Hol_datatype`

datatype_thm_to_string (EmitTeX)

```
EmitTeX.datatype_thm_to_string : thm -> string
```

Converts a datatype theorem to a string.

An invocation of `datatype_thm_to_string thm`, where `thm` is a datatype theorem produced by `Hol_datatype`, will return a string that corresponds with the orginal datatype declaration.

Failure

Will fail if the supplied theorem is not a datatype theorem, as created by `Hol_datatype`.

Example

```
> new_theory "example";
<<HOL message: Restarting theory "example">>
<<HOL warning: Theory.callhooks: Hook Parse.clear_consts_from_grammar failed on event Ne
Exception raised at Term.dest_thy_const: not a const
>>
val it = (): unit
> val _ = Hol_datatype `example = First | Second`;
<<HOL message: Defined type: "example">>
> EmitTeX.datatype_thm_to_string (theorem "datatype_example");
val it = "example = First | Second": string
```

See also

`bossLib.Hol_datatype`

DECIDE (bossLib)

```
bossLib.DECIDE : term -> thm
```

Invoke decision procedure(s).

An application `DECIDE M`, where `M` is a boolean term, attempts to prove `M` using a propositional tautology checker and a linear arithmetic decision procedure.

Failure

The invocation fails if `M` is not of boolean type. It also fails if `M` is not a tautology or an instance of a theorem of linear arithmetic.

Example

```
> DECIDE (Term `p /\ p /\ r ==> r`);
val it = ⊢ p ∧ p ∧ r ⇒ r: thm

> DECIDE (Term `x < 17 /\ y < 26 ==> x + y < 17 + 26`);
val it = ⊢ x < 17 ∧ y < 26 ⇒ x + y < 17 + 26: thm
```

Comments

`DECIDE` is currently somewhat underpowered. Formerly it was implemented by a cooperating decision procedure mechanism. However, most proofs seemed to go somewhat smoother with simplification using the `arith_ss` simpset, so we have adopted a simpler implementation. That should not be taken as final, since cooperating decision procedures are an important component in highly automated proof systems.

See also

`bossLib.RW_TAC`, `bossLib.arith_ss`

DECIDE_TAC (bossLib)

`bossLib.DECIDE_TAC : tactic`

Invoke decision procedure(s).

DECIDE_TAC is the tactical version of DECIDE.

Failure

As for DECIDE

See also

bossLib.DECIDE

declare_monad (monadsyntax)

```
monadsyntax.declare_monad :
  string * { bind : term, unit : term, ignorebind : term option,
             choice : term option, fail : term option, guard : term option }
    ->
  unit
```

Declares a monad type for which the do/od syntax can be used.

A call to `declare_monad(mname, minfo)` alters the internal "monad database" so that a subsequent call to `enable_monad mname` will cause do/od syntax to try to use the terms in `minfo` as interpretations of that syntax. The only compulsory values are the `unit` and `bind` values, which should have types conforming to the pattern `:α M` and `:α -> β M` respectively. For example, the list monad would have `M` instantiated by the pattern `:_ list`, while the reader monad would have `M` instantiated by the pattern `:'env -> _`.

The `ignorebind` field allows the user to provide a specific constant to interpret a `bind` where the second argument ignores the value. If this is not provided, then syntax such as `do M1; M2; od` will be interpreted as `bind M1 (K M2)`, where `K` is the constant combinator.

The remaining fields are used when the monad has a notion of failure. For example, the option monad uses `NONE` as the appropriate value for `fail`. The `choice` term should be of type `:α M -> α M -> α M`, and should return the first value if it is not a failure, or otherwise use the second argument. The supported syntax for `choice` is ++.

Finally, the guard field should be a term of type :bool -> unit M. It is rendered as assert b with b a boolean value. If b is true, the monad "returns" the unit value; if b is false the monad fails.

The information declared with a call to declare_monad is exported with the current theory and is thus available to descendent theories.

Failure

Never fails. However, the terms present in the monad-information record must have appropriate types if strange type-checking errors on subsequent uses of the do/od syntax are to be avoided.

Example

A set monad could be declared:

```
> monadsyntax.declare_monad("set", {
   unit = "λa. {a}", bind = "λs f. BIGUNION (IMAGE f s)",
   ignorebind = NONE,
   fail = SOME "{}", guard = SOME "λb. if b then {()} else {}",
   choice = SOME "$UNION"
  });
val it = (): unit
```

Comments

This function does not even care if the constants have the right respective types; it certainly doesn't care if the constants satisfy the monadic axioms.

See also

monadsyntax.all_monads, monadsyntax.enable_monad

decls (Term)

```
Term.decls : string -> term list
```

Returns a list of constants having the same name.

An invocation `Term.decls s` returns a list of constants found in the current theory having the name `s`. If there are no constants with name `s`, then the empty list is returned.

Failure

Never fails.

Example

```
> decls "+";
val it = [“$+”]: term list

> map dest_thy_const it;
val it = [{Name = "+", Thy = "arithmetic", Ty = “:num -> num -> num”}]:
   {Name: string, Thy: string, Ty: hol_type} list
```

Comments

Useful for untangling confusion arising from overloading and also the possibility to declare two different constants with the same name in different theories.

See also

`Type.decls`, `Term.dest_thy_const`

decls (Type)

```
Type.decls : string -> {Thy : string, Tyop : string} list
```

Lists all theories a named type operator is declared in.

An invocation `Type.decls s` finds all theories in the ancestry of the current theory with a type constant having the given name.

Failure

Never fails.

Example

```
> Type.decls "prod";
val it = [{Thy = "pair", Tyop = "prod"}]: {Thy: string, Tyop: string} list
```

Comments

There is also a function `Term.decls` that performs a similar operation on term constants.

See also

`Theory.ancestry`, `Term.decls`, `Theory.constants`

DEEP_INTRO_TAC (Tactic)

```
Tactic.DEEP_INTRO_TAC : thm -> tactic
```

Applies an introduction-rule backwards; instantiating a predicate variable.

The function `DEEP_INTRO_TAC` expects a theorem of the form

```
antecedents ==> P (term-pattern)
```

where `P` is a variable, and `term-pattern` is a pattern describing the form of an expected sub-term in the goal. When `th` is of this form, the tactic `DEEP_INTRO_TAC th` finds a higher-order instantiation for the variable `P` and a first order instantiation for the variables in `term-pattern` such that the instantiated conclusion of `th` is identical to the goal. It then applies `MATCH_MP_TAC` to turn the goal into an instantiation of the antecedents of `th`.

Failure

Fails if there is no (free) instance of `term-pattern` in the goal. Also fails if `th` is not of the required form.

Example

The theorem `SELECT_ELIM_THM` states

```
|- !P Q. (?x. P x) /\ (!x. P x ==> Q x) ==> Q ($@ P)
```

This is of the required form for use by `DEEP_INTRO_TAC`, and can be used to transform a goal mentioning Hilbert Choice (the `@` operator) into one that doesn't. Indeed, this is how `SELECT_ELIM_TAC` is implemented.

See also

`Tactic.MATCH_MP_TAC`, `Tactic.SELECT_ELIM_TAC`

Define (TotalDefn)

```
TotalDefn.Define : term quotation -> thm
```

Re-exported from `bossLib.Define`. See that entry for full documentation.

Define (bossLib)

```
bossLib.Define : term quotation -> thm
```

Also exported as `TotalDefn.Define`.

General-purpose function definition facility.

`Define` takes a high-level specification of a HOL function, and attempts to define the function in the logic. If this attempt is successful, the specification is derived from the definition. The derived specification is returned to the user, and also stored in the current theory. `Define` may be used to define abbreviations, recursive functions, and mutually recursive functions. An induction theorem may be stored in the current theory as a by-product of `Define`'s activity. This induction theorem follows the recursion structure of the function, and may be useful when proving properties of the function.

`Define` takes as input a quotation representing a conjunction of equations. The specified function(s) may be phrased using ML-style pattern-matching. A call `Define` `<spec>` should conform with the following grammar:

```
      spec ::= <eqn>
           |   (<eqn>) /\ <spec>

       eqn ::= <alphanumeric> <pat> ... <pat> = <term>


       pat ::= <variable>
           |   <wildcard>
           |   <cname>                            (* 0-ary constructor *)
           |   (<cname>_n <pat>_1 ... <pat>_n)    (* constructor appl. *)

     cname ::= <alphanumeric> | <symbolic>

  wildcard ::= _
           |   _<wildcard>
```

When processing the specification of a recursive function, `Define` must perform a termination proof. It automatically constructs termination conditions for the function, and invokes a termination prover in an attempt to prove the termination conditions.

If the function is primitive recursive, in the sense that it exactly follows the recursion pattern of a previously declared HOL datatype, then this proof always succeeds, and `Define` stores the derived equations in the current theory segment. Otherwise, the function is not an instance of primitive recursion, and the termination prover may succeed or fail.

If it succeeds, then `Define` stores the specified equations in the current theory segment. An induction theorem customized for the defined function is also stored in the current segment. Note, however, that an induction theorem is not stored for primitive recursive functions, since that theorem would be identical to the induction theorem resulting from the declaration of the datatype.

If the termination proof fails, then `Define` fails.

In general, `Define` attempts to derive exactly the specified conjunction of equations. However, the rich syntax of patterns allows some ambiguity. For example, the input

```
Define `(f 0 _ = 1)
  /\    (f _ 0 = 2)`
```

is ambiguous at `f 0 0`: should the result be `1` or `2`? The system attempts to resolve this ambiguity in the same way as compilers and interpreters for functional languages. Namely, a conjunction of equations is treated as being processed left-conjunct first, followed by processing the right conjunct. Therefore, in the example above, the right-hand side of the first clause is taken as the value of `f 0 0`. In the implementation, ambiguities arising from such overlapping patterns are systematically translated away in a pre-processing step.

Another case of vagueness in patterns is shown above: the specification is 'incomplete' since it does not tell us how `f` should behave when applied to two non-zero arguments: e.g., `f (SUC m) (SUC n)`. In the implementation, such missing clauses are filled in, and have the value `ARB`. This 'pattern-completion' step is a way of turning descriptions of partial functions into total functions suitable for HOL. However, since the user has not completely specified the function, the system takes that as a hint that the user is not interested in using the function at the missing-but-filled-in clauses, and so such clauses are dropped from the final theorem.

In summary, `Define` will derive the unambiguous and complete equations

```
|- (f 0 (SUC v4) = 1) /\
   (f 0 0 = 1) /\
   (f (SUC v2) 0 = 2)
   (f (SUC v2) (SUC v4) = ARB)
```

from the above ambiguous and incomplete equations. The odd-looking variable names are due to the pre-processing steps described above. The above result is only an intermediate value: in the final result returned by `Define`, the last equation is droppped:

```
|- (f 0 (SUC v4) = 1) /\
   (f 0 0 = 1) /\
   (f (SUC v2) 0 = 2)
```

`Define` automatically generates names with which to store the definition and, (if it exists) the associated induction theorem, in the current theory. The name for storing the definition is built by concatenating the name of the function with the value of the reference variable `Defn.def_suffix`. The name for storing the induction theorem is built by concatenating the name of the function with the value of the reference variable `Defn.ind_suffix`. For mutually recursive functions, where there is a choice of names, the name of the function in the first clause is taken.

Since the names used to store elements in the current theory segment are transformed into ML bindings after the theory is exported, it is required that every invocation of

Define generates names that will be valid ML identifiers. For this reason, Define requires alphanumeric function names. If one wishes to define symbolic identifiers, the ML function xDefine should be used.

Failure

Define fails if its input fails to parse and typecheck.

Define fails if the name of the function being defined is not alphanumeric.

Define fails if there are more free variables on the right hand sides of the recursion equations than the left.

Define fails if it cannot prove the termination of the specified recursive function. In that case, one has to embark on the following multi-step process in order to get the same effect as if Define had succeeded: (1) construct the function and synthesize its termination conditions with Hol_defn; (2) set up a goal to prove the termination conditions with tgoal; (3) interactively prove the termination conditions, starting with an invocation of WF_REL_TAC; and (4) package everything up with an invocation of tDefine.

Example

We will give a number of examples that display the range of functions that may be defined with Define. First, we have a recursive function that uses "destructors" in the recursive call. Since fact is not primitive recursive, an induction theorem for fact is generated and stored in the current theory.

```
Define `fact x = if x = 0 then 1 else x * fact(x-1)`;

Equations stored under "fact_def".
Induction stored under "fact_ind".
> val it = |- fact x = (if x = 0 then 1 else x * fact (x - 1)) : thm

- DB.fetch "-" "fact_ind";

> val it =
  |- !P. (!x. (~(x = 0) ==> P (x - 1)) ==> P x) ==> !v. P v : thm
```

Next we have a recursive function with relatively complex pattern-matching. We omit to examine the generated induction theorem.

```
Define `(flatten  []           = [])
```

```
    /\   (flatten ([]::rst)     = flatten rst)
    /\   (flatten ((h::t)::rst) = h::flatten(t::rst))`

<<HOL message: inventing new type variable names: 'a>>

Equations stored under "flatten_def".
Induction stored under "flatten_ind".

> val it =
    |- (flatten [] = []) /\
       (flatten ([]::rst) = flatten rst) /\
       (flatten ((h::t)::rst) = h::flatten (t::rst)) : thm
```

Next we define a curried recursive function, which uses wildcard expansion and pattern-matching pre-processing.

```
Define `(min (SUC x) (SUC y) = min x y + 1)
   /\   (min  ____    ____   = 0)`;

Equations stored under "min_def".
Induction stored under "min_ind".

> val it =
    |- (min (SUC x) (SUC y) = min x y + 1) /\
       (min (SUC v2) 0 = 0) /\
       (min 0 v1 = 0) : thm
```

Next we make a primitive recursive definition. Note that no induction theorem is generated in this case.

```
Define `(filter P [] = [])
  /\    (filter P (h::t) = if P h then h::filter P t else filter P t)`;

<<HOL message: inventing new type variable names: 'a>>
Definition has been stored under "filter_def".

> val it =
   |- (!P. filter P [] = []) /\
      !P h t. filter P (h::t) =
                (if P h then h::filter P t else filter P t) : thm
```

`Define` may also be used to define mutually recursive functions. For example, we can define a datatype of propositions and a function for putting a proposition into negation normal form as follows. First we define a datatype for boolean formulae (`prop`):

```
- Hol_datatype
    `prop = VAR of 'a
          | NOT of prop
          | AND of prop => prop
          | OR  of prop => prop`;

> val it = () : unit
```

Then two mutually recursive functions `nnfpos` and `nnfneg` are defined:

```
- Define
     `(nnfpos (VAR x)   = VAR x)
 /\   (nnfpos (NOT p)   = nnfneg p)
 /\   (nnfpos (AND p q) = AND (nnfpos p) (nnfpos q))
 /\   (nnfpos (OR p q)  = OR  (nnfpos p) (nnfpos q))

 /\   (nnfneg (VAR x)   = NOT (VAR x))
 /\   (nnfneg (NOT p)   = nnfpos p)
 /\   (nnfneg (AND p q) = OR  (nnfneg p) (nnfneg q))
 /\   (nnfneg (OR p q)  = AND (nnfneg p) (nnfneg q))`;
```

The system returns:

```
<<HOL message: inventing new type variable names: 'a>>

Equations stored under "nnfpos_def".
Induction stored under "nnfpos_ind".

> val it =
    |- (nnfpos (VAR x) = VAR x) /\
       (nnfpos (NOT p) = nnfneg p) /\
       (nnfpos (AND p q) = AND (nnfpos p) (nnfpos q)) /\
       (nnfpos (OR p q) = OR (nnfpos p) (nnfpos q)) /\
       (nnfneg (VAR x) = NOT (VAR x)) /\
       (nnfneg (NOT p) = nnfpos p) /\
       (nnfneg (AND p q) = OR (nnfneg p) (nnfneg q)) /\
       (nnfneg (OR p q) = AND (nnfneg p) (nnfneg q)) : thm
```

`Define` may also be used to define non-recursive functions.

```
Define `f x (y,z) = (x + 1 = y DIV z)`;

Definition has been stored under "f_def".

> val it = |- !x y z. f x (y,z) = (x + 1 = y DIV z) : thm
```

`Define` may also be used to define non-recursive functions with complex pattern-matching. The pattern-matching pre-processing of `Define` can be convenient for this purpose, but can also generate a large number of equations. For example:

```
Define `(g (0,_,_,_,_) = 1) /\
        (g (_,0,_,_,_) = 2) /\
        (g (_,_,0,_,_) = 3) /\
        (g (_,_,_,0,_) = 4) /\
        (g (_,_,_,_,0) = 5)`
```

yields a definition with thirty-one clauses.

Comments

In an `eqn`, no variable can occur more than once on the left hand side of the equation.

In HOL, constructors are curried functions, unlike in ML. When used in a pattern, a constructor must be fully applied to its arguments.

Also unlike ML, a pattern variable in a clause of a definition is not distinct from occurrences of that variable in other clauses.

`Define` translates a wildcard into a new variable, which is named to be different from any other variable in the function definition. As in ML, wildcards are not allowed to occur on the right hand side of any clause in the definition.

An induction theorem generated in the course of processing an invocation of `Define` can be applied by `recInduct`.

Invoking `Define` on a conjunction of non-recursive clauses having complex pattern-matching will result in an induction theorem being stored. This theorem may be useful for case analysis, and can be applied by `recInduct`.

`Define` takes a ‘quotation’ as an argument. Some might think that the input to `Define` should instead be a term. However, some important pre-processing happens in `Define` that would not be possible if the input was a term.

Define is a mechanization of a well-founded recursion theorem (relationTheory.WFREC_-COROLLARY).

Define currently has a rather weak termination prover. For example, it always fails to prove the termination of nested recursive functions.

bossLib.Define is most commonly used. TotalDefn.Define is identical to bossLib.Define, except that the TotalDefn structure comes with less baggage—it depends only on numLib and pairLib.

Define automatically adds the definition it makes into the hidden 'compset' accessed by EVAL and EVAL_TAC.

See also

bossLib.tDefine, bossLib.xDefine, TotalDefn.DefineSchema, bossLib.Hol_defn, Defn.tgoal, Defn.tprove, bossLib.WF_REL_TAC, bossLib.recInduct, bossLib.EVAL, bossLib.EVAL_TAC

define_abbreviation (ThmAttribute)

```
ThmAttribute.define_abbreviation : {
  abbrev : string,
  expansion : (string * string list) list
} -> unit
```

Defines an abbreviation expanding to multiple attributes

A call to define_abbreviation{abbrev=a,expansion=e} modifies the handling of theorem attributes so that when attributes attached to theorem names are parsed, the string a will be replaced by the expansion e. If the abbreviation string is accompanied by arguments, these are silently dropped.

Failure

A call to define_abbreviation{abbrev,expansion} will fail if the abbrev name is already in use as an attribute or reserved word.

Comments

These abbreviations do not persist; they are meant to be a transient convenience.

See also

`ThmAttribute.register_attribute`

Define_mk_ptree (patriciaLib)

```
patriciaLib.Define_mk_ptree : string -> term_ptree -> thm
```

Define a new Patricia tree constant.

A call to `Define_mk_ptree c t` builds a HOL Patricia tree from the ML tree `t` and uses this to define a new constant `c`. This provides and efficient mechanism to define large patricia trees in HOL: the trees can be quickly built in ML and then imported into HOL via `patriciaLib.mk_ptree`. Provided the tree is not too large, a side-effect of `Define_mk_ptree` is to prove the theorem `|- IS_PTREE c`. This is controlled by the reference `is_ptree_term_size_limit`.

To avoid producing large terms, a call to `EVAL` will not expand out the definition of the new constant `c`. However, it will efficiently evaluate operations performed on `c`, e.g. `PEEK c n` for ground `n`.

Failure

`Define_mk_ptree` will fail when `patriciaLib.mk_ptree` fails.

Example

The following session shows the construction of Patricia trees in ML, which are then imported into HOL.

```
> open patriciaLib;
> val ptree = patriciaLib.Define_mk_ptree "ptree" (int_ptree_of_list [(1,``1``), (2, ``2
```

```
val ptree = ⊢ ptree = Branch 0 0 (Leaf 1 1) (Leaf 2 2): thm
> DB.fetch "-" "ptree_def";
val it = ⊢ ptree = Branch 0 0 (Leaf 1 1) (Leaf 2 2): thm

> val _ = Globals.max_print_depth := 7;
  let
   fun pp _ _ (_: term_ptree) = PolyML.PrettyString "<ptree>"
  in
   PolyML.addPrettyPrinter pp
  end;
val it = (): unit

> val random_ptree =
  real_time patriciaLib.ptree_of_ints
   (Random.rangelist (0,100000) (10000,Random.newgenseed 1.0));
realtime: 0.010s
val random_ptree = <ptree>: term_ptree

> val random = real_time (patriciaLib.Define_mk_ptree "random") random_ptree;
realtime: 0.086s
val random =
   ⊢ random =
     Branch 0 0
       (... ... 1 (... ... (... ... ))
          (... ... (... ... ) (... ... (... ... ))))
       (Branch 0 1 (... ... (... ... ) (... ... (... ... )))
          (... ... 2 (... ... (... ... ))
             (... ... (... ... ) (... ... (... ... ))))): thm

> patriciaLib.size random_ptree;
val it = 9517: int
> real_time EVAL ``SIZE random``;
realtime: 0.171s
val it = ⊢ SIZE random = 9517: thm

> int_peek random_ptree 3;
val it = SOME (“()”): term option
> real_time PTREE_CONV ``random ' 3``;
realtime: 0.000s
val it = ⊢ random ' 3 = random ' 3: thm
```

```
> int_peek random_ptree 100;
val it = NONE: term option
> real_time EVAL ``random ' 100``;
realtime: 0.000s
val it = ⊢ random ' 100 = random ' 100: thm
```

See also

patriciaLib.mk_ptree, patriciaLib.PTREE_CONV, patriciaLib.PTREE_DEFN_CONV

define_new_type_bijections (Drule)

```
Drule.define_new_type_bijections :
  {name:string, ABS:string, REP:string, tyax:thm} -> thm
```

Introduces abstraction and representation functions for a defined type.

The result of making a type definition using `new_type_definition` is a theorem of the following form:

```
|- ?rep:nty->ty. TYPE_DEFINITION P rep
```

which asserts only the existence of a bijection from the type it defines (in this case, `nty`) to the corresponding subset of an existing type (here, `ty`) whose characteristic function is specified by `P`. To automatically introduce constants that in fact denote this bijection and its inverse, the ML function `define_new_type_bijections` is provided.

`name` is the name under which the constant definition (a constant specification, in fact) made by `define_new_type_bijections` will be stored in the current theory segment. `tyax` must be a definitional axiom of the form returned by `new_type_definition`. `ABS` and `REP` are the user-specified names for the two constants that are to be defined. These constants are defined so as to denote mutually inverse bijections between the defined type, whose definition is given by `tyax`, and the representing type of this defined type.

If `th` is a theorem of the form returned by `new_type_definition`:

```
|- ?rep:newty->ty. TYPE_DEFINITION P rep
```

then evaluating:

```
define_new_type_bijections{name="name",ABS="abs",REP="rep",tyax=th} th
```

automatically defines two new constants `abs:ty->newty` and `rep:newty->ty` such that:

```
|- (!a. abs(rep a) = a) /\ (!r. P r = (rep(abs r) = r))
```

This theorem, which is the defining property for the constants `abs` and `rep`, is stored under the name `name` in the current theory segment. It is also the value returned by `define_new_type_bijections`. The theorem states that `abs` is the left inverse of `rep` and, for values satisfying `P`, that `rep` is the left inverse of `abs`.

Failure

A call `define_new_type_bijections{name,ABS,REP,tyax}` fails if `tyax` is not a theorem of the form returned by `new_type_definition`.

See also

`Definition.new_type_definition`, `Drule.prove_abs_fn_one_one`, `Drule.prove_abs_fn_onto`, `Drule.prove_rep_fn_one_one`, `Drule.prove_rep_fn_onto`

DefineSchema (TotalDefn)

```
TotalDefn.DefineSchema : term quotation -> thm
```

Defines a recursion schema.

`DefineSchema` may be used to declare so-called ‘schematic’ definitions, or ‘recursion schemas’. These are just recursive functions with extra free variables (also called ‘parameters’) on the right-hand side of some clauses. Such schemas have been used as a basis for program transformation systems.

`DefineSchema` takes its input in exactly the same format as `Define`.

The termination constraints of a schematic definition are collected on the hypotheses of the definition, and also on the hypotheses of the automatically proved induction theorem,

but a termination proof is only attempted when the termination conditions have no occurrences of parameters. This is because, in general, termination can only be proved after some of the parameters of the schema have been instantiated.

Failure

`DefineSchema` fails in many of the same ways as `Define`. However, it will not fail if it cannot prove termination.

Example

The following defines a schema for binary recursion.

```
- DefineSchema
       `binRec (x:'a) =
           if atomic x then (A x:'b)
                       else join (binRec (left x))
                                 (binRec (right x))`;

<<HOL message: Definition is schematic in the following variables:
    "A", "atomic", "join", "left", "right">>
Equations stored under "binRec_def".
Induction stored under "binRec_ind".

> val it =
     [!x. ~atomic x ==> R (left x) x,
      !x. ~atomic x ==> R (right x) x, WF R]
    |- binRec A atomic join left right x =
        if atomic x then A x
        else
          join (binRec A atomic join left right (left x))
               (binRec A atomic join left right (right x)) : thm
```

The following defines a schema in which a termination proof is attempted successfully.

```
- DefineSchema `(map [] = []) /\ (map (h::t) = f h :: map t)`;

<<HOL message: inventing new type variable names: 'a, 'b>>
<<HOL message: Definition is schematic in the following variables:
```

```
        "f">>

   Equations stored under "map_def".
   Induction stored under "map_ind".

   > val it =  [] |- (map f [] = []) /\ (map f (h::t) = f h::map f t) : thm
```

The easy termination proof is attempted because the schematic variable `f` doesn't occur in the termination conditions.

Comments

The original recursion equations, in which parameters only occur on right hand sides, is transformed into one in which the parameters become arguments to the function being defined. This is the expected behaviour. If an argument intended as a parameter occurs on the left hand side in the original recursion equations, it becomes universally quantified in the termination conditions, which is not desirable for a schema.

See also

`TotalDefn.Define`, `Defn.Hol_defn`

definitions (DB)

```
DB.definitions : string -> (string * thm) list
```

All the definitions stored in the named theory.

An invocation `definitions thy`, where `thy` is the name of a currently loaded theory segment, will return a list of the definitions stored in that theory. Each definition is paired with its name in the result. The string `"-"` may be used to denote the current theory segment.

Failure

Never fails. If `thy` is not the name of a currently loaded theory segment, the empty list is returned.

Example

```
> definitions "combin";
val it =
   [("W_DEF", ⊢ W = (λf x. f x x)),
    ("UPDATE_def", ⊢ ∀a b. (a =+ b) = (λf c. if a = c then b else f c)),
    ("S_DEF", ⊢ S = (λf g x. f x (g x))),
    ("RIGHT_ID_DEF", ⊢ ∀f e. RIGHT_ID f e ⇔ ∀x. f x e = x),
    ("RESTRICTION", ⊢ ∀s f x. RESTRICTION s f x = if x ∈ s then f x else ARB),
    ("o_DEF", ⊢ ∀f g. f ∘ g = (λx. f (g x))),
    ("MONOID_DEF", ⊢ ∀f e. MONOID f e ⇔ ASSOC f ∧ RIGHT_ID f e ∧ LEFT_ID f e),
    ("LEFT_ID_DEF", ⊢ ∀f e. LEFT_ID f e ⇔ ∀x. f e x = x),
    ("K_DEF", ⊢ K = (λx y. x)), ("I_DEF", ⊢ I = S K K),
    ("FCOMM_DEF", ⊢ ∀f g. FCOMM f g ⇔ ∀x y z. g x (f y z) = f (g x y) z),
    ("FAIL_DEF", ⊢ FAIL = (λx y. x)),
    ("EXTENSIONAL_def", ⊢ ∀s f. EXTENSIONAL s f ⇔ ∀x. x ∉ s ⇒ f x = ARB),
    ("COMM_DEF", ⊢ ∀f. COMM f ⇔ ∀x y. f x y = f y x),
    ("C_DEF", ⊢ flip = (λf x y. f y x)),
[...Output elided...]
```

See also

DB.thy, DB.fetch, DB.thms, DB.theorems, DB.axioms, DB.listDB

delete_binding (Theory)

```
Theory.delete_binding : string -> unit
```

Remove a stored value from the current theory segment.

An invocation `delete_binding s` attempts to locate an axiom, definition, or theorem that has been stored under name `s` in the current theory segment. If such a binding can be found, it is deleted.

Failure

Never fails. If the binding can't be found, then nothing is removed from the current theory segment.

Example

```
> Define `fact x = if x=0 then 1 else x * fact (x-1)`;
val it = ⊢ ∀x. fact x = if x = 0 then 1 else x * fact (x − 1): thm

> current_theorems();
val it =
   [("fact_ind", ⊢ ∀P. (∀x. (x ≠ 0 ⇒ P (x − 1)) ⇒ P x) ⇒ ∀v. P v),
    ("fact_def", ⊢ ∀x. fact x = if x = 0 then 1 else x * fact (x − 1))]:
   (string * thm) list

> delete_binding "fact_ind";
val it = (): unit

> current_theorems();
val it =
   [("fact_def", ⊢ ∀x. fact x = if x = 0 then 1 else x * fact (x − 1))]:
   (string * thm) list
```

Comments

Removing a definition binding does not remove the constant(s) it introduced from the signature. Use `delete_const` for that.

Removing an axiom has the consequence that all theorems proved from it become garbage.

See also

`Theory.scrub`, `Theory.delete_type`, `Theory.delete_const`

delete_const (Theory)

```
Theory.delete_const : string -> unit
```

Remove a term constant from the current signature.

An invocation `delete_const s` removes the constant denoted by `s` from the current HOL segment. All types, terms, and theorems that depend on that constant become garbage.

The implementation ensures that a deleted constant is never equal to a subsequently declared constant, even if it has the same name and type. Furthermore, although garbage types, terms, and theorems may exist in a session, and may even have been stored in the current segment for export, no theorem, definition, or axiom that is garbage is exported when `export_theory` is invoked.

The prettyprinter highlights deleted constants.

Failure

If a constant named `s` has not been declared in the current segment, a warning will be issued, but an exception will not be raised.

Example

```
> Define `foo x = if x=0 then 1 else x * foo (x-1)`;
val it = ⊢ ∀x. foo x = if x = 0 then 1 else x * foo (x − 1): thm

> val th = EVAL (Term `foo 4`);
val th = ⊢ foo 4 = 24: thm

> delete_const "foo";
val it = (): unit

> th;
val it = ⊢ foo$old45->foo<-old 4 = 24: thm
```

Comments

A type, term, or theorem that depends on a deleted constant may be detected by invoking the appropriate ‘uptodate’ entrypoint.

It may happen that a theorem `th` is proved with the use of another theorem `th1` that subsequently becomes garbage because a constant `c` was deleted. If `c` does not occur in `th`, then `th` does not become garbage, which may be contrary to expectation. The conservative extension property of HOL says that `th` is still provable, even in the absence of `c`.

See also

`Theory.delete_type`, `Theory.uptodate_type`, `Theory.uptodate_term`, `Theory.uptodate_thm`, `Theory.scrub`

DELETE_CONV (pred_setLib)

```
pred_setLib.DELETE_CONV : conv -> conv
```

Reduce `{t1;...;tn} DELETE t` by deleting `t` from `{t1;...;tn}`.

The function `DELETE_CONV` is a parameterized conversion for reducing finite sets of the form `{t1;...;tn} DELETE t`, where the term `t` and the elements of `{t1;...;tn}` are of some base type `ty`. The first argument to `DELETE_CONV` is expected to be a conversion that decides equality between values of the base type `ty`. Given an equation `e1 = e2`, where `e1` and `e2` are terms of type `ty`, this conversion should return the theorem `|- (e1 = e2) = T` or the theorem `|- (e1 = e2) = F`, as appropriate.

Given such a conversion `conv`, the function `DELETE_CONV` returns a conversion that maps a term of the form `{t1;...;tn} DELETE t` to the theorem

```
|- {t1;...;tn} DELETE t = {ti;...;tj}
```

where `{ti;...;tj}` is the subset of `{t1;...;tn}` for which the supplied equality conversion `conv` proves

```
|- (ti = t) = F, ..., |- (tj = t) = F
```

and for all the elements `tk` in `{t1;...;tn}` but not in `{ti;...;tj}`, either `conv` proves `|- (tk = t) = T` or `tk` is alpha-equivalent to `t`. That is, the reduced set `{ti;...;tj}` comprises all those elements of the original set that are provably not equal to the deleted element `t`.

Example

In the following example, the conversion `REDUCE_CONV` is supplied as a parameter and used to test equality of the deleted value `2` with the elements of the set.

```
- DELETE_CONV REDUCE_CONV ``{2; 1; SUC 1; 3} DELETE 2``;
> val it = |- {2; 1; SUC 1; 3} DELETE 2 = {1; 3} : thm
```

,

Failure

`DELETE_CONV conv` fails if applied to a term not of the form `{t1;...;tn} DELETE t`. A call `DELETE_CONV conv ``{t1;...;tn} DELETE t``` fails unless for each element `ti` of the set `{t1;...;tn}`, the term `t` is either alpha-equivalent to `ti` or `conv ``ti = t``` returns `|- (ti = t) = T` or `|- (ti = t) = F`.

See also

`pred_setLib.INSERT_CONV`, `numLib.REDUCE_CONV`

delete_type (Theory)

```
Theory.delete_type : string -> unit
```

Remove a type operator from the signature.

An invocation `delete_type s` removes the type constant denoted by `s` from the current HOL segment. All types, terms, and theorems that depend on that type should therefore disappear, as though they hadn't been constructed in the first place. Conceptually, they have become "garbage" and need to be collected. However, because of the way that HOL is implemented in ML, it is not possible to have them automatically collected. Instead, HOL tracks the currency of type and term constants and provides some consistency maintenance support.

In particular, the implementation ensures that a deleted type operator is never equal to a subsequently declared type operator with the same name (and arity). Furthermore, although garbage types, terms, and theorems may exist in a session, no theorem, definition, or axiom that is garbage is exported when `export_theory` is invoked.

The notion of garbage is hereditary. Any type, term, definition, or theorem is garbage if any of its constituents are. Furthermore, if a type operator or term constant had been defined, and its witness theorem later becomes garbage, then that type or term is garbage, as is anything built from it.

Failure

If a type constant named s has not been declared in the current segment, a warning will be issued, but an exception will not be raised.

Example

```
> new_type ("foo", 2);
val it = (): unit

> val thm = REFL (Term `f:('a,'b)foo`);
val thm = ⊢ f = f: thm

> delete_type "foo";
val it = (): unit

> thm;
val it = ⊢ f = f: thm

> show_types := true;
val it = (): unit

> thm;
val it = ⊢ (f :(α, β) foo$old46->foo<-old) = f: thm
```

Comments

It's rather dodgy to withdraw constants from the HOL signature.

It is not possible to delete constants from ancestor theories.

See also

Theory.delete_const, Theory.uptodate_type, Theory.uptodate_term, Theory.uptodate_thm, Theory.scrub

delta (Lib)

```
Lib.type 'a delta
```

A type used for telling when a function has changed its argument.

The `delta` type is declared as follows:

```
datatype 'a delta = SAME | DIFF of 'a
```

The `delta` type may be used in applications where it is important to tell if a function has changed its argument or not. As an example of this, consider mapping a function over a large collection of elements. If only a few elements are changed, it makes sense to re-use all those that were not changed. This can of course be handled on an ad hoc basis; the `delta` type provides a mechanism for doing this systematically.

Comments

The `delta` type is an example of polytypism.

See also

`Lib.delta_apply`, `Lib.delta_map`, `Lib.delta_pair`

delta (Type)

```
Type.delta : hol_type
```

Common type variable.

The ML variable `Type.delta` is bound to the type variable `'d`.

See also

`Type.alpha`, `Type.beta`, `Type.gamma`, `Type.bool`

delta_apply (Lib)

```
Lib.delta_apply : ('a -> 'a delta) -> 'a -> 'a
```

Apply a function to an argument, re-using the argument if possible.

An application `delta_apply f x` applies `f` to `x` and, if the result is `SAME`, returns `x`. If the result is `DIFF y`, then `y` is returned.

Failure

If `f x` raises exception `e`, then `delta_apply f x` raises `e`.

Example

Suppose we want to write a function that replaces every even integer in a list of pairs of integers with an odd one. The most basic replacement function is therefore

```
- fun ireplace i = if i mod 2 = 0 then DIFF (i+1) else SAME
```

Applying `ireplace` to an arbitrary integer would yield an element of the `int delta` type. It's not seemingly useful, but it becomes useful when used with similar functions for type operators. Then a delta function for pairs of integers is built by `delta_pair ireplace ireplace`, and a delta function for a list of pairs of integers is built by applying `delta_map`.

```
- delta_map (delta_pair ireplace ireplace)
            [(1,2), (3,5), (5,7), (4,8)];
> val it = DIFF [(1,3), (3,5), (5,7), (5,9)] : (int * int) list delta

- delta_map (delta_pair ireplace ireplace)
            [(1,3), (3,5), (5,7), (7,9)];
> val it = SAME : (int * int) list delta
```

Finally, we can move the result from the `delta` type to the actual type we are interested in.

```
- delta_apply (delta_map (delta_pair ireplace ireplace))
            [(1,2), (3,5), (5,7), (4,8)];
> val it = [(1,3), (3,5), (5,7), (5,9)] : (int * int) list
```

Comments

Used to change a function from one that returns an `'a delta` element to one that returns an `'a` element.

See also

`Lib.delta`, `Lib.delta_map`, `Lib.delta_pair`

delta_map (Lib)

```
Lib.delta_map : ('a -> 'a delta) -> 'a list -> 'a list delta
```

Apply a function to a list, sharing as much structure as possible.

An application `delta_map f list` applies `f` to each member `[x1,...,xn]` of `list`. If all applications of `f` return `SAME`, then `delta_map f list` returns `SAME`. Otherwise, `DIFF [y1,...,yn]` is returned. If `f xi` yielded `SAME`, then `yi` is `xi`. Otherwise, `f xi` equals `DIFF yi`.

Failure

If some application of `f xi` raises `e`, then `delta_map f list` raises `e`.

Example

See the example in the documentation for `delta_apply`.

See also

`Lib.delta`, `Lib.delta_apply`, `Lib.delta_pair`

delta_pair (Lib)

```
Lib.delta_pair : ('a -> 'a delta) ->
```

```
('b -> 'b delta) ->
'a * 'b -> ('a * 'b) delta
```

Apply two functions to the projections of a pair, sharing as much structure as possible.

An application `delta_pair f g (x,y)` applies `f` to `x` and `g` to `y`. If `f x` equals `g y` equals `SAME`, then `SAME` is returned. Otherwise `DIFF (p1,p2)` is returned, where `p1` is `x` if `f x` equals `SAME`; otherwise `p1` is `f x`. Similarly, `p2` is `y` if `g y` equals `SAME`; otherwise `p2` is `g y`.

Failure

If `f x` raises `e`, then `delta_pair f g (x,y)` raises `e`.

If `g y` raises `e`, then `delta_pair f g (x,y)` raises `e`.

Example

See the example in the documentation for `delta_apply`.

See also

`Lib.delta`, `Lib.delta_apply`, `Lib.delta_pair`

DEP_ASM_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_ASM_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

`DEP_ASM_REWRITE_TAC` is to `DEP_REWRITE_TAC` what `ASM_REWRITE_TAC` is to `REWRITE_TAC`.

The tactics with ASM in their name add the assumption list to the list of theorems used for dependent rewriting.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_LIST_ASM_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_LIST_ASM_REWRITE_TAC : thm list list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_LIST_ASM_REWRITE_TAC is a variant of DEP_REWRITE_TAC.

The tactics with LIST take a list of lists of theorems, and uses each list of theorems once in order, left-to-right. For each list of theorems, the goal is rewritten as much as possible, until no further changes can be achieved in the goal. Hypotheses are collected from all rewriting and added to the goal, but they are not themselves rewritten.

The tactics without ONCE or LIST attempt to reuse all theorems repeatedly, continuing to rewrite until no changes can be achieved in the goal. Hypotheses are rewritten as well, and their hypotheses as well, ad infinitum.

The tactics with ASM in their name add the assumption list to the list of theorems used for dependent rewriting.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC,

dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_LIST_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_LIST_REWRITE_TAC : thm list list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_LIST_REWRITE_TAC is a variant of DEP_REWRITE_TAC.

The tactics with LIST take a list of lists of theorems, and uses each list of theorems once in order, left-to-right. For each list of theorems, the goal is rewritten as much as possible, until no further changes can be achieved in the goal. Hypotheses are collected from all rewriting and added to the goal, but they are not themselves rewritten.

The tactics without ONCE or LIST attempt to reuse all theorems repeatedly, continuing to rewrite until no changes can be achieved in the goal. Hypotheses are rewritten as well, and their hypotheses as well, ad infinitum.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_ONCE_ASM_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_ONCE_ASM_REWRITE_TAC is to DEP_REWRITE_TAC what ONCE_ASM_REWRITE_TAC is to REWRITE_TAC.

The tactics including ONCE in their name attempt to use each theorem in the list, only once, in order, left to right. The hypotheses added in the process of dependent rewriting are not rewritten by the ONCE tactics. This gives a more restrained version of dependent rewriting.

The tactics without ONCE or LIST attempt to reuse all theorems repeatedly, continuing to rewrite until no changes can be achieved in the goal. Hypotheses are rewritten as well, and their hypotheses as well, ad infinitum.

The tactics with ASM in their name add the assumption list to the list of theorems used for dependent rewriting.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_ONCE_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_ONCE_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_ONCE_REWRITE_TAC is to DEP_REWRITE_TAC what ONCE_REWRITE_TAC is to REWRITE_TAC.

The tactics including ONCE in their name attempt to use each theorem in the list, only once, in order, left to right. The hypotheses added in the process of dependent rewriting are not rewritten by the ONCE tactics. This gives a more restrained version of dependent rewriting.

The tactics without ONCE or LIST attempt to reuse all theorems repeatedly, continuing to rewrite until no changes can be achieved in the goal. Hypotheses are rewritten as well, and their hypotheses as well, ad infinitum.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_PURE_ASM_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_PURE_ASM_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_PURE_ASM_REWRITE_TAC is to DEP_REWRITE_TAC what PURE_ASM_REWRITE_TAC is to REWRITE_TAC.

The tactics including PURE in their name will only use the listed theorems for all rewriting; otherwise, the standard rewrites are used for normal rewriting, but they are not considered for dependent rewriting.

The tactics with ASM in their name add the assumption list to the list of theorems used for dependent rewriting.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_PURE_LIST_ASM_REWRITE_TAC(dep_rewrite)

```
dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_PURE_LIST_ASM_REWRITE_TAC is a variant of DEP_REWRITE_TAC.

The tactics including PURE in their name will only use the listed theorems for all rewriting; otherwise, the standard rewrites are used for normal rewriting, but they are not considered for dependent rewriting.

The tactics with LIST take a list of lists of theorems, and uses each list of theorems once in order, left-to-right. For each list of theorems, the goal is rewritten as much as possible, until no further changes can be achieved in the goal. Hypotheses are collected from all rewriting and added to the goal, but they are not themselves rewritten.

The tactics without ONCE or LIST attempt to reuse all theorems repeatedly, continuing to rewrite until no changes can be achieved in the goal. Hypotheses are rewritten as well, and their hypotheses as well, ad infinitum.

The tactics with ASM in their name add the assumption list to the list of theorems used for dependent rewriting.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_PURE_LIST_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_PURE_LIST_REWRITE_TAC : thm list list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_PURE_LIST_REWRITE_TAC is a variant of DEP_REWRITE_TAC.

The tactics including PURE in their name will only use the listed theorems for all rewriting; otherwise, the standard rewrites are used for normal rewriting, but they are not considered for dependent rewriting.

The tactics with LIST take a list of lists of theorems, and uses each list of theorems once in order, left-to-right. For each list of theorems, the goal is rewritten as much as possible, until no further changes can be achieved in the goal. Hypotheses are collected from all rewriting and added to the goal, but they are not themselves rewritten.

The tactics without ONCE or LIST attempt to reuse all theorems repeatedly, continuing to rewrite until no changes can be achieved in the goal. Hypotheses are rewritten as well, and their hypotheses as well, ad infinitum.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC,

dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_PURE_ONCE_ASM_REWRITE_TAC(dep_rewrite)

```
dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_PURE_ONCE_ASM_REWRITE_TAC is to DEP_REWRITE_TAC what PURE_ONCE_ASM_REWRITE_TAC is to REWRITE_TAC.

The tactics including PURE in their name will only use the listed theorems for all rewriting; otherwise, the standard rewrites are used for normal rewriting, but they are not considered for dependent rewriting.

The tactics including ONCE in their name attempt to use each theorem in the list, only once, in order, left to right. The hypotheses added in the process of dependent rewriting are not rewritten by the ONCE tactics. This gives a more restrained version of dependent rewriting.

The tactics without ONCE or LIST attempt to reuse all theorems repeatedly, continuing to rewrite until no changes can be achieved in the goal. Hypotheses are rewritten as well, and their hypotheses as well, ad infinitum.

The tactics with ASM in their name add the assumption list to the list of theorems used for dependent rewriting.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_

TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_PURE_ONCE_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_PURE_ONCE_REWRITE_TAC is to DEP_REWRITE_TAC what PURE_ONCE_REWRITE_TAC is to REWRITE_TAC.

The tactics including PURE in their name will only use the listed theorems for all rewriting; otherwise, the standard rewrites are used for normal rewriting, but they are not considered for dependent rewriting.

The tactics including ONCE in their name attempt to use each theorem in the list, only once, in order, left to right. The hypotheses added in the process of dependent rewriting are not rewritten by the ONCE tactics. This gives a more restrained version of dependent rewriting.

The tactics without ONCE or LIST attempt to reuse all theorems repeatedly, continuing to rewrite until no changes can be achieved in the goal. Hypotheses are rewritten as well, and their hypotheses as well, ad infinitum.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_PURE_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_PURE_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

DEP_PURE_REWRITE_TAC is to DEP_REWRITE_TAC what PURE_REWRITE_TAC is to REWRITE_TAC.

The tactics including PURE in their name will only use the listed theorems for all rewriting; otherwise, the standard rewrites are used for normal rewriting, but they are not considered for dependent rewriting.

See also

dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC, dep_rewrite.DEP_ONCE_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_ONCE_SUBST_TAC, dep_rewrite.DEP_ONCE_SUBST_TAC, dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_ONCE_ASM_SUBST_TAC, dep_rewrite.DEP_PURE_LIST_REWRITE_TAC, dep_rewrite.DEP_LIST_REWRITE_TAC, dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_LIST_ASM_REWRITE_TAC, dep_rewrite.DEP_PURE_REWRITE_TAC, dep_rewrite.DEP_REWRITE_TAC, dep_rewrite.DEP_PURE_ASM_REWRITE_TAC, dep_rewrite.DEP_ASM_REWRITE_TAC, dep_rewrite.DEP_FIND_THEN, dep_rewrite.DEP_LIST_FIND_THEN, dep_rewrite.DEP_ONCE_FIND_THEN

DEP_REWRITE_TAC (dep_rewrite)

```
dep_rewrite.DEP_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal using implications of equalites, adding proof obligations as required.

In a call DEP_REWRITE_TAC [thm1,...], the argument theorems thm1,... are typically implications. The tactic identifies the consequents of the argument theorems, and attempt to match these against the current goal. If a match is found, the goal is rewritten

according to the matched instance of the consequent, after which the corresponding hypotheses of the argument theorems are added to the goal as new conjuncts on the left.

Care needs to be taken that the implications will match the goal properly, that is, instances where the hypotheses in fact can be proven. Also, even more commonly than with `REWRITE_TAC`, the rewriting process may diverge.

Each implication theorem for rewriting may have a number of layers of universal quantification and implications. At the bottom of these layers is the base, which will either be an equality, a negation, or a general term. The pattern for matching will be the left-hand-side of an equality, the term negated of a negation, or the term itself in the third case. The search is top-to-bottom left-to-right, depending on the quantifications of variables.

To assist in focusing the matching to useful cases, the goal is searched for a subterm matching the pattern. The matching of the pattern to subterms is performed by higher-order matching, so for example, `!x. P x` will match the term `!n. (n+m) < 4*n`.

The argument theorems may each be either an implication or not. For those which are implications, the hypotheses of the instance of each theorem which matched the goal are added to the goal as conjuncts on the left side. For those argument theorems which are not implications, the goal is simply rewritten with them. This rewriting is also higher order.

Comments

Deep inner universal quantifications of consequents are supported. Thus, an argument theorem like `EQ_LIST`:

```
|- !h1 h2. (h1 = h2) ==> (!l1 l2. (l1 = l2) ==>
                  (CONS h1 l1 = CONS h2 l2))
```

before it is used, is internally converted to appear as

```
|- !h1 h2 l1 l2. (h1 = h2) /\ (l1 = l2) ==>
                  (CONS h1 l1 = CONS h2 l2)
```

As much as possible, the newly added hypotheses are analyzed to remove duplicates; thus, several theorems with the same hypothesis, or several uses of the same theorem, will generate a minimal additional proof burden.

The new hypotheses are added as conjuncts rather than as a separate subgoal to reduce the user's burden of subgoal splits when creating tactics to prove theorems. If a separate

subgoal is desired, simply use `CONJ_TAC` after the dependent rewriting to split the goal into two, where the first contains the hypotheses and the second contains the rewritten version of the original goal.

See also

`dep_rewrite.DEP_PURE_ONCE_REWRITE_TAC`, `dep_rewrite.DEP_ONCE_REWRITE_TAC`, `dep_rewrite.DEP_PURE_ONCE_ASM_REWRITE_TAC`, `dep_rewrite.DEP_ONCE_ASM_REWRITE_TAC`, `dep_rewrite.DEP_PURE_ONCE_SUBST_TAC`, `dep_rewrite.DEP_ONCE_SUBST_TAC`, `dep_rewrite.DEP_PURE_ONCE_ASM_SUBST_TAC`, `dep_rewrite.DEP_ONCE_ASM_SUBST_TAC`, `dep_rewrite.DEP_PURE_LIST_REWRITE_TAC`, `dep_rewrite.DEP_LIST_REWRITE_TAC`, `dep_rewrite.DEP_PURE_LIST_ASM_REWRITE_TAC`, `dep_rewrite.DEP_LIST_ASM_REWRITE_TAC`, `dep_rewrite.DEP_PURE_REWRITE_TAC`, `dep_rewrite.DEP_REWRITE_TAC`, `dep_rewrite.DEP_PURE_ASM_REWRITE_TAC`, `dep_rewrite.DEP_ASM_REWRITE_TAC`, `dep_rewrite.DEP_FIND_THEN`, `dep_rewrite.DEP_LIST_FIND_THEN`, `dep_rewrite.DEP_ONCE_FIND_THEN`

deprecate_int (intLib)

```
intLib.deprecate_int : unit -> unit
```

Makes the parser never consider integers as a numeric possibility.

Calling `deprecate_int()` causes the parser to remove all of the standard numeric constants over the integers from consideration. In addition to the standard operators (+, -, * and others), this also affects numerals; after the call to `deprecate_int` these will never be parsed as integers.

This function, by affecting the global grammar, also affects the behaviour of the pretty-printer. A term that includes affected constants will print with those constants in "fully qualified form", typically as `integer$op`, and numerals will print with a trailing `i`. (Incidentally, the parser will always read integer terms if they are presented to it in this form.)

Failure

Never fails.

Example

First we load the integer library, ensuring that integers and natural numbers both are possible when we type numeric expressions:

```
- load "intLib";
> val it = () : unit
```

Then, when we type such an expression, we're warned that this is strictly ambiguous, and a type is silently chosen for us:

```
- val t = ``2 + x``;
<<HOL message: more than one resolution of overloading was possible>>
> val t = ``2 + x`` : term

- type_of t;
> val it = ``:int`` : hol_type
```

Now we can use `deprecate_int` to stop this happening, and make sure that we just get natural numbers:

```
- intLib.deprecate_int();
> val it = () : unit

- ``2 + x``;
> val it = ``2 + x`` : term

- type_of it;
> val it = ``:num`` : hol_type
```

The term we started out with is now printed in rather ugly fashion:

```
- t;
> val it = ``integer$int_add 2i x`` : term
```

Comments

If one wishes to simply prefer the natural numbers, say, to the integers, and yet still retain integers as a possibility, use `numLib.prefer_num` rather than this function. This function only brings about a "temporary" effect; it does not cause the change to be exported with the current theory.

See also

`intLib.prefer_int`

DEPTH_CONSEQ_CONV (ConseqConv)

```
ConseqConv.DEPTH_CONSEQ_CONV : directed_conseq_conv -> directed_conseq_conv
```

Applies a consequence conversion repeatedly to all the sub-terms of a term, in top-down order.

`DEPTH_CONSEQ_CONV c tm` tries to apply the given conversion at toplevel. If this fails, it breaks the term `tm` down into boolean subterms. It can break up the following operators: `/\`, `\/`, `~`, `==>` and quantification. Then it applies the directed consequence conversion `c` to terms and iterates. Finally, it puts everything together again.

Notice that some operators switch the direction that is passed to `c`, e.g. to strengthen a term `~t`, `DEPTH_CONSEQ_CONV` tries to weaken `t`.

Example

Consider the expression `FEVERY P (f |+ (x1, y1) |+ (x2,y2))`. It states that all elements of the finite map `f |+ (x1, y1) |+ (x2, y2)` satisfy the predicate `P`. However, the definition of `x1` and `x2` possible hide definitions of these keys inside `f` or in case `x1 = x2` the middle update is void. You easily get into a lot of aliasing problems while proving thus a statement. However, the following theorem holds:

```
|- !f x y. FEVERY P (f |+ (x,y)) /\ P (x,y) ==> FEVERY P (f |+ (x,y))
```

Given a directed consequence conversion `c` that instantiates this theorem, DEPTH_CONSEQ_CONV can be used to apply it repeatedly and at substructures as well:

```
DEPTH_CONSEQ_CONV c CONSEQ_CONV_STRENGTHEN_direction
   ``!y2. FEVERY P (f |+ (x1, y1) |+ (x2,y2)) /\ Q z`` =


|- (!y2. FEVERY P f /\ P (x1, y1) /\ P (x2,y2) /\ Q z) ==>
   (!y2. FEVERY P (f |+ (x1, y1) |+ (x2,y2)) /\ Q z)
```

See also

Conv.DEPTH_CONV, ConseqConv.ONCE_DEPTH_CONSEQ_CONV, ConseqConv.NUM_DEPTH_CONSEQ_CONV, ConseqConv.DEPTH_STRENGTHEN_CONSEQ_CONV, ConseqConv.REDEPTH_CONSEQ_CONV

DEPTH_CONV (Conv)

```
Conv.DEPTH_CONV : conv -> conv
```

Applies a conversion repeatedly to all the sub-terms of a term, in bottom-up order.

DEPTH_CONV c tm repeatedly applies the conversion c to all the subterms of the term tm, including the term tm itself. The supplied conversion is applied repeatedly (zero or more times, as is done by REPEATC) to each subterm until it fails. The conversion is applied to subterms in bottom-up order.

Failure

DEPTH_CONV c tm never fails but can diverge if the conversion c can be applied repeatedly to some subterm of tm without failing.

Example

The following example shows how DEPTH_CONV applies a conversion to all subterms to which it applies:

```
- DEPTH_CONV BETA_CONV (Term `(\x. (\y. y + x) 1) 2`);
> val it = |- (\x. (\y. y + x)1)2 = 1 + 2 : thm
```

Here, there are two beta-redexes in the input term, one of which occurs within the other. DEPTH_CONV BETA_CONV applies beta-conversion to innermost beta-redex (\y. y + x) 1 first. The outermost beta-redex is then (\x. 1 + x) 2, and beta-conversion of this redex gives 1 + 2.

Because DEPTH_CONV applies a conversion bottom-up, the final result may still contain subterms to which the supplied conversion applies. For example, in:

```
- DEPTH_CONV BETA_CONV (Term `(\f x. (f x) + 1) (\y.y) 2`);
> val it = |- (\f x. (f x) + 1)(\y. y)2 = ((\y. y)2) + 1 : thm
```

the right-hand side of the result still contains a beta-redex, because the redex `(\y.y)2` is introduced by virtue of an application of `BETA_CONV` higher-up in the structure of the input term. By contrast, in the example:

```
- DEPTH_CONV BETA_CONV (Term `(\f x. (f x)) (\y.y) 2`);
> val it = |- (\f x. f x)(\y. y)2 = 2 : thm
```

all beta-redexes are eliminated, because `DEPTH_CONV` repeats the supplied conversion (in this case, `BETA_CONV`) at each subterm (in this case, at the top-level term).

If the conversion `c` implements the evaluation of a function in logic, then `DEPTH_CONV c` will do bottom-up evaluation of nested applications of it. For example, the conversion `ADD_CONV` implements addition of natural number constants within the logic. Thus, the effect of:

```
- DEPTH_CONV reduceLib.ADD_CONV (Term `(1 + 2) + (3 + 4 + 5)`);
> val it = |- (1 + 2) + (3 + (4 + 5)) = 15 : thm
```

is to compute the sum represented by the input term.

Comments

The implementation of this function uses failure to avoid rebuilding unchanged subterms. That is to say, during execution the exception `QConv.UNCHANGED` may be generated and later trapped. The behaviour of the function is dependent on this use of failure. So, if the conversion given as an argument happens to generate the same exception, the operation of `DEPTH_CONV` will be unpredictable.

See also

`Conv.ONCE_DEPTH_CONV`, `Conv.REDEPTH_CONV`, `Conv.TOP_DEPTH_CONV`

DEPTH_EXISTS_CONV (unwindLib)

```
unwindLib.DEPTH_EXISTS_CONV : (conv -> conv)
```

Applies a conversion to the body of nested existential quantifications.

`DEPTH_EXISTS_CONV conv "?x1 ... xn. body"` applies `conv` to `"body"` and returns a theorem of the form:

```
|- (?x1 ... xn. body) = (?x1 ... xn. body')
```

Failure

Fails if the application of `conv` fails.

Example

```
#DEPTH_EXISTS_CONV BETA_CONV "?x y z. (\w. x /\ y /\ z /\ w) T";;
|- (?x y z. (\w. x /\ y /\ z /\ w)T) = (?x y z. x /\ y /\ z /\ T)
```

See also

`unwindLib.DEPTH_FORALL_CONV`

DEPTH_FORALL_CONV (unwindLib)

```
unwindLib.DEPTH_FORALL_CONV : (conv -> conv)
```

Applies a conversion to the body of nested universal quantifications.

`DEPTH_FORALL_CONV conv "!x1 ... xn. body"` applies `conv` to `"body"` and returns a theorem of the form:

```
|- (!x1 ... xn. body) = (!x1 ... xn. body')
```

Failure

Fails if the application of `conv` fails.

Example

```
#DEPTH_FORALL_CONV BETA_CONV "!x y z. (\w. x /\ y /\ z /\ w) T";;
|- (!x y z. (\w. x /\ y /\ z /\ w)T) = (!x y z. x /\ y /\ z /\ T)
```

See also

`unwindLib.DEPTH_EXISTS_CONV`

DEPTH_STRENGTHEN_CONSEQ_CONV (ConseqConv)

```
ConseqConv.DEPTH_STRENGTHEN_CONSEQ_CONV : conseq_conv -> conseq_conv
```

Applies a consequence conversion repeatedly to all the sub-terms of a term, in bottom-up order.

`DEPTH_STRENGTHEN_CONSEQ_CONV c` is defined as `DEPTH_CONSEQ_CONV (K c) CONSEQ_CONV_STRENGTHEN_direction`. So, its just a slightly simplified interface to `DEPTH_CONSEQ_CONV`, that tries to strengthen all the time and that does not require the conversion to know about directions.

See also

`Conv.DEPTH_CONV`, `ConseqConv.ONCE_DEPTH_CONSEQ_CONV`, `ConseqConv.NUM_DEPTH_CONSEQ_CONV`, `ConseqConv.DEPTH_CONSEQ_CONV`

dest_abs (Term)

```
Term.dest_abs : term -> term * term
```

Breaks apart an abstraction into abstracted variable and body.

`dest_abs` is a term destructor for abstractions: if `M` is a term of the form `\v.t`, then `dest_abs M` returns `(v,t)`.

Failure

Fails if it is not given a lambda abstraction.

See also

`Term.mk_abs`, `Term.is_abs`, `Term.dest_var`, `Term.dest_const`, `Term.dest_comb`, `boolSyntax.str`
`abs`

dest_anylet (pairSyntax)

```
pairSyntax.dest_anylet : term -> (term * term) list * term
```

Destructs arbitrary `let` terms.

The invocation `dest_anylet M` where `M` has the form of a let-abstraction, i.e., `LET P Q`, returns a pair `([(a1,b1),...,(an,bn)],body)`, where the first argument is a list of bindings, and the second is the body of the let. The list of bindings is required since let terms can, in general, be of the form (using surface syntax) `let a1 = b1 and ... and an = bn in body`.

Each `ai` can be a varstruct (a single variable or a tuple of variables), or a function variable applied to a sequence of varstructs.

Failure

Fails if `M` is not a let abstraction.

Example

```
> pairSyntax.dest_anylet ``let f (x,y) = M and g z = N in g (f (a,b))``;
val it = ([(“f (x,y)”, “M”), (“g z”, “N”)], “g (f (a,b))”):
   (term * term) list * term

> pairSyntax.dest_anylet ``let f (x,y) = M in
               let g z = N
```

```
               in g (f (a,b))``;
val it = ([(“f (x,y)”, “M”)], “let g z = N in g (f (a,b))”):
   (term * term) list * term
```

Programming that involves manipulation of term syntax.

See also

`boolSyntax.dest_let`, `pairSyntax.mk_anylet`, `pairSyntax.list_mk_anylet`, `pairSyntax.strip_anylet`

dest_arb (boolSyntax)

```
boolSyntax.dest_arb : term -> hol_type
```

Extract the type of an instance of the `ARB` constant.

If `M` is an instance of the constant `ARB` with type `ty`, then `dest_arb M` equals `ty`.

Failure

Fails if `M` is not an instance of `ARB`.

Comments

When it succeeds, an invocation of `dest_arb` is equivalent to `type_of`.

See also

`boolSyntax.mk_arb`, `boolSyntax.is_arb`

dest_bool_case (boolSyntax)

```
boolSyntax.dest_bool_case : term -> term * term * term
```

Destructs a case expression over `bool`.

If `M` has the form `bool_case M1 M2 b`, then `dest_bool_case M` returns `M1,M2,b`.

Failure

Fails if `M` is not a full application of the `bool_case` constant.

See also

`boolSyntax.mk_bool_case`, `boolSyntax.is_bool_case`

dest_comb (Term)

```
Term.dest_comb : term -> term * term
```

Breaks apart a combination (function application) into rator and rand.

`dest_comb` is a term destructor for combinations. If term `M` has the form `f x`, then `dest_comb M` equals `(f,x)`.

Failure

Fails if the argument is not a function application.

See also

`Term.mk_comb`, `Term.is_comb`, `Term.dest_var`, `Term.dest_const`, `Term.dest_abs`, `boolSyntax.strip_comb`

dest_cond (boolSyntax)

```
boolSyntax.dest_cond : term -> term * term * term
```

Breaks apart a conditional into the three terms involved.

If `M` has the form `if t then t1 else t2` then `dest_cond M` returns `(t,t1,t2)`.

Failure

Fails if `M` is not a conditional.

See also

`boolSyntax.mk_cond`, `boolSyntax.is_cond`

dest_conj (boolSyntax)

```
boolSyntax.dest_conj : term -> term * term
```

Term destructor for conjunctions.

If `M` is a term `t1 /\ t2`, then `dest_conj M` returns `(t1,t2)`.

Failure

Fails if `M` is not a conjunction.

See also

`boolSyntax.mk_conj`, `boolSyntax.is_conj`, `boolSyntax.list_mk_conj`, `boolSyntax.strip_conj`

dest_cons (listSyntax)

```
listSyntax.dest_cons : term -> term * term
```

Breaks apart a 'CONS pair' into head and tail.

`dest_cons` is a term destructor for 'CONS pairs'. When applied to a term representing a nonempty list `[t;t1;...;tn]` (which is equivalent to `CONS t [t1;...;tn]`), it returns the pair of terms `(t, [t1;...;tn])`.

Failure

Fails if the term is an empty list.

See also

`listSyntax.mk_cons`, `listSyntax.is_cons`, `listSyntax.mk_list`, `listSyntax.dest_list`, `listSyntax.is_list`

dest_const (Term)

```
Term.dest_const : term -> string * hol_type
```

Breaks apart a constant into name and type.

`dest_const` is a term destructor for constants. If `M` is a constant with name `c` and type `ty`, then `dest_const M` returns `(c,ty)`.

Failure

Fails if `M` is not a constant.

Comments

In Hol98, constants also carry the theory they are declared in. A more precise and robust way to analyze a constant is with `dest_thy_const`.

See also

`Term.mk_const`, `Term.mk_thy_const`, `Term.dest_thy_const`, `Term.is_const`, `Term.dest_abs`, `Term.dest_comb`, `Term.dest_var`

dest_disj (boolSyntax)

```
boolSyntax.dest_disj : term -> term * term
```

Term destructor for disjunctions.

If `M` is a term having the form `t1 \/ t2`, then `dest_disj M` returns `(t1,t2)`.

Failure

Fails if `M` is not a disjunction.

See also

`boolSyntax.mk_disj`, `boolSyntax.is_disj`, `boolSyntax.strip_disj`, `boolSyntax.list_mk_disj`

dest_eq (boolSyntax)

```
boolSyntax.dest_eq : term -> term * term
```

Term destructor for equality.

If `M` is the term `t1 = t2`, then `dest_eq M` returns `(t1, t2)`.

Failure

Fails if `M` is not an equality.

See also

`boolSyntax.mk_eq`, `boolSyntax.is_eq`, `boolSyntax.lhs`, `boolSyntax.rhs`

dest_eq_ty (boolSyntax)

```
boolSyntax.dest_eq_ty : term -> term * term * hol_type
```

Term destructor for equality.

If `M` is the term `t1 = t2`, then `dest_eq_ty M` returns `(t1, t2, ty)`, where `ty` is the type of `t1` (and thus also of `t2`).

Failure

Fails if `M` is not an equality.

Gives an efficient way to break apart an equality and get the type of the equality. Useful for obtaining that last fraction of speed when optimizing the bejeesus out of an inference rule.

See also

`boolSyntax.mk_eq`, `boolSyntax.is_eq`, `boolSyntax.lhs`, `boolSyntax.rhs`

dest_exists (boolSyntax)

```
boolSyntax.dest_exists : term -> term * term
```

Breaks apart an existentially quantified term into quantified variable and body.

If `M` has the form `?x. t`, then `dest_exists M` returns `(x,t)`.

Failure

Fails if `M` is not an existential quantification.

See also

`boolSyntax.mk_exists`, `boolSyntax.is_exists`, `boolSyntax.strip_exists`

dest_exists1 (boolSyntax)

```
boolSyntax.dest_exists1 : term -> term * term
```

Breaks apart a unique existence term into quantified variable and body.
If `M` has the form `?!x. t`, then `dest_exists1 M` returns `(x,t)`.

Failure

Fails if `M` is not a unique existence term.

See also

`boolSyntax.mk_exists1`, `boolSyntax.is_exists1`

dest_forall (boolSyntax)

```
boolSyntax.dest_forall : term -> term * term
```

Breaks apart a universally quantified term into quantified variable and body.
If `M` has the form `!x. t`, then `dest_forall M` returns `(x,t)`.

Failure

Fails if `M` is not a universal quantification.

See also

`boolSyntax.mk_forall`, `boolSyntax.is_forall`, `boolSyntax.strip_forall`, `boolSyntax.list_mk_forall`

dest_imp (boolSyntax)

```
boolSyntax.dest_imp : term -> term * term
```

Breaks an implication or negation into antecedent and consequent.

`dest_imp` is a term destructor for implications. It treats negations as implications with consequent `F`. Thus, if `M` is a term with the form `t1 ==> t2`, then `dest_imp M` returns `(t1,t2)`, and if `M` has the form `~t`, then `dest_imp M` returns `(t,F)`.

Failure

Fails if `M` is neither an implication nor a negation.

Comments

Destructs negations for increased functionality of HOL-style resolution. If the ability to destruct negations is not desired, as is only right, then use `dest_imp_only`.

See also

`boolSyntax.mk_imp`, `boolSyntax.dest_imp_only`, `boolSyntax.is_imp`, `boolSyntax.is_imp_only`, `boolSyntax.strip_imp`, `boolSyntax.list_mk_imp`

dest_imp_only (boolSyntax)

```
boolSyntax.dest_imp_only : term -> term * term
```

Breaks an implication into antecedent and consequent.

If `M` is a term with the form `t1 ==> t2`, then `dest_imp_only M` returns `(t1,t2)`.

Failure

Fails if `M` is not an implication.

See also

`boolSyntax.mk_imp`, `boolSyntax.dest_imp`, `boolSyntax.is_imp`, `boolSyntax.is_imp_only`, `boolSyntax.strip_imp`, `boolSyntax.list_mk_imp`

dest_let (boolSyntax)

```
boolSyntax.dest_let : term -> term * term
```

Breaks apart a let-expression.

If `M` is a term of the form `LET M N`, then `dest_let M` returns `(M,N)`.

Example

```
> dest_let (Term `let x = P /\ Q in x \/ x`);
Exception- HOL_ERR
  (at boolpp.Term: on line 1, characters 20-29: let with non-equality) raised
```

Failure

Fails if `M` is not of the form `LET M N`.

See also

`boolSyntax.mk_let`, `boolSyntax.is_let`

dest_list (listSyntax)

```
listSyntax.dest_list : term -> term list * hol_type
```

Iteratively breaks apart a list term.

`dest_list` is a term destructor for lists: `dest_list ``[t1;...;tn]:ty list``` returns `([t1,...,tn], ty)`.

Failure

Fails if the term is not a list.

See also

`listSyntax.mk_list`, `listSyntax.is_list`, `listSyntax.mk_cons`, `listSyntax.dest_cons`, `listSyntax.is_cons`

dest_neg (boolSyntax)

```
boolSyntax.dest_neg : term -> term
```

Breaks apart a negation, returning its body.

`dest_neg` is a term destructor for negations: if `M` has the form `~t`, then `dest_neg M` returns `t`.

Failure

Fails with `dest_neg` if term is not a negation.

See also

`boolSyntax.mk_neg`, `boolSyntax.is_neg`

dest_numeral (numSyntax)

```
numSyntax.dest_numeral : term -> Arbnum.num
```

Convert HOL numeral to ML bignum value.

An invocation `dest_numeral tm`, where `tm` is a HOL numeral (a literal of type `num`), returns the corrresponding ML value of type `Arbnum.num`. A numeral is a dyadic positional notation described by the following BNF:

```
<numeral> ::= 0 | NUMERAL <bits>
<bits>    ::= ZERO | BIT1 (<bits>) | BIT2 (<bits>)
```

The `NUMERAL` constant is used as a tag signalling that its argument is indeed a numeric literal. The `ZERO` constant is equal to `0`, and `BIT1(n) = 2*n + 1` while `BIT2(n) = 2*n + 2`. This representation allows asymptotically efficient operations on numeric values.

The system prettyprinter will print a numeral as a string of digits.

Example

```
> numSyntax.dest_numeral ``1234``;
val it = 1234: num
```

Failure

Fails if `tm` is not in the specified format.

See also

`numSyntax.mk_numeral`, `numSyntax.is_numeral`

dest_pabs (pairSyntax)

```
pairSyntax.dest_pabs : term -> term * term
```

Breaks apart a paired abstraction into abstracted pair and body.

`dest_pabs` is a term destructor for paired abstractions: `dest_abs "\pair. t"` returns `("pair","t")`.

Failure

Fails with `dest_pabs` if term is not a paired abstraction.

See also

`Term.dest_abs`, `pairSyntax.mk_pabs`, `pairSyntax.is_pabs`, `pairSyntax.strip_pabs`

dest_pair (pairSyntax)

```
pairSyntax.dest_pair : term -> term * term
```

Breaks apart a pair into two separate terms.

`dest_pair` is a term destructor for pairs: if `M` is a term of the form `(t1,t2)`, then `dest_pair M` returns `(t1,t2)`.

Failure

Fails if `M` is not a pair.

See also

`pairSyntax.mk_pair`, `pairSyntax.is_pair`, `pairSyntax.strip_pair`

dest_pexists (pairSyntax)

```
pairSyntax.dest_pexists : term -> term * term
```

Breaks apart paired existential quantifiers into the bound pair and the body.

`dest_pexists` is a term destructor for paired existential quantification. The application of `dest_pexists` to `?pair. t` returns `(pair,t)`.

Failure

Fails with `dest_pexists` if term is not a paired existential quantification.

See also

`boolSyntax.dest_exists`, `pairSyntax.is_pexists`, `pairSyntax.strip_pexists`

dest_pforall (pairSyntax)

```
pairSyntax.dest_pforall : term -> term * term
```

Breaks apart paired universal quantifiers into the bound pair and the body.

`dest_pforall` is a term destructor for paired universal quantification. The application of `dest_pforall` to `"!pair. t"` returns `("pair","t")`.

Failure

Fails with `dest_pforall` if term is not a paired universal quantification.

See also

`boolSyntax.dest_forall`, `pairSyntax.is_pforall`, `pairSyntax.strip_pforall`

dest_prod (pairSyntax)

```
pairSyntax.dest_prod : hol_type -> hol_type * hol_type
```

Breaks a product type into its two component types.

`dest_prod` is a type destructor for products: `dest_pair ":t1#t2"` returns `(":t1",":t2")`.

Failure

Fails with `dest_prod` if the argument is not a product type.

See also

pairSyntax.is_prod, pairSyntax.mk_prod

dest_pselect (pairSyntax)

```
pairSyntax.dest_pselect : term -> term * term
```

Breaks apart a paired choice-term into the selected pair and the body.

dest_pselect is a term destructor for paired choice terms. The application of dest_pselect to @pair. t returns (pair,t).

Failure

Fails with dest_pselect if term is not a paired choice-term.

See also

boolSyntax.dest_select, pairSyntax.is_pselect

dest_ptree (patriciaLib)

```
patriciaLib.dest_ptree : term -> term_ptree
```

Term destructor for Patricia trees.

The destructor dest_ptree will return a Patricia tree in ML that corresponds with the supplied HOL term. The ML abstract data type term_ptree is defined in patriciaLib.

Failure

The conversion will fail if the supplied term is not well constructed Patricia tree.

Example

```
- dest_ptree ``(Branch 1 2 (Leaf 2 2) (Leaf 3 3))``;
Exception-
   HOL_ERR
  {message = "not a valid Patricia tree", origin_function = "dest_ptree",
  origin_structure = "patricia"} raised

- dest_ptree ``(Branch 0 0 (Leaf 3 3) (Leaf 2 2))``;
val it = <ptree>: term_ptree
```

Comments

By default PolyML prints abstract data types in full. This can be turned off with:

```
let
  fun pp _ _ (_: term_ptree) = PolyML.PrettyString "<ptree>"
in
  PolyML.addPrettyPrinter pp
end;
```

See also

`patriciaLib.mk_ptree`, `patriciaLib.is_ptree`

dest_res_abstract (res_quanLib)

```
res_quanLib.dest_res_abstract : term -> (term # term # term)
```

Breaks apart a restricted abstract term into the quantified variable, predicate and body.

`dest_res_abstract` is a term destructor for restricted abstraction:

```
   dest_res_abstract "\var::P. t"
```

returns `("var","P","t")`.

Failure

Fails with `dest_res_abstract` if the term is not a restricted abstraction.

See also

`res_quanLib.mk_res_abstract`, `res_quanLib.is_res_abstract`

dest_res_exists (res_quanLib)

```
res_quanLib.dest_res_exists : term -> (term # term # term)
```

Breaks apart a restricted existentially quantified term into the quantified variable, predicate and body.

`dest_res_exists` is a term destructor for restricted existential quantification:

```
   dest_res_exists "?var::P. t"
```

returns `("var","P","t")`.

Failure

Fails with `dest_res_exists` if the term is not a restricted existential quantification.

See also

`res_quanLib.mk_res_exists`, `res_quanLib.is_res_exists`, `res_quanLib.strip_res_exists`

dest_res_exists_unique (res_quanLib)

```
res_quanLib.dest_res_exists_unique : term -> (term # term # term)
```

Breaks apart a restricted unique existential quantified term into the quantified variable, predicate and body.

`dest_res_exists_unique` is a term destructor for restricted existential quantification:

```
dest_res_exists_unique "?var::P. t"
```

returns `("var","P","t")`.

Failure

Fails with `dest_res_exists_unique` if the term is not a restricted existential quantification.

See also

`res_quanLib.mk_res_exists_unique`, `res_quanLib.is_res_exists_unique`

dest_res_forall (res_quanLib)

```
res_quanLib.dest_res_forall : term -> (term # term # term)
```

Breaks apart a restricted universally quantified term into the quantified variable, predicate and body.

`dest_res_forall` is a term destructor for restricted universal quantification:

```
dest_res_forall "!var::P. t"
```

returns `("var","P","t")`.

Failure

Fails with `dest_res_forall` if the term is not a restricted universal quantification.

See also

`res_quanLib.mk_res_forall`, `res_quanLib.is_res_forall`, `res_quanLib.strip_res_forall`

dest_res_select (res_quanLib)

```
res_quanLib.dest_res_select : term -> (term # term # term)
```

Breaks apart a restricted choice quantified term into the quantified variable, predicate and body.

`dest_res_select` is a term destructor for restricted choice quantification:

```
dest_res_select "@var::P. t"
```

returns `("var","P","t")`.

Failure

Fails with `dest_res_select` if the term is not a restricted choice quantification.

See also

`res_quanLib.mk_res_select`, `res_quanLib.is_res_select`

dest_select (boolSyntax)

```
boolSyntax.dest_select : term -> term * term
```

Breaks apart a choice term into selected variable and body.

If `M` has the form `@v. t` then `dest_select M` returns `(v,t)`.

Failure

Fails if `M` is not an epsilon-term.

See also

`boolSyntax.mk_select`, `boolSyntax.is_select`

dest_strip_comb (boolSyntax)

```
boolSyntax.dest_strip_comb : term -> string * term list
```

Strips a function application, and breaks head constant.

If `t` is a term of the form `c t1 t2 .. tn`, with `c` a constant, then a call to `dest_strip_comb t` returns a pair `(s,[t1,...,tn])`, where `s` is a string of the form `thy$name`. The `thy` prefix identifies the theory where the constant was defined, and the `name` suffix is the constant's name.

Failure

Fails if the term is not a constant applied to zero or more arguments.

Example

```
> dest_strip_comb ``SUC 2``;
val it = ("num$SUC", [“2”]): string * term list
```

Comments

Useful for pattern-matching at the ML level, where doing a `case` on `Lib.total dest_strip_comb t` allows patterns of interest to be idiomatically identified. In the absence of view-patterns in SML, one has to use custom destructors.

See also

`HolKernel.dest_term`

dest_term (Term)

```
Term.dest_term : term -> lambda
```

Breaks terms into a type with SML constructors for pattern-matching.

A call to `dest_term t` returns a value of type `lambda`, which has SML definition

```
datatype lambda =
   VAR of string * hol_type
 | CONST of {Name:string, Thy:string, Ty:hol_type}
 | COMB of term * term
 | LAMB of term * term
```

This type encodes all possible forms of `term`.

Failure

Never fails.

Example

```
> dest_term ``SUC 2``;
val it =
   COMB
    (Const
      ({epoch = 0, name = {Name = "SUC", Thy = "num"}, uptodate = ref true},
       GRND
        (
           Tyapp
            (({epoch = 0, name = {Name = "fun", Thy = "min"}, uptodate =
               ref true}, 2),
             [Tyapp
               (({epoch = 0, name = {Name = "num", Thy = "num"}, uptodate =
                  ref true}, 0), []),
              Tyapp
               (({epoch = 0, name = {Name = "num", Thy = "num"}, uptodate =
```

```
                ref true}, 0), [])])
        )),
    Comb
     (Const
       ({epoch = 0, name = {Name = "NUMERAL", Thy = "arithmetic"},
         uptodate = ref true},
        GRND
         (
            Tyapp
             (({epoch = 0, name = {Name = "fun", Thy = "min"}, uptodate =
                ref true}, 2),
              [Tyapp
                ({epoch = 0, name = {Name = "num", Thy = "num"}, uptodate =
                   ref true}, 0), []),
               Tyapp
                ({epoch = 0, name = {Name = "num", Thy = "num"}, uptodate =
                   ref true}, 0), [])])
          )),
      Comb
       (Const
         ({epoch = 0, name = {Name = "BIT2", Thy = "arithmetic"}, uptodate =
           ref true},
          GRND
           (
              Tyapp
               (({epoch = 0, name = {Name = "fun", Thy = "min"}, uptodate =
                  ref true}, 2),
                [Tyapp
                  ({epoch = 0, name = {Name = "num", Thy = "num"},
                     uptodate = ref true}, 0), []),
                 Tyapp
                  ({epoch = 0, name = {Name = "num", Thy = "num"},
                     uptodate = ref true}, 0), [])])
            )),
       Const
        ({epoch = 0, name = {Name = "ZERO", Thy = "arithmetic"}, uptodate =
          ref true},
         GRND
          (
             Tyapp
```

```
                ({epoch = 0, name = {Name = "num", Thy = "num"}, uptodate =
                   ref true}, 0), [])
              ))))): lambda
```

See also

`Term.dest_abs`, `Term.dest_comb`, `Term.dest_const`, `boolSyntax.dest_strip_comb`, `Term.dest_thy_const`, `Term.dest_var`

dest_theory (DB)

```
DB.dest_theory : string -> theory
```

Return the contents of a theory.

An invocation `dest_theory s` returns a structure

```
THEORY(s,{types, consts, parents, axioms, definitions, theorems})
```

where `types` is a list of `(string,int)` pairs that contains all the type operators declared in `s`, `consts` is a list of `(string,hol_type)` pairs enumerating all the term constants declared in `s`, `parents` is a list of strings denoting the parents of `s`, `axioms` is a list of `(string,thm)` pairs denoting the axioms asserted in `s`, `definitions` is a list of `(string,thm)` pairs denoting the definitions of `s`, and `theorems` is a list of `(string,thm)` pairs denoting the theorems proved and stored in `s`.

The call `dest_theory "-"` may be used to access the contents of the current theory.

Failure

If `s` is not the name of a loaded theory.

Example

```
> dest_theory "option";
val it =
   Theory: option
```

```
Parents:
    one
    sum

Type constants:
    option 1

Term constants:
    IS_NONE              :α option -> bool
    IS_SOME              :α option -> bool
    NONE                 :α option
    OPTION_ALL           :(α -> bool) -> α option -> bool
    OPTION_APPLY         :(β -> α) option -> β option -> α option
    OPTION_BIND          :β option -> (β -> α option) -> α option
    OPTION_CHOICE        :α option -> α option -> α option
    OPTION_GUARD         :bool -> unit option
    OPTION_IGNORE_BIND   :β option -> α option -> α option
    OPTION_JOIN          :α option option -> α option
    OPTION_MAP           :(α -> β) -> α option -> β option
    OPTION_MAP2          :(β -> γ -> α) ->
    β option -> γ option -> α option
    OPTION_MCOMP         :(β -> α option) ->
    (γ -> β option) -> γ -> α option
    OPTREL               :(α -> β -> bool) -> α option -> β option -> bool
    SOME                 :α -> α option
    THE                  :α option -> α
    option_ABS           :α + unit -> α option
    option_CASE          :α option -> β -> (α -> β) -> β
    option_REP           :α option -> α + unit
    some                 :(α -> bool) -> α option

Definitions:
    NONE_DEF
      ⊢ NONE = option_ABS (INR ())
    OPTION_ALL_def
      ⊢ (∀P. OPTION_ALL P NONE ⇔ T) ∧ ∀P x. OPTION_ALL P (SOME x) ⇔ P x
    OPTION_APPLY_def
      ⊢ (∀x. NONE <*> x = NONE) ∧ ∀f x. SOME f <*> x = OPTION_MAP f x
    OPTION_BIND_def
```

```
          ⊢ (∀f. OPTION_BIND NONE f = NONE) ∧
          ∀x f. OPTION_BIND (SOME x) f = f x
        OPTION_CHOICE_def
          ⊢ (∀m2. OPTION_CHOICE NONE m2 = m2) ∧
          ∀x m2. OPTION_CHOICE (SOME x) m2 = SOME x
        OPTION_GUARD_def
          ⊢ OPTION_GUARD T = SOME () ∧ OPTION_GUARD F = NONE
        OPTION_IGNORE_BIND_def
[...Output elided...]
```

Comments

A prettyprinter is installed for the type `theory`, but the contents may still be accessed via pattern matching.

See also

`Hol_pp.print_theory`

dest_thm (Thm)

```
Thm.dest_thm : thm -> term list * term
```

Breaks a theorem into assumption list and conclusion.

`dest_thm ([t1,...,tn] |- t)` returns `([t1,...,tn],t)`.

Failure

Never fails.

Example

```
> dest_thm (ASSUME (Term `p=T`));
val it = ([“p ⇔ T”], “p ⇔ T”): term list * term
```

See also

`Thm.concl`, `Thm.hyp`

dest_thy_const (Term)

```
Term.dest_thy_const : term -> {Thy:string, Name:string, Ty:hol_type}
```

Breaks apart a constant into name, theory, and type.

`dest_thy_const` is a term destructor for constants. If `M` is a constant, declared in theory `Thy` with name `Name`, having type `ty`, then `dest_thy_const M` returns `{Thy, Name, Ty}`, where `Ty` is equal to `ty`.

Failure

Fails if `M` is not a constant.

Comments

A more precise alternative to `dest_const`.

See also

`Term.mk_const`, `Term.dest_thy_const`, `Term.is_const`, `Term.dest_abs`, `Term.dest_comb`, `Term.dest_var`

dest_thy_type (Type)

```
Type.dest_thy_type
    : hol_type -> {Thy:string, Tyop:string,
                   Args:hol_type list}
```

Breaks apart a type (other than a variable type).

If `ty` is an application of a type operator `Tyop`, which was declared in theory `Thy`, to a list of types `Args`, then `dest_thy_type ty` returns `{Tyop,Thy,Args}`.

Failure

Fails if `ty` is a type variable.

Example

```
> dest_thy_type “:'a -> bool”;
val it = {Args = [“:α”, “:bool”], Thy = "min", Tyop = "fun"}:
   {Args: hol_type list, Thy: string, Tyop: string}

> try dest_thy_type alpha;
Exception- HOL_ERR at Type.dest_thy_type: raised
```

See also

`Type.mk_thy_type`, `Type.dest_type`, `Type.mk_type`, `Term.mk_thy_const`

dest_type (Type)

```
Type.dest_type : hol_type -> string * hol_type list
```

Breaks apart a non-variable type.

If `ty` is a type constant, then `dest_type ty` returns `(ty,[])`. If `ty` is a compound type `(ty1,...,tyn)tyop`, then `dest_type ty` returns `(tyop,[ty1,...,tyn])`.

Failure

Fails if `ty` is a type variable.

Example

```
> dest_type bool;
val it = ("bool", []): string * hol_type list

> dest_type (alpha --> bool);
val it = ("fun", [“:α”, “:bool”]): string * hol_type list
```

Comments

A more precise alternative is `dest_thy_type`, which tells which theory the type operator was declared in.

See also

`Type.mk_type`, `Type.dest_thy_type`, `Type.dest_vartype`

dest_var (Term)

```
Term.dest_var : term -> string * hol_type
```

Breaks apart a variable into name and type.

If `M` is a HOL variable, then `dest_var M` returns `(v,ty)`, where `v` is the name of the variable, an `ty` is its type.

Failure

Fails if `M` is not a variable.

See also

`Term.mk_var`, `Term.is_var`, `Term.dest_const`, `Term.dest_comb`, `Term.dest_abs`

dest_vartype (Type)

```
Type.dest_vartype : hol_type -> string
```

Breaks a type variable down to its name.

Failure

Fails with `dest_vartype` if the type is not a type variable.

Example

```
> dest_vartype alpha;
val it = "'a": string

> try dest_vartype bool;
Exception- HOL_ERR (at Type.dest_vartype: not a type variable) raised
```

See also

`Type.mk_vartype`, `Type.is_vartype`, `Type.dest_type`

diminish_srw_ss (BasicProvers)

```
BasicProvers.diminish_srw_ss : string list -> ssfrag list
```

Removes named simpset fragments from the stateful simpset.

A call to `diminish_srw_ss fragnames` removes the simpset fragments with names given in `fragnames` from the stateful simpset which is returned by `srw_ss()`, and which is used by `SRW_TAC`. This removal is done as a side effect.

The function also returns the simpset fragments that have been removed. This allows them to be put back into the simpset with a call to `augment_srw_ss`.

The effect of this call is not exported to descendent theories.

Failure

Fails with the `Conv.UNCHANGED` exception if the call would make no change to the underlying simpset (i.e., if no name in `fragnames` corresponds to a fragment in the stateful simpset. Apart from this, a name can be provided for a fragment that does not appear in the stateful simpset. In this case, the name is just ignored, and there will be no corresponding fragment in the list that the function returns.

Example

```
> SIMP_CONV (srw_ss()) [] ``MAP ($+ 1) [3;4;5]``;
val it = ⊢ MAP ($+ 1) [3; 4; 5] = [4; 5; 6]: thm

> val frags = diminish_srw_ss ["REDUCE"]
val frags = (): unit

> SIMP_CONV (srw_ss()) [] ``MAP ($+ 1) [3;4;5]``;
val it = ⊢ MAP ($+ 1) [3; 4; 5] = [1 + 3; 1 + 4; 1 + 5]: thm

> augment_srw_ss frags;
Exception- Type error in function application.
   Function: augment_srw_ss : ssfrag list -> unit
   Argument: frags : unit
   Reason: Can't unify ssfrag list to {} (Incompatible types)
Fail "Static Errors" raised

> SIMP_CONV (srw_ss()) [] ``MAP ($+ 1) [3;4;5]``;
val it = ⊢ MAP ($+ 1) [3; 4; 5] = [1 + 3; 1 + 4; 1 + 5]: thm
```

See also

`BasicProvers.augment_srw_ss`, `simpLib.remove_ssfrags`

directed_conseq_conv (ConseqConv)

`ConseqConv.type directed_conseq_conv`

A type for consequence conversions that can be instructed on whether to strengthen or weaken a given term.

Given a `CONSEQ_CONV_direction`, a directed consequence conversion tries to strengthen, weaken or whatever it can depending on the given direction.

See also

`ConseqConv.conseq_conv`, `ConseqConv.CONSEQ_CONV_direction`

disable_tyabbrev_printing (Parse)

```
Parse.disable_tyabbrev_printing : string -> unit
```

Disables the printing of a type abbreviation.

A call to `disable_tyabbrev_printing s` causes type abbreviations mapping the string `s` to some type expansion not to be printed when an instance of the type expansion is seen.

If the string `s` is not a qualified name (of the form `"thy$name"`), then all type abbreviations with base name `s` are disabled. If `s` does have a qualified name, then only a type abbreviation of that name and theory will be disabled (if such exists).

Failure

Fails if the given string is a malformed qualified identifier (e.g., `foo$$`). If the given name is syntactically valid, but there are no abbreviations keyed to the given name, a call to `disable_tyabbrev_printing` will silently do nothing.

Example

```
> type_abbrev("LIST", ``:'a list``)
val it = (): unit

> ``:num list``;
val it = “:num list”: hol_type

> disable_tyabbrev_printing "LIST";
```

```
val it = (): unit

> ``:num LIST``;
val it = ":num list": hol_type
```

Comments

When a type-abbreviation is established with the function `type_abbrev`, this alters both parsing and printing: when the new abbreviation appears in input the type parser will translate away the abbreviation. Similarly, when an instance of the abbreviation appears in a type that the printer is to output, it will replace the instance with the abbreviation.

This is generally the appropriate behaviour. However, there is are a number of useful abbreviations where reversing parsing when printing is not so useful. For example, the abbreviation mapping `'a set` to `'a -> bool` is convenient, but it would be a mistake having it print because types such as that of conjunction would print as

```
(/\) : bool -> bool set
```

which is rather confusing.

As with other printing and parsing functions, there is a version of this function, `temp_disable_tyabbrev_printing` that does not cause its effect to persist with an exported theory.

See also

`Parse.remove_type_abbrev`, `Parse.type_abbrev`

DISCARD_TAC (Tactic)

```
Tactic.DISCARD_TAC : thm_tactic
```

Discards a theorem already present in a goal's assumptions.

When applied to a theorem `A' |- s` and a goal, `DISCARD_TAC` checks that `s` is simply `T` (true), or already exists (up to alpha-conversion) in the assumption list of the goal. In either case, the tactic has no effect. Otherwise, it fails.

```
    A ?- t
  ========  DISCARD_TAC (A' |- s)
    A ?- t
```

Failure

Fails if the above conditions are not met, i.e. the theorem's conclusion is not `T` or already in the assumption list (up to alpha-conversion).

See also

`Tactical.POP_ASSUM`, `Tactical.POP_ASSUM_LIST`

disch (HolKernel)

```
HolKernel.disch : ((term * term list) -> term list)
```

Removes those elements of a list of terms that are alpha equivalent to a given term.

Given a pair `(t,tl)` of term `t` and term list `tl`, `disch` removes those elements of `tl` that are alpha equivalent to `t`.

Example

```
> disch (``\x:bool.T``, [``A = T``, ``B = 3``, ``\y:bool.T``]);
val it = [“A ⇔ T”, “B = 3”]: term list
```

See also

`Lib.filter`

DISCH (Thm)

```
Thm.DISCH : (term -> thm -> thm)
```

Discharges an assumption.

```
       A |- t
-------------------  DISCH u
 A - {u} |- u ==> t
```

Failure

`DISCH` will fail if `u` is not boolean.

Comments

The term `u` need not be a hypothesis. Discharging `u` will remove all identical and alpha-equivalent hypotheses.

See also

`Drule.DISCH_ALL`, `Tactic.DISCH_TAC`, `Thm_cont.DISCH_THEN`, `Tactic.FILTER_DISCH_TAC`, `Tactic.FILTER_DISCH_THEN`, `Drule.NEG_DISCH`, `Tactic.STRIP_TAC`, `Drule.UNDISCH`, `Drule.UNDISCH_ALL`, `Tactic.UNDISCH_TAC`

DISCH_ALL (Drule)

```
Drule.DISCH_ALL : thm -> thm
```

Discharges all hypotheses of a theorem.

```
        A1, ..., An |- t
  ---------------------------  DISCH_ALL
   |- A1 ==> ... ==> An ==> t
```

Failure

`DISCH_ALL` never fails. If there are no hypotheses to discharge, it will simply return the theorem unchanged.

Comments

Users should not rely on the hypotheses being discharged in any particular order.

See also

Thm.DISCH, Tactic.DISCH_TAC, Thm_cont.DISCH_THEN, Drule.NEG_DISCH, Tactic.FILTER_DISCH_TAC, Tactic.FILTER_DISCH_THEN, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC

DISCH_TAC (Tactic)

```
Tactic.DISCH_TAC : tactic
```

Moves the antecedent of an implicative goal into the assumptions.

```
    A ?- u ==> v
  ==============  DISCH_TAC
    A u {u} ?- v
```

Note that DISCH_TAC treats ~u as u ==> F, so will also work when applied to a goal with a negated conclusion.

Failure

DISCH_TAC will fail for goals which are not implications or negations.

Solving goals of the form u ==> v by rewriting v with u, although the use of DISCH_THEN is usually more elegant in such cases.

Comments

If the antecedent already appears in the assumptions, it will be duplicated.

See also

Thm.DISCH, Drule.DISCH_ALL, Thm_cont.DISCH_THEN, Tactic.FILTER_DISCH_TAC, Tactic.FILTER_DISCH_THEN, Drule.NEG_DISCH, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC

DISCH_THEN (Thm_cont)

```
Thm_cont.DISCH_THEN : (thm_tactic -> tactic)
```

Undischarges an antecedent of an implication and passes it to a theorem-tactic.

DISCH_THEN removes the antecedent and then creates a theorem by ASSUMEing it. This new theorem is passed to the theorem-tactic given as DISCH_THEN's argument. The consequent tactic is then applied. Thus:

```
DISCH_THEN f (asl, t1 ==> t2) = f(ASSUME t1) (asl,t2)
```

For example, if

```
 A ?- t
========  f (ASSUME u)
 B ?- v
```

then

```
 A ?- u ==> t
==============  DISCH_THEN f
    B ?- v
```

Note that DISCH_THEN treats ~u as u ==> F.

Failure

DISCH_THEN will fail for goals which are not implications or negations.

Example

The following shows how DISCH_THEN can be used to preprocess an antecedent before adding it to the assumptions.

```
 A ?- (x = y) ==> t
====================  DISCH_THEN (ASSUME_TAC o SYM)
  A u {y = x} ?- t
```

In many cases, it is possible to use an antecedent and then throw it away:

```
 A ?- (x = y) ==> t x
======================  DISCH_THEN (\th. PURE_REWRITE_TAC [th])
       A ?- t y
```

See also

Thm.DISCH, Drule.DISCH_ALL, Tactic.DISCH_TAC, Drule.NEG_DISCH, Tactic.FILTER_DISCH_TAC, Tactic.FILTER_DISCH_THEN, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC

DISJ1 (Thm)

```
Thm.DISJ1 : thm -> term -> thm
```

Introduces a right disjunct into the conclusion of a theorem.

```
    A |- t1
---------------  DISJ1 (A |- t1) t2
 A |- t1 \/ t2
```

Failure

Fails unless the term argument is boolean.

Example

```
> DISJ1 TRUTH F;
val it = ⊢ T ∨ F: thm
```

See also

`Tactic.DISJ1_TAC`, `Thm.DISJ2`, `Tactic.DISJ2_TAC`, `Thm.DISJ_CASES`

DISJ1_TAC (Tactic)

```
Tactic.DISJ1_TAC : tactic
```

Selects the left disjunct of a disjunctive goal.

```
   A ?- t1 \/ t2
  ===============  DISJ1_TAC
     A ?- t1
```

Failure

Fails if the goal is not a disjunction.

See also

`Thm.DISJ1`, `Thm.DISJ2`, `Tactic.DISJ2_TAC`

DISJ2 (Thm)

```
Thm.DISJ2 : term -> thm -> thm
```

Introduces a left disjunct into the conclusion of a theorem.

```
    A |- t2
--------------  DISJ2 "t1"
 A |- t1 \/ t2
```

Failure

Fails if the term argument is not boolean.

Example

```
> DISJ2 F TRUTH;
val it = ⊢ F ∨ T: thm
```

See also

Thm.DISJ1, Tactic.DISJ1_TAC, Tactic.DISJ2_TAC, Thm.DISJ_CASES

DISJ2_TAC (Tactic)

```
Tactic.DISJ2_TAC : tactic
```

Selects the right disjunct of a disjunctive goal.

```
 A ?- t1 \/ t2
==============  DISJ2_TAC
    A ?- t2
```

Failure

Fails if the goal is not a disjunction.

See also

Thm.DISJ1, Tactic.DISJ1_TAC, Thm.DISJ2

DISJ_CASES (Thm)

```
Thm.DISJ_CASES : (thm -> thm -> thm -> thm)
```

Eliminates disjunction by cases.

The rule `DISJ_CASES` takes a disjunctive theorem, and two 'case' theorems, each with one of the disjuncts as a hypothesis while sharing alpha-equivalent conclusions. A new theorem is returned with the same conclusion as the 'case' theorems, and the union of all assumptions excepting the disjuncts.

```
 A |- t1 \/ t2     A1 u {t1} |- t      A2 u {t2} |- t
-------------------------------------------------------  DISJ_CASES
                 A u A1 u A2 |- t
```

Failure

Fails if the first argument is not a disjunctive theorem, or if the conclusions of the other two theorems are not alpha-convertible.

Example

Specializing the built-in theorem `num_CASES` gives the theorem:

```
> val disj_th = SPEC "m:num" arithmeticTheory.num_CASES;
val disj_th = ⊢ m = 0 ∨ ∃n. m = SUC n: thm
```

Using two additional theorems, each having one disjunct as a hypothesis:

```
> show_assums := true;
val it = (): unit
> val th1 = EQT_ELIM $
    REWRITE_CONV [ASSUME "m = 0", prim_recTheory.PRE] "PRE m = m ⇔ m = 0";
val th1 =  [m = 0] ⊢ PRE m = m ⇔ m = 0: thm

> val th2_0 =
    REWRITE_CONV [ASSUME "m = SUC n", prim_recTheory.PRE,
```

```
                    arithmeticTheory.SUC_NOT_ZERO,
                    GSYM prim_recTheory.SUC_ID]
                  “PRE m = m ⇔ m = 0”;
val th2_0 =  [m = SUC n] ⊢ (PRE m = m ⇔ m = 0) ⇔ T: thm

> val th2 = CHOOSE (“n:num”, ASSUME “∃n. m = SUC n”) (EQT_ELIM th2_0);
val th2 =  [∃n. m = SUC n] ⊢ PRE m = m ⇔ m = 0: thm
```

a new theorem can be derived:

```
> DISJ_CASES disj_th th1 th2;
val it =  [] ⊢ PRE m = m ⇔ m = 0: thm
```

Comments

Neither of the ‘case’ theorems is required to have either disjunct as a hypothesis, but otherwise `DISJ_CASES` is pointless.

See also

`Thm.CHOOSE`, `Tactic.DISJ_CASES_TAC`, `Thm_cont.DISJ_CASES_THEN`, `Thm_cont.DISJ_CASES_THEN2`, `Drule.DISJ_CASES_UNION`, `Thm.DISJ1`, `Thm.DISJ2`

DISJ_CASES_TAC (Tactic)

```
Tactic.DISJ_CASES_TAC : thm_tactic
```

Produces a case split based on a disjunctive theorem.

Given a theorem `th` of the form `A |- u \/ v`, `DISJ_CASES_TAC th` applied to a goal produces two subgoals, one with `u` as an assumption and one with `v`:

```
            A ?- t
============================  DISJ_CASES_TAC (A |- u \/ v)
 A u {u} ?- t   A u {v}?- t
```

Failure

Fails if the given theorem does not have a disjunctive conclusion.

Example

Given the simple fact about arithmetic `th`, `|- (m = 0) \/ (?n. m = SUC n)`, the tactic `DISJ_CASES_TAC th` can be used to produce a case split:

```
- DISJ_CASES_TAC th ([],Term`(P:num -> bool) m`);
([([`m = 0`], `P m`),
  ([`?n. m = SUC n`], `P m`)], fn) : tactic_result
```

Performing a case analysis according to a disjunctive theorem.

See also

`Tactic.ASSUME_TAC`, `Tactic.ASM_CASES_TAC`, `Tactic.COND_CASES_TAC`, `Thm_cont.DISJ_CASES_THEN`, `Tactic.STRUCT_CASES_TAC`

DISJ_CASES_THEN (Thm_cont)

`Thm_cont.DISJ_CASES_THEN : thm_tactical`

Applies a theorem-tactic to each disjunct of a disjunctive theorem.

If the theorem-tactic `f:thm->tactic` applied to either `ASSUME`d disjunct produces results as follows when applied to a goal `(A ?- t)`:

```
 A ?- t                                  A ?- t
=========  f (u |- u)      and          =========  f (v |- v)
 A ?- t1                                 A ?- t2
```

then applying `DISJ_CASES_THEN f (|- u \/ v)` to the goal `(A ?- t)` produces two subgoals.

```
        A ?- t
======================  DISJ_CASES_THEN f (|- u \/ v)
 A ?- t1      A ?- t2
```

Failure

Fails if the theorem is not a disjunction. An invalid tactic is produced if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

Example

Given the theorem

```
th = |- (m = 0) \/ (?n. m = SUC n)
```

and a goal of the form `?- (PRE m = m) = (m = 0)`, applying the tactic

```
DISJ_CASES_THEN ASSUME_TAC th
```

produces two subgoals, each with one disjunct as an added assumption:

```
?n. m = SUC n ?- (PRE m = m) = (m = 0)
```

```
m = 0 ?- (PRE m = m) = (m = 0)
```

Building cases tactics. For example, `DISJ_CASES_TAC` could be defined by:

```
val DISJ_CASES_TAC = DISJ_CASES_THEN ASSUME_TAC
```

Comments

Use `DISJ_CASES_THEN2` to apply different tactic generating functions to each case.

See also

`Thm_cont.CHOOSE_THEN`, `Thm_cont.CONJUNCTS_THEN`, `Thm_cont.CONJUNCTS_THEN2`, `Thm.DISJ_CASES`, `Tactic.DISJ_CASES_TAC`, `Thm_cont.DISJ_CASES_THEN2`, `Thm_cont.DISJ_CASES_THENL`, `Thm_cont.STRIP_THM_THEN`

DISJ_CASES_THEN2 (Thm_cont)

```
Thm_cont.DISJ_CASES_THEN2 : (thm_tactic -> thm_tactical)
```

Applies separate theorem-tactics to the two disjuncts of a theorem.

If the theorem-tactics `f1` and `f2`, applied to the `ASSUME`d left and right disjunct of a theorem `|- u \/ v` respectively, produce results as follows when applied to a goal `(A ?- t)`:

```
 A ?- t                                  A ?- t
=========  f1 (u |- u)      and         =========  f2 (v |- v)
 A ?- t1                                 A ?- t2
```

then applying `DISJ_CASES_THEN2 f1 f2 (|- u \/ v)` to the goal `(A ?- t)` produces two subgoals.

```
        A ?- t
======================  DISJ_CASES_THEN2 f1 f2 (|- u \/ v)
 A ?- t1      A ?- t2
```

Failure

Fails if the theorem is not a disjunction. An invalid tactic is produced if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

Example

Given the theorem

```
th = |- (m = 0) \/ (?n. m = SUC n)
```

and a goal of the form `?- (PRE m = m) = (m = 0)`, applying the tactic

```
DISJ_CASES_THEN2 SUBST1_TAC ASSUME_TAC th
```

to the goal will produce two subgoals

```
?n. m = SUC n ?- (PRE m = m) = (m = 0)

?- (PRE 0 = 0) = (0 = 0)
```

The first subgoal has had the disjunct `m = 0` used for a substitution, and the second has added the disjunct to the assumption list. Alternatively, applying the tactic

```
DISJ_CASES_THEN2 SUBST1_TAC (CHOOSE_THEN SUBST1_TAC) th
```

to the goal produces the subgoals:

```
?- (PRE(SUC n) = SUC n) = (SUC n = 0)

?- (PRE 0 = 0) = (0 = 0)
```

Building cases tacticals. For example, `DISJ_CASES_THEN` could be defined by:

```
let DISJ_CASES_THEN f = DISJ_CASES_THEN2 f f
```

See also

`Thm_cont.STRIP_THM_THEN`, `Thm_cont.CHOOSE_THEN`, `Thm_cont.CONJUNCTS_THEN`, `Thm_cont.CONJUNCTS_THEN2`, `Thm_cont.DISJ_CASES_THEN`, `Thm_cont.DISJ_CASES_THENL`

DISJ_CASES_THENL (Thm_cont)

```
Thm_cont.DISJ_CASES_THENL : (thm_tactic list -> thm_tactic)
```

Applies theorem-tactics in a list to the corresponding disjuncts in a theorem.

If the theorem-tactics `f1...fn` applied to the `ASSUME`d disjuncts of a theorem

```
|- d1 \/ d2 \/...\/ dn
```

produce results as follows when applied to a goal `(A ?- t)`:

```
 A ?- t                                  A ?- t
=========  f1 (d1 |- d1) and ... and =========  fn (dn |- dn)
 A ?- t1                                 A ?- tn
```

then applying `DISJ_CASES_THENL [f1;...;fn] (|- d1 \/...\/ dn)` to the goal `(A ?- t)` produces n subgoals.

```
         A ?- t
======================  DISJ_CASES_THENL [f1;...;fn] (|- d1 \/...\/ dn)
 A ?- t1  ...  A ?- tn
```

`DISJ_CASES_THENL` is defined using iteration, hence for theorems with more than `n` disjuncts, `dn` would itself be disjunctive.

Failure

Fails if the number of tactic generating functions in the list exceeds the number of disjuncts in the theorem. An invalid tactic is produced if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

Used when the goal is to be split into several cases, where a different tactic-generating function is to be applied to each case.

See also

`Thm_cont.CHOOSE_THEN`, `Thm_cont.CONJUNCTS_THEN`, `Thm_cont.CONJUNCTS_THEN2`, `Thm_cont.DISJ_CASES_THEN`, `Thm_cont.DISJ_CASES_THEN2`, `Thm_cont.STRIP_THM_THEN`

DISJ_CASES_UNION (Drule)

```
Drule.DISJ_CASES_UNION : thm -> thm -> thm -> thm
```

Makes an inference for each arm of a disjunct.

Given a disjunctive theorem, and two additional theorems each having one disjunct as a hypothesis, a new theorem with a conclusion that is the disjunction of the conclusions of the last two theorems is produced. The hypotheses include the union of hypotheses of all three theorems less the two disjuncts.

```
 A |- t1 \/ t2    A1 u {t1} |- t3     A2 u {t2} |- t4
-------------------------------------------------------  DISJ_CASES_UNION
              A u A1 u A2 |- t3 \/ t4
```

Failure

Fails if the first theorem is not a disjunction.

Example

The built-in theorem `LESS_CASES` can be specialized to:

```
th1 = |- m < n \/ n <= m
```

and used with two additional theorems:

```
th2 = (m < n |- (m MOD n = m))
th3 = ({0 < n, n <= m} |- (m MOD n) = ((m - n) MOD n))
```

to derive a new theorem:

```
- DISJ_CASES_UNION th1 th2 th3;
val it = [0 < n] |- (m MOD n = m) \/ (m MOD n = (m - n) MOD n) : thm
```

See also

`Thm.DISJ_CASES`, `Tactic.DISJ_CASES_TAC`, `Thm.DISJ1`, `Thm.DISJ2`

DISJ_IMP (Drule)

```
Drule.DISJ_IMP : (thm -> thm)
```

Converts a disjunctive theorem to an equivalent implicative theorem.

The left disjunct of a disjunctive theorem becomes the negated antecedent of the newly generated theorem.

```
    A |- t1 \/ t2
-----------------  DISJ_IMP
 A |- ~t1 ==> t2
```

Failure

Fails if the theorem is not a disjunction.

Example

Specializing the built-in theorem `LESS_CASES` gives the theorem:

```
th = |- m < n \/ n <= m
```

to which `DISJ_IMP` may be applied:

```
- DISJ_IMP th;
> val it = |- ~m < n ==> n <= m : thm
```

See also

`Thm.DISJ_CASES`

DISJ_INEQS_FALSE_CONV (Arith)

```
Arith.DISJ_INEQS_FALSE_CONV : conv
```

Proves a disjunction of conjunctions of normalised inequalities is false, provided each conjunction is unsatisfiable.

`DISJ_INEQS_FALSE_CONV` converts an unsatisfiable normalised arithmetic formula to false. The formula must be a disjunction of conjunctions of less-than-or-equal-to inequalities. The inequalities must have the following form: Each variable must appear on only one side of the inequality and each side must be a linear sum in which any constant appears first followed by products of a constant and a variable. On each side the variables must be ordered lexicographically, and if the coefficient of the variable is `1`, the `1` must appear explicitly.

Failure

Fails if the formula is not of the correct form or is satisfiable. The function will also fail on certain unsatisfiable formulae due to incompleteness of the procedure used.

Example

```
#DISJ_INEQS_FALSE_CONV
# "(1 * n) <= ((1 * m) + (1 * p)) /\
#  ((1 * m) + (1 * p)) <= (1 * n) /\
#  (5 + (4 * n)) <= ((3 * m) + (1 * p)) \/
#  2 <= 0";;
|- (1 * n) <= ((1 * m) + (1 * p)) /\
   ((1 * m) + (1 * p)) <= (1 * n) /\
   (5 + (4 * n)) <= ((3 * m) + (1 * p)) \/
   2 <= 0 =
   F
```

See also

Arith.ARITH_FORM_NORM_CONV

disjunction (boolSyntax)

```
boolSyntax.disjunction : term
```

Constant denoting logical disjunction.

The ML variable boolSyntax.disjunction is bound to the term bool$\/.

See also

boolSyntax.equality, boolSyntax.implication, boolSyntax.select, boolSyntax.T, boolSyntax.F, boolSyntax.universal, boolSyntax.existential, boolSyntax.exists1, boolSyntax.conjunction, boolSyntax.negation, boolSyntax.conditional, boolSyntax.bool_case, boolSyntax.let_tm, boolSyntax.arb

DISJUNCTS_AC (Drule)

```
Drule.DISJUNCTS_AC : term * term -> thm
```

Prove equivalence under idempotence, symmetry and associativity of disjunction.

`DISJUNCTS_AC` takes a pair of terms `(t1, t2)` and proves `|- t1 = t2` if `t1` and `t2` are equivalent up to idempotence, symmetry and associativity of disjunction. That is, if `t1` and `t2` are two (different) arbitrarily-nested disjunctions of the same set of terms, then `DISJUNCTS_AC (t1,t2)` returns `|- t1 = t2`. Otherwise, it fails.

Failure

Fails if `t1` and `t2` are not equivalent, as described above.

Example

```
> DISJUNCTS_AC (Term `(P \/ Q) \/ R`, Term `R \/ (Q \/ R) \/ P`);
val it = ⊢ (P ∨ Q) ∨ R ⇔ R ∨ (Q ∨ R) ∨ P: thm
```

Used to reorder a disjunction. First sort the disjuncts in a term `t1` into the desired order (e.g., lexicographic order, for normalization) to get a new term `t2`, then call `DISJUNCTS_AC(t1,t2)`.

See also

`Drule.CONJUNCTS_AC`

DIV_CONV (reduceLib)

```
reduceLib.DIV_CONV : conv
```

Calculates by inference the result of dividing, with truncation, one numeral by another.

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `DIV_CONV ``m DIV n``` returns the theorem:

```
   |- m DIV n = s
```

where s is the numeral that denotes the result of dividing the natural number denoted by m by the natural number denoted by n, with truncation.

Failure

DIV_CONV tm fails unless tm is of the form ``m DIV n``, where m and n are numerals, or if n denotes zero.

Example

```
> Arithconv.DIV_CONV ``0 DIV 0``;
val it = ⊢ 0 DIV 0 = 0: thm

> Arithconv.DIV_CONV ``x DIV 12``;
Exception- HOL_ERR
  (at Conv.RAND_CONV:
     at Thm.dest_cexp: term is not a compute value) raised

> Arithconv.DIV_CONV ``0 DIV 12``;
val it = ⊢ 0 DIV 12 = 0: thm

> Arithconv.DIV_CONV ``7 DIV 2``;
val it = ⊢ 7 DIV 2 = 3: thm
```

DNF_ss (boolSimps)

```
boolSimps.DNF_ss : ssfrag
```

A simpset fragment that does aggressive propositional and quantifier normalisation.

Adding the DNF_ss simpset fragment to a simpset augments it with rewrites that make the simplifier normalise “towards” disjunctive normal form. This normalisation at the propositional level does leave implications alone (rather than convert them to disjunctions). DNF_ss also includes normalisations pertaining to quantifiers. The complete list of rewrites is

```
|- !P Q. (!x. P x /\ Q x) <=> (!x. P x) /\ !x. Q x
|- !P Q. (?x. P x \/ Q x) <=> (?x. P x) \/ ?x. Q x
|- !P Q R. P \/ Q ==> R <=> (P ==> R) /\ (Q ==> R)
|- !P Q R. P ==> Q /\ R <=> (P ==> Q) /\ (P ==> R)
|- !A B C. (B \/ C) /\ A <=> B /\ A \/ C /\ A
|- !A B C. A /\ (B \/ C) <=> A /\ B \/ A /\ C
|- !P Q. (?x. P x) ==> Q <=> !x. P x ==> Q
|- !P Q. P ==> (!x. Q x) <=> !x. P ==> Q x
|- !P Q. (?x. P x) /\ Q <=> ?x. P x /\ Q
|- !P Q. P /\ (?x. Q x) <=> ?x. P /\ Q x
```

Failure

As a value rather than a function, `DNF_ss` can't fail.

Example

```
> SIMP_CONV (bool_ss ++ DNF_ss) []
           ``!x. (?y. P x y) /\ Q z ==> R1 x z /\ R2 z x``;
val it =
   ⊢ (∀x. (∃y. P x y) ∧ Q z ⇒ R1 x z ∧ R2 z x) ⇔
     (∀x y. P x y ∧ Q z ⇒ R1 x z) ∧ ∀x y. P x y ∧ Q z ⇒ R2 z x: thm
```

Comments

The `DNF_ss` fragment interacts well with the one-point elimination rules for equalities under quantifiers (provided in `bool_ss` and its descendants).

See also

`boolSimps.bool_ss`, `simpLib.SIMP_CONV`

dom_rng (Type)

```
Type.dom_rng : hol_type -> hol_type * hol_type
```

Breaks a function type into domain and range types.

If `ty` has the form `ty1 -> ty2`, then `dom_rng ty` yields `(ty1,ty2)`.

Failure

Fails if `ty` is not a function type.

Example

```
> dom_rng (bool --> alpha);
val it = (":bool", ":α"): hol_type * hol_type

> try dom_rng bool;
Exception- HOL_ERR (at Type.dom_rng: not a function type) raised
```

See also

`Type.-->`, `Type.dest_type`, `Type.dest_thy_type`

drule (Tactic)

```
Tactic.drule : thm -> tactic
```

Performs one step of resolution (or modus ponens) against implication theorem.

If theorem `th` is of the form `A |- t`, where `t` is of the form `!x1..xn. P .. /\ ... ==> Q` or `!x1..xn. P .. ==> Q`, then a call to `drule th (asl,g)` looks for an assumption in `asl` that matches the pattern `P ..` in `t`. It then performs instantiation of `th`'s universally quantified and free variables, transforms any conjunctions on the left into a minimal number of chained implications, and calls `MP` once to generate a consequent theorem `A |- t'`. This theorem is then passed to `MP_TAC`, turning the goal into `(asl, t' ==> g)`. (We assume that `A` is a subset of `asl`; otherwise the tactic is invalid.)

Failure

A call to `drule th (asl,g)` will fail if `th` is not a (possibly universally quantified) implication (or negation), or if there are no assumptions in `asl` matching the "first"

hypothesis of `th`.

Example

The `DIV_LESS` theorem states:

```
!n d. 0 < n /\ 1 < d ==> (n DIV d < n)
```

Then:

```
> drule arithmeticTheory.DIV_LESS (["1 < x", "0 < y"], "P:bool");
val it =
   ([(["1 < x", "0 < y"], "(!d. 1 < d ==> y DIV d < y) ==> P")], fn):
   goal list * validation
```

Comments

The `drule` tactic is similar to, but a great deal more controlled than, the `IMP_RES_TAC` tactic, which will look for a great many more possible instantiations and resolutions to perform. `IMP_RES_TAC` also puts all of its derived consequences into the assumption list; `drule` can be sure that there will be just one such consequence, and puts this into the goal directly.

The related `dxrule` tactic removes the matching assumption from the assumption list. In this example above, the resulting assumption list would just contain `1 < x`, having lost the `0 < y` which was resolved against the `DIV_LESS` theorem.

The `drule` tactic uses the `MP_TAC` `thm_tactic` as its fixed continuation; the `drule_then` variation takes a `thm_tactic` continuation as its first parameter and applies this to the result of the instantiation and `MP` call. There is also a `dxrule_then`, that combines both variations described here.

Finally, note that the implicational theorem may itself have come from the goal's assumptions, extracted with tools such as `FIRST_ASSUM` and `PAT_ASSUM`.

See also

`Tactic.drule_all`, `Tactic.IMP_RES_TAC`

drule_all (Tactic)

```
Tactic.drule_all : thm -> tactic
```

Attempts to discharge all of a theorem's antecedents from assumptions

If `th` is a theorem with a conclusion that is a (possibly universally quantified) implication (or negation), the theorem-tactic `drule_all` (implicitly) normalises it have form

```
!v1 .. vn. P1 ==> (P2 ==> (P3 ==> ... Q)...)
```

where each `Pi` is not a conjunction. (In other words, `P /\ Q ==> R` is normalised to `P ==> (Q ==> R)`.) An application of `drule_all th` to a goal then attempts to find a consistent instantiation so that all of the `Pi` antecedents can be discharged by appeal to the goal's assumptions. If this repeated instantiation and use of `MP` is possible, then the (instantiated) conclusion `Q` is added to the goal with the `MP_TAC` `thm_tactic`.

When finding assumptions, those that have been most recently added to the assumption list will be preferred.

Failure

An invocation of `drule_all th` can only fail when applied to a goal. It can then fail if `th` is not an implication, or if all of `th`'s implications cannot be eliminated by matching against the goal's assumptions.

Example

The `LESS_LESS_EQ_TRANS` theorem states:

```
!m n p. m < n /\ n <= p ==> m < p
```

Then:

```
> drule_all arithmeticTheory.LESS_LESS_EQ_TRANS
   (["x < w", "1 < x", "z <= y", "x <= z", "y < z"], "P:bool");
val it =
   ([(["x < w", "1 < x", "z <= y", "x <= z", "y < z"],
      "1 < z ==> P")], fn): goal list * validation
```

Note how the other possible instance of the theorem (chaining `y < z` and `z <= y`) is not found.

Comments

The variant `dxrule_all` removes used assumptions from the assumption list as they resolve against the theorem. The variant `drule_all_then` allows a continuation other than `MP_TAC` to be used. The variant `dxrule_all_then` combines both.

A negated conclusion (`~Q`) is not treated as an implication (`Q ==> F`) so the tactic will not attempt to find an instantiation of `Q` among the assumptions.

See also

`Tactic.drule`, `Tactic.IMP_RES_TAC`

e (proofManagerLib)

```
proofManagerLib.e : tactic -> proof
```

Applies a tactic to the current goal, stacking the resulting subgoals.

The function `e` is part of the subgoal package. It is an abbreviation for `expand`. For a description of the subgoal package, see `set_goal`.

Failure

As for `expand`.

Doing a step in an interactive goal-directed proof.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

eall (proofManagerLib)

```
proofManagerLib.eall : tactic -> proof
```

Applies a tactic to all goals in the current goal list, replacing the list with the resulting subgoals.

`eall tac` applies `tac` to all the goals in the current goal list, replacing the goal list with the list of all the resulting subgoals. It is an abbreviation for `expand_list (ALLGOALS tac)`.

For interactively constructing suitable compound tactics: in an interactive proof, the effect of `e (tac1 THEN tac2)` can be obtained by `e tac1` and then `eall tac2`.

See also

`proofManagerLib.expand_list`, `proofManagerLib.elt`, `Tactical.ALLGOALS`, `proofManagerLib.e`
`proofManagerLib.set_goal`

el (Lib)

```
Lib.el : int -> 'a list -> 'a
```

Extracts a specified element from a list.

`el i [x1,...,xn]` returns `xi`. Note that the elements are numbered starting from `1`, not `0`.

Failure

Fails with `el` if the integer argument is less than 1 or greater than the length of the list.

Example

```
> el 3 [1,2,7,1];
val it = 7: int
```

See also

`Lib.index`

EL_CONV (listLib)

`listLib.EL_CONV : conv`

Computes by inference the result of indexing an element from a list.

For any object language list of the form "`[x0;...xk;...;xn]`", the result of evaluating

```
EL_CONV "EL k [x0;...xk;...;xn]"
```

is the theorem

```
|- EL k [x0;...;xk;...;xn] = xk
```

Failure

`EL_CONV tm` fails if `tm` is not of the form described above, or `k` is not less than the length of the list.

See also

`listLib.ELL_CONV`

ELL_CONV (listLib)

`listLib.ELL_CONV : conv`

Computes by inference the result of indexing an element of a list from the tail end.

For any object language list of the form "`[xn-1;...;xk;...x0]`", the result of evaluating

```
ELL_CONV “ELL k [xn-1;...;xk;...;x0]”
```

is the theorem

```
|- ELL k [xn-1;...;xk;...;x0] = xk
```

where `k` must not be greater then the length of the list. Note that `ELL` indexes the list elements from the tail end.

Failure

`ELL_CONV tm` fails if `tm` is not of the form described above, or `k` is not less than the length of the list.

See also

`listLib.EL_CONV`

elt (proofManagerLib)

```
proofManagerLib.elt : list_tactic -> proof
```

Applies a list-tactic to the current goal list, replacing it with the resulting subgoals.

The function `elt` is part of the subgoal package. It is an abbreviation for `expand_list`. For a description of the subgoal package, see `set_goal`.

Failure

As for `expand_list`.

Doing a step in an interactive goal-directed proof, where the step may affect the list of goals produced by the previous step.

See also

`proofManagerLib.expand_list`, `proofManagerLib.set_goal`

emit_ERR (Feedback)

```
Feedback.emit_ERR : bool ref
```

Flag controlling output of HOL_ERR exceptions.

The boolean flag emit_ERR tells whether an application of HOL_ERR should be printed. Its value is consulted by Raise when it attempts to print a textual representation of its argument exception. This flag is not commonly used, and it may disappear or change in the future.

The default value of emit_ERR is true.

Example

```
> Raise (mk_HOL_ERR "Module" "function" "message");
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.

Exception raised at Module.function: message
HOL_ERR (at Module.function: message) raised

> emit_ERR := false;
val it = (): unit

> Raise (mk_HOL_ERR "Module" "function" "message");
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
HOL_ERR (at Module.function: message) raised
```

See also

Feedback, Feedback.Raise, Feedback.emit_MESG, Feedback.emit_WARNING

emit_MESG (Feedback)

```
Feedback.emit_MESG : bool ref
```

Flag controlling output of HOL_MESG function.

The boolean flag emit_MESG is consulted by HOL_MESG when it attempts to print its argument. This flag is not commonly used, and it may disappear or change in the future.

The default value of emit_MESG is true.

Example

```
> HOL_MESG "Joy to the world.";
<<HOL message: Joy to the world.>>
val it = (): unit

> emit_MESG := false;
val it = (): unit

> HOL_MESG "Peace on Earth.";
val it = (): unit
```

See also

Feedback, Feedback.HOL_MESG, Feedback.emit_ERR, Feedback.emit_WARNING

emit_WARNING (Feedback)

```
Feedback.emit_WARNING : bool ref
```

Flag controlling output of HOL_WARNING function.

The boolean flag emit_WARNING is consulted by HOL_WARNING when it attempts to print its argument. This flag is not commonly used, and it may disappear or change in the future.

The default value of emit_WARNING is true.

Example

```
> HOL_WARNING "Clock" "watcher" "Time is running out.";
<<HOL warning: Clock.watcher: Time is running out.>>
val it = (): unit

> emit_WARNING := false;
val it = (): unit

> HOL_WARNING "Clock" "watcher" "Time is running out.";
val it = (): unit
```

See also

Feedback, Feedback.HOL_WARNING, Feedback.emit_ERR, Feedback.emit_MESG

empty_rewrites (Rewrite)

```
Rewrite.empty_rewrites: rewrites
```

The empty database of rewrite rules.

Used to build other rewrite sets.

See also

Rewrite.bool_rewrites, Rewrite.implicit_rewrites, Rewrite.add_rewrites, Rewrite.add_implicit_rewrites, Rewrite.set_implicit_rewrites

empty_tmset (Term)

```
Term.empty_tmset : term set
```

Empty set of terms.

The value `empty_tmset` represents an empty set of terms. The set has a built-in ordering, which is given by `Term.compare`.

Comments

Used as a starting point for building sets of terms.

See also

`Term.compare`, `Term.empty_varset`

empty_varset (Term)

```
Term.empty_varset : term set
```

Empty set of term variables.

The value `empty_varset` represents an empty set of term variables. The set has a built-in ordering, which is given by `Term.var_compare`.

Comments

Used as a starting point for building sets of variables.

See also

`Term.var_compare`, `Term.empty_tmset`

enable_monad (monadsyntax)

```
monadsyntax.enable_monad : string -> unit
```

Enables a particular monadic type for use with do/od syntax.

A call to `enable_monad mname`, where `mname` is an SML value of type `string`, enables the stored information about monad `mname` to govern the interpretation of the do/od syntax. If multiple monads are enabled, normal overloading resolution will decide between them.

Failure

Fails if `mname` is not the name of a stored monad in the internal database (which can be examined with a call to `monadsyntax.all_monads()`. Will have little effect if monad syntax has not been generally enabled with a prior call to `enable_monadsyntax`.

Example

In what follows, `oHD` is the function which maps a non-empty list to `SOME` applied to that list's first element, and the empty list to `NONE`. The `++` is the monad choice function (the option monad has a notion of failure). Thus, the function below that is bound to SML variable `f` is one that either increments the first element of a list and returns that value, or returns 0.

```
> monadsyntax.enable_monadsyntax(); monadsyntax.enable_monad "option";
val it = (): unit
val it = (): unit

> val f = “λl. do x <- oHD l; return (x + 1); od ++ return 0”
val f = “λl. do x <- oHD l; SOME (x + 1) od ++ SOME 0”: term

> EVAL “^f [3; 10]”;
val it = ⊢ (λl. do x <- oHD l; SOME (x + 1) od ++ SOME 0) [3; 10] = SOME 4:
   thm

> EVAL “^f []”;
val it = ⊢ (λl. do x <- oHD l; SOME (x + 1) od ++ SOME 0) [] = SOME 0: thm
```

Note how the `return` keyword is not printed as such by the parser; it would be too confusing if all occurrences of common functions such as `SOME` were printed as `return`.

Comments

As with other parsing and printing functions, there is a `temp_enable_monad` function whose changes to the parser and printer do not persist to descendent theories.

See also

`monadsyntax.all_monads`, `monadsyntax.declare_monad`, `monadsyntax.enable_monadsyntax`

enable_monadsyntax (monadsyntax)

```
monadsyntax.enable_monadsyntax : unit -> unit
```

Enables parsing and printing of monadic do/od syntax.

A call to `enable_monadsyntax()` alters the parser and pretty-printer to support the do/od syntax for writing monadic values. This call should be followed by calls to `enable_monad` (or `weak_enable_monad`) so that the do/od syntax can be linked to actual monadic types.

Failure

Never fails.

Example

This first example gives a clear demonstration of the nature of the syntactic translation that the monad syntax implements because there is no specific enabled monad for the syntax to map to:

```
> monadsyntax.enable_monadsyntax();
val it = () : unit

> “do M1 ; M2; od”;
val it = “monad_unitbind M1 M2” : term;

> “do v <- M1; w <- M2 v 3; return (v + w); od”;
```

```
val it = “monad_bind M1 (λv. monad_bind (M2 v 3) (λw. return (v + w)))”
         : term
```

The `monad_bind`, `monad_unitbind` and `return` terms above are variables that would be instantiated with the appropriate terms given the available choices of enabled monads.

See also

`monadsyntax.all_monads`, `monadsyntax.enable_monad`

end_itlist (Lib)

```
Lib.end_itlist : ('a -> 'a -> 'a) -> 'a list -> 'a
```

List iteration function. Applies a binary function between adjacent elements of a list.

`end_itlist f [x1,...,xn]` returns `f x1 ( ... (f x(n-1) xn)...)`. Returns `x` for a one-element list `[x]`.

Failure

Fails if list is empty, or if an application of `f` raises an exception.

Example

```
> end_itlist (curry op+) [1,2,3,4];
val it = 10: int
```

See also

`Lib.itlist`, `Lib.rev_itlist`, `Lib.itlist2`, `Lib.rev_itlist2`

end_time (Lib)

```
Lib.end_time : Timer.cpu_timer -> unit
```

Check a running timer, and print out how long it has been running.

An application `end_time timer` looks to see how long `timer` has been running, and prints out the elapsed runtime, garbage collection time, and system time.

Failure

Never fails.

Example

```
> val clock = start_time();
val clock = ?: Timer.cpu_timer

> use "foo.sml";
Exception- Io
  {cause = SysErr ("No such file or directory", SOME ENOENT), function =
   "TextIO.openIn", name = "foo.sml"} raised

> end_time clock;
runtime: 0.00024s,    gctime: 0.00000s,     systime: 0.00002s.
val it = (): unit
```

Comments

A `start_time ... end_time` pair is for use when calling `time` would be clumsy, e.g., when multiple function applications are to be timed.

See also

`Lib.start_time`, `Lib.time`

enth (proofManagerLib)

```
proofManagerLib.enth : tactic -> int -> proof
```

Applies a tactic to one goal, referenced by number, in the current goal list, replacing that goal with the resulting subgoals.

`enth tac i` applies `tac` to all the i'th goal in the current goal list, replacing that goal in the goal list with the subgoals produced by `tac`. It is an abbreviation for `expand_list (NTH_GOAL tac i)`.

For interactively constructing suitable compound tactics, for example to test whether a particular subgoal can be proved easily, before attacking the other subgoals.

See also

`proofManagerLib.expand_list`, `proofManagerLib.elt`, `Tactical.NTH_GOAL`, `proofManagerLib.set_goal`, `proofManagerLib.r`

enumerate (Lib)

```
Lib.enumerate : int -> 'a list -> (int * 'a) list
```

Number each element of a list, in ascending order.

An invocation of `enumerate i [x1, ..., xn]` returns the list `[(i,x1), (i+1,x2), ..., (i+n-1,xn)]`.

Failure

Never fails.

Example

```
> enumerate 0 ["komodo", "iguana", "gecko", "gila"];
val it = [(0, "komodo"), (1, "iguana"), (2, "gecko"), (3, "gila")]:
   (int * string) list
```

EQ_IMP_RULE (Thm)

```
Thm.EQ_IMP_RULE : thm -> thm * thm
```

Derives forward and backward implication from equality of boolean terms.

When applied to a theorem `A |- t1 = t2`, where `t1` and `t2` both have type `bool`, the inference rule `EQ_IMP_RULE` returns the theorems `A |- t1 ==> t2` and `A |- t2 ==> t1`.

```
            A |- t1 = t2
-----------------------------------  EQ_IMP_RULE
 A |- t1 ==> t2     A |- t2 ==> t1
```

Failure

Fails unless the conclusion of the given theorem is an equation between boolean terms.

See also

`Thm.EQ_MP`, `Tactic.EQ_TAC`, `Drule.IMP_ANTISYM_RULE`

EQ_LENGTH_INDUCT_TAC (listLib)

```
listLib.EQ_LENGTH_INDUCT_TAC : tactic
```

Performs tactical proof by structural induction on two equal length lists.

`EQ_LENGTH_INDUCT_TAC` reduces a goal `!x y . (LENGTH x = LENGTH y) ==> t[x,y]`, where `x` and `y` range over lists, to two subgoals corresponding to the base and step cases in a proof by induction on the length of `x` and `y`. The induction hypothesis appears among the assumptions of the subgoal for the step case. The specification of `EQ_LENGTH_INDUCT_TAC` is:

```
      A ?- !x y . (LENGTH x = LENGTH y) ==> t[x,y]
================================================  EQ_LENGTH_INDUCT_TAC
                          A ?- t[NIL/x][NIL/y]
 A u {{LENGTH x = LENGTH y, t[x'/x, y'/y]}} ?-
      !h h'. t[(CONS h x)/x, (CONS h' y)/y]
```

Failure

`EQ_LENGTH_INDUCT_TAC g` fails unless the conclusion of the goal `g` has the form

```
!x y . (LENGTH x = LENGTH y) ==> t[x,y]
```

where the variables `x` and `y` have types `(xty)list` and `(yty)list` for some types `xty` and `yty`. It also fails if either of the variables `x` or `y` appear free in the assumptions.

Use this tactic to perform structural induction over two lists that have identical length.

See also

`listLib.LIST_INDUCT_TAC`, `listLib.SNOC_INDUCT_TAC`, `listLib.EQ_LENGTH_SNOC_-INDUCT_TAC`

EQ_LENGTH_SNOC_INDUCT_TAC (listLib)

```
listLib.EQ_LENGTH_SNOC_INDUCT_TAC : tactic
```

Performs tactical proof by structural induction on two equal length lists from the tail end.

`EQ_LENGTH_SNOC_INDUCT_TAC` reduces a goal `!x y . (LENGTH x = LENGTH y) ==> t[x,y]`, where `x` and `y` range over lists, to two subgoals corresponding to the base and step cases in a proof by induction on the length of `x` and `y`. The induction hypothesis appears among the assumptions of the subgoal for the step case. The specification of `EQ_LENGTH_SNOC_INDUCT_TAC` is:

```
    A ?- !x y . (LENGTH x = LENGTH y) ==> t[x,y]
================================================  EQ_LENGTH_SNOC_INDUCT_TAC
```

```
                              A ?- t[NIL/x][NIL/y]
    A u {{LENGTH x = LENGTH y, t[x'/x, y'/y]}} ?-
         !h h'. t[(SNOC h x)/x, (SNOC h' y)/y]
```

Failure

`EQ_LENGTH_SNOC_INDUCT_TAC g` fails unless the conclusion of the goal `g` has the form

```
!x y . (LENGTH x = LENGTH y) ==> t[x,y]
```

where the variables `x` and `y` have types `(xty)list` and `(yty)list` for some types `xty` and `yty`. It also fails if either of the variables `x` or `y` appear free in the assumptions.

Use this tactic to perform structural induction on two lists that have identical length.

See also

`listLib.EQ_LENGTH_INDUCT_TAC`, `listLib.LIST_INDUCT_TAC`, `listLib.SNOC_INDUCT_TAC`

EQ_MP (Thm)

```
Thm.EQ_MP : thm -> thm -> thm
```

Equality version of the Modus Ponens rule.

When applied to theorems `A1 |- t1 = t2` and `A2 |- t1`, the inference rule `EQ_MP` returns the theorem `A1 u A2 |- t2`.

```
 A1 |- t1 = t2   A2 |- t1
-------------------------  EQ_MP
      A1 u A2 |- t2
```

Failure

Fails unless the first theorem is equational and its left side is the same as the conclusion of the second theorem (and is therefore of type `bool`), up to alpha-conversion.

See also

Thm.EQ_IMP_RULE, Drule.IMP_ANTISYM_RULE, Thm.MP

EQ_MP_TAC (Tactic)

```
Tactic.EQ_MP_TAC : thm -> tactic
```

A tactic that reverses EQ_MP, requiring proof of an equality.

A call to EQ_MP_TAC th, with th's conclusion being boolean term p, changes a goal (G, q) to be (G,p <=> q). If p <=> q is indeed provable, then an application of EQ_MP to that theorem and the provided th will be a proof of q (all in the context of assumptions G).

Failure

Never fails.

Example

```
> EQ_MP_TAC (CONJ TRUTH TRUTH) ([], “p ∧ q”);
val it = ([([], “T ∧ T ⇔ p ∧ q”)], fn): goal list * validation
```

Comments

Application of this tactic might be a prelude to showing that a number of sub-terms from the theorem's conclusion and the goal are equal (with tactics such as AP_TERM_TAC and CONG_TAC).

See also

Tactic.AP_TERM_TAC, Tactic.AP_THM_TAC, Tactic.CONG_TAC, Thm.EQ_MP, Tactic.MP_-TAC

EQ_TAC (Tactic)

```
Tactic.EQ_TAC : tactic
```

Reduces goal of equality of boolean terms to forward and backward implication.

When applied to a goal `A ?- t1 = t2`, where `t1` and `t2` have type `bool`, the tactic `EQ_TAC` returns the subgoals `A ?- t1 ==> t2` and `A ?- t2 ==> t1`.

```
           A ?- t1 = t2
================================  EQ_TAC
 A ?- t1 ==> t2   A ?- t2 ==> t1
```

Failure

Fails unless the conclusion of the goal is an equation between boolean terms.

Comments

Also available under the names `eq_tac` and `iff_tac`.

See also

`Thm.EQ_IMP_RULE`, `Drule.IMP_ANTISYM_RULE`

EQF_ELIM (Drule)

```
Drule.EQF_ELIM : (thm -> thm)
```

Replaces equality with `F` by negation.

```
 A |- tm = F
-------------  EQF_ELIM
  A |- ~tm
```

Failure

Fails if the argument theorem is not of the form `A |- tm = F`.

See also

`Drule.EQF_INTRO`, `Drule.EQT_ELIM`, `Drule.EQT_INTRO`

EQF_INTRO (Drule)

```
Drule.EQF_INTRO : (thm -> thm)
```

Converts negation to equality with `F`.

```
    A |- ~tm
-------------  EQF_INTRO
 A |- tm = F
```

Failure

Fails if the argument theorem is not a negation.

See also

`Drule.EQF_ELIM`, `Drule.EQT_ELIM`, `Drule.EQT_INTRO`

EQT_ELIM (Drule)

```
Drule.EQT_ELIM : (thm -> thm)
```

Eliminates equality with `T`.

```
 A |- tm = T
-------------  EQT_ELIM
   A |- tm
```

Failure

Fails if the argument theorem is not of the form `A |- tm = T`.

See also

`Drule.EQT_INTRO`, `Drule.EQF_ELIM`, `Drule.EQF_INTRO`

EQT_INTRO (Drule)

```
Drule.EQT_INTRO : thm -> thm
```

Introduces equality with `T`.

```
    A |- tm
-------------  EQT_INTRO
 A |- tm = T
```

Failure

Never fails.

See also

`Drule.EQT_ELIM`, `Drule.EQF_ELIM`, `Drule.EQF_INTRO`

equal (Lib)

```
Lib.equal : ''a -> ''a -> bool
```

Curried form of ML equality.

In some programming situations it is useful to use equality in a curried form. Although it is easy to code up on demand, the `equal` function is provided for convenience.

Failure

Never fails.

Example

```
> filter (equal 1) [1,2,1,4,5];
val it = [1, 1]: int list
```

equality (boolSyntax)

```
boolSyntax.equality : term
```

Constant denoting logical equality.

The ML variable `boolSyntax.equality` is bound to the term `min$=`.

See also

`boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.negation`, `boolSyntax.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`

ERR_outstream (Feedback)

```
Feedback.ERR_outstream : (string -> unit) ref
```

Reference to output stream used when printing `HOL_ERR`

The value of reference cell `ERR_outstream` controls where `Raise` prints its argument.

The default value of `ERR_outstream` is a function that prints its argument to `TextIO.stdErr`.

Example

```
> val outputs = ref ([] : string list);
val outputs = ref []: string list ref

> ERR_outstream := (fn s => outputs := !outputs @ [s]);
val it = (): unit

> Raise (mk_HOL_ERR "Foo" "bar" "incomprehensible input");
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
HOL_ERR (at Foo.bar: incomprehensible input) raised

> outputs;
val it = ref ["\nException raised at Foo.bar: incomprehensible input\n"]:
   string list ref
```

See also

Feedback, Feedback.HOL_ERR, Feedback.Raise, Feedback.MESG_outstream, Feedback.WARNING_outstream

ERR_to_string (Feedback)

```
Feedback.ERR_to_string : (error_record -> string) ref
```

Alterable function for formatting HOL_ERR.

ERR_to_string is a reference to a function for formatting the argument to an application of HOL_ERR. It can be used to customize Raise.

The default value of ERR_to_string is format_ERR.

Example

```
> fun alt_ERR_report {origin_structure,origin_function,message} =
    String.concat["This just in from ",origin_function, " at ",
```

```
                  origin_structure, " : ", message, "\n"];
val alt_ERR_report = fn:
   {message: string, origin_function: string, origin_structure: string} ->
     string

> ERR_to_string := alt_ERR_report;
Exception- Type error in function application.
   Function: := :
      (hol_error -> string) ref * (hol_error -> string) -> unit
   Argument: (ERR_to_string, alt_ERR_report) :
      (hol_error -> string) ref *
      ({message: string,
        origin_function: string, origin_structure: string} -> string)
   Reason:
      Can't unify hol_error to
         {message: string,
           origin_function: string, origin_structure: string}
         (Incompatible types)
Fail "Static Errors" raised

> Raise (HOL_ERR {origin_structure = "Foo",
                  origin_function = "bar",
                  message = "incomprehensible input"});
Exception- Type error in function application.
   Function: HOL_ERR : hol_error -> exn
   Argument:
      {origin_structure = "Foo", origin_function = "bar",
        message = "incomprehensible input"} :
      {message: string,
        origin_function: string, origin_structure: string}
   Reason:
      Can't unify hol_error to
         {message: string,
           origin_function: string, origin_structure: string}
         (Incompatible types)
Fail "Static Errors" raised
```

See also

Feedback, Feedback.error_record, Feedback.HOL_ERR, Feedback.Raise, Feedback.MESG_to_string, Feedback.WARNING_to_string

error_record (Feedback)

```
Feedback.type error_record = {origin_structure : string,
                      origin_function  : string,
                      message          : string}
```

Type abbreviation for HOL exceptions in Feedback module.

The type abbreviation `error_record` declares the standard format of HOL exceptions. The `origin_structure` field denotes the module that the exception has been raised in, the `origin_function` field gives the name of the function the exception has been raised in, and the `message` field should give an explanation of why the exception has been raised.

See also

Feedback, Feedback.HOL_ERR, Feedback.format_ERR, Feedback.ERR_to_string

eta (proofManagerLib)

```
proofManagerLib.eta : tactic -> proof
```

Applies a tactic to all goals, on which it succeeds, in the current goal list, replacing the list with the resulting subgoals.

`eta tac` tries to apply `tac` to all the goals in the current goal list; replacing the goal list with the list of all the resulting subgoals. If it fails on a goal, it has no effect on that goal. It is an abbreviation for `expand_list (TRYALL tac)`.

For interactively constructing suitable compound tactics: in an interactive proof, the effect of `e (tac1 THEN TRY tac2)` can be obtained by `e tac1` and then `eta tac2`.

See also

`proofManagerLib.expand_list`, `proofManagerLib.elt`, `Tactical.TRYALL`, `Tactical.TRY`, `proofManagerLib.eall`, `proofManagerLib.set_goal`

ETA_CONV (Drule)

```
Drule.ETA_CONV : conv
```

Performs a toplevel eta-conversion.

`ETA_CONV` maps an eta-redex `\x. (t x)`, where `x` does not occur free in `t`, to the theorem `|- (\x. (t x)) = t`.

Failure

Fails if the input term is not an eta-redex.

See also

`Drule.RIGHT_ETA`, `Term.eta_conv`

eta_conv (Term)

```
Term.eta_conv : term -> term
```

Performs one step of eta-reduction.

Eta-reduction is an important operation in the lambda calculus. A step of eta-reduction may be performed by `eta_conv M`, where `M` is a lambda abstraction of the following form: `\v. (N v)`, i.e., a lambda abstraction whose body is an application of a term `N` to the bound variable `v`. Moreover, `v` must not occur free in `M`. If this proviso is met, an invocation `eta_conv (\v. (N v))` is equal to `N`.

Failure

If `M` is not of the specified form, or if `v` occurs free in `N`.

Example

```
> eta_conv (Term `\n. PRE n`);
val it = “PRE”: term
```

Comments

Eta-reduction embodies the principle of extensionality, which is basic to the HOL logic.

See also

`Drule.ETA_CONV`, `Drule.RIGHT_ETA`

etyvar (Type)

```
Type.etyvar : hol_type
```

Common type variable.

The ML variable `Type.etyvar` is bound to the type variable `'e`.

See also

`Type.alpha`, `Type.beta`, `Type.gamma`, `Type.delta`, `Type.ftyvar`, `Type.bool`

EVAL (bossLib)

```
bossLib.EVAL : conv
```

Evaluate a term by deduction.

An invocation `EVAL M` symbolically evaluates `M` by applying the defining equations of constants occurring in `M`. These equations are held in a mutable datastructure that is automatically added to by `Hol_datatype`, `Define`, and `tprove`. The underlying algorithm is call-by-value with a few differences, see the entry for `CBV_CONV` for details.

Failure

Never fails, but may diverge.

Example

```
> EVAL (Term `REVERSE (MAP (\x. x + a) [x;y;z])`);
val it = ⊢ REVERSE (MAP (λx. x + a) [x; y; z]) = [z + a; y + a; x + a]: thm
```

Comments

In order for recursive functions over numbers to be applied by `EVAL`, pattern matching over `SUC` and `0` needs to be replaced by destructors. For example, the equations for `FACT` would have to be rephrased as `FACT n = if n = 0 then 1 else n * FACT (n-1)`.

See also

`computeLib.CBV_CONV`, `computeLib.RESTR_EVAL_CONV`, `bossLib.EVAL_TAC`, `computeLib.monitoring`, `bossLib.Define`

EVAL_RULE (bossLib)

```
bossLib.EVAL_RULE : thm -> thm
```

Evaluate conclusion of a theorem.

An invocation `EVAL_RULE th` symbolically evaluates the conclusion of `th` by applying the defining equations of constants which occur in the conclusion of `th`. These equations are held in a mutable datastructure that is automatically added to by `Hol_datatype`, `Define`, and `tprove`. The underlying algorithm is call-by-value with a few differences, see the entry for `CBV_CONV` for details.

Failure

Never fails, but may diverge.

Example

```
> val th = ASSUME(Term `x = MAP FACT (REVERSE [1;2;3;4;5;6;7;8;9;10])`);
val th =  [.] ⊢ x = MAP FACT (REVERSE [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]): thm

> EVAL_RULE th;
val it =  [.] ⊢ x = [3628800; 362880; 40320; 5040; 720; 120; 24; 6; 2; 1]:
   thm

> hyp it;
val it = [“x = MAP FACT (REVERSE [1; 2; 3; 4; 5; 6; 7; 8; 9; 10])”]:
   term list
```

Comments

In order for recursive functions over numbers to be applied by EVAL_RULE, pattern matching over SUC and 0 needs to be replaced by destructors. For example, the equations for FACT would have to be rephrased as `FACT n = if n = 0 then 1 else n * FACT (n-1)`.

See also

bossLib.EVAL, bossLib.EVAL_TAC, computeLib.CBV_CONV

EVAL_TAC (bossLib)

```
bossLib.EVAL_TAC : tactic
```

Evaluate a goal deductively.

Applying EVAL_TAC to a goal `A ?- g` results in EVAL being applied to g to obtain `|- g = g'`. This theorem is used to transform the goal to `A ?- g'`.

The notion of evaluation is based around rules for replacing constants by their (equational) definitions. Thus `EVAL_TAC` is currently suited to evaluation of expressions that look like functional programs. Evaluation of inductive relations is not currently supported.

Failure

Shouldn't fail, but may diverge.

Example

`EVAL_TAC` reduces the goal `?- P (REVERSE (FLAT [[x; y]; [a; b; c; d]]))` to the goal

```
?- P [d; c; b; a; y; x]
```

Comments

The main problem with `EVAL_TAC` is knowing when it will terminate. One typical cause of non-termination is that a constant in the goal has not been added to `the_compset`. Another is that a test in a conditional in the expression may involve a variable.

Symbolic evaluation.

See also

`bossLib.EVAL`

EVALn (bossLib)

```
bossLib.EVALn : int -> conv
```

Evaluate a term by deduction, limiting number of steps taken.

An invocation `EVALn n M` symbolically evaluates `M` by applying the defining equations of constants occurring in `M`, stopping when `M` reduces to a normal form, or after `n` reduction steps have occurred. If `n` is large enough and there is a normal form, the behaviour will be the same as `EVAL M`, which see.

Failure

Never fails.

Example

In the example below, a custom pretty-printer hides a potentially large term involving the terms that are used to represent intermediates stages of numeral computation.

```
- EVALn 50 “MAP (\x. x * x) [1;2;3;4;5]”;
val it =
   ⊢ MAP (λx. x²) [1; 2; 3; 4; 5] =
     1::4:: <..num comp'n..> ::MAP (λx. x²) [4; 5]: thm
```

See also

computeLib.CBV_CONV, computeLib.RESTR_EVAL_CONV, bossLib.EVAL_TAC, computeLib.monitori
bossLib.Define

EVERY (Tactical)

```
Tactical.EVERY : (tactic list -> tactic)
```

Sequentially applies all the tactics in a given list of tactics.

When applied to a list of tactics [T1; ... ;Tn], and a goal g, the tactical EVERY applies each tactic in sequence to every subgoal generated by the previous one. This can be represented as:

```
EVERY [T1;...;Tn] = T1 THEN ... THEN Tn
```

If the tactic list is empty, the resulting tactic has no effect.

Failure

The application of EVERY to a tactic list never fails. The resulting tactic fails iff any of the component tactics do.

Comments

It is possible to use EVERY instead of THEN, but probably stylistically inferior. EVERY is more useful when applied to a list of tactics generated by a function.

See also

Tactical.FIRST, Tactical.MAP_EVERY, Tactical.THEN

EVERY_ASSUM (Tactical)

```
Tactical.EVERY_ASSUM : (thm_tactic -> tactic)
```

Sequentially applies all tactics given by mapping a function over the assumptions of a goal.

When applied to a theorem-tactic `f` and a goal `({A1,...,An} ?- C)`, the EVERY_ASSUM tactical maps `f` over a list of ASSUMEd assumptions then applies the resulting tactics, in sequence, to the goal:

```
EVERY_ASSUM f ({A1,...,An} ?- C)
 = (f(A1 |- A1) THEN ... THEN f(An |- An)) ({A1,...,An} ?- C)
```

If the goal has no assumptions, then EVERY_ASSUM has no effect.

Failure

The application of EVERY_ASSUM to a theorem-tactic and a goal fails if the theorem-tactic fails when applied to any of the ASSUMEd assumptions of the goal, or if any of the resulting tactics fail when applied sequentially.

See also

Tactical.ASSUM_LIST, Tactical.MAP_EVERY, Tactical.MAP_FIRST, Tactical.THEN

EVERY_CONJ_CONV (Conv)

```
Conv.EVERY_CONJ_CONV : conv -> conv
```

Applies a conversion to every top-level conjunct in a term.

The term `EVERY_CONJ_CONV c t` takes the conversion `c` and applies this to every top-level conjunct within term `t`. A top-level conjunct is a sub-term that can be reached from the root of the term by breaking apart only conjunctions. The terms affected by `c` are those that would be returned by a call to `strip_conj c`. In particular, if the term as a whole is not a conjunction, then the conversion will be applied to the whole term.

If the result of the application of the conversion to one of the conjuncts is one of the constants true or false, then one of two standard rewrites is applied, simplifying the resulting term. If one of the conjuncts is converted to false, then the conversion will not be applied to the remaining conjuncts (the conjuncts are worked on from left to right), and the result of the whole application will simply be false. Alternatively, conjuncts that are converted to true will not appear in the final result at all.

Failure

Fails if the conversion argument fails when applied to one of the top-level conjuncts in a term.

Example

```
> EVERY_CONJ_CONV BETA_CONV (Term`(\x. x /\ y) p`);
val it = ⊢ (λx. x ∧ y) p ⇔ p ∧ y: thm
> EVERY_CONJ_CONV BETA_CONV (Term`(\y. y /\ p) q /\ (\z. z) r`);
val it = ⊢ (λy. y ∧ p) q ∧ (λz. z) r ⇔ (q ∧ p) ∧ r: thm
```

Useful for applying a conversion to all of the “significant” sub-terms within a term without having to worry about the exact structure of its conjunctive skeleton.

See also

`Conv.EVERY_DISJ_CONV`, `Conv.RATOR_CONV`, `Conv.RAND_CONV`, `Conv.LAND_CONV`

EVERY_CONSEQ_CONV (ConseqConv)

```
ConseqConv.EVERY_CONSEQ_CONV : (conseq_conv list -> conseq_conv)
```

Applies in sequence all the consequence conversions in a given list of conversions.

See also

`ConseqConv.THEN_CONSEQ_CONV`, `Conv.EVERY_CONV`

EVERY_CONV (Conv)

```
Conv.EVERY_CONV : (conv list -> conv)
```

Applies in sequence all the conversions in a given list of conversions.

`EVERY_CONV [c1;...;cn] "t"` returns the result of applying the conversions `c1`, ..., `cn` in sequence to the term `"t"`. The conversions are applied in the order in which they are given in the list. In particular, if `ci "ti"` returns `|- ti=ti+1` for `i` from `1` to `n`, then `EVERY_CONV [c1;...;cn] "t1"` returns `|- t1=t(n+1)`. If the supplied list of conversions is empty, then `EVERY_CONV` returns the identity conversion. That is, `EVERY_CONV [] "t"` raises `UNCHANGED`, which indicates the result `|- t=t`.

Failure

`EVERY_CONV [c1;...;cn] "t"` fails if any one of the conversions `c1`, ..., `cn` fails (other than by raising `UNCHANGED`) when applied in sequence as specified above.

See also

`Conv.UNCHANGED`, `Conv.THENC`

EVERY_DISJ_CONV (Conv)

```
Conv.EVERY_DISJ_CONV : conv -> conv
```

Applies a conversion to every top-level disjunct in a term.

The term `EVERY_DISJ_CONV c t` takes the conversion `c` and applies this to every top-level disjunct within term `t`. A top-level disjunct is a sub-term that can be reached from the root of the term by breaking apart only disjunctions. The terms affected by `c` are those that would be returned by a call to `strip_disj c`. In particular, if the term as a whole is not a disjunction, then the conversion will be applied to the whole term.

If the result of the application of the conversion to one of the disjuncts is one of the constants true or false, then one of two standard rewrites is applied, simplifying the resulting term. If one of the disjuncts is converted to true, then the conversion will not be applied to the remaining disjuncts (the disjuncts are worked on from left to right), and the result of the whole application will simply be true. Alternatively, disjuncts that are converted to false will not appear in the final result at all.

Failure

Fails if the conversion argument fails when applied to one of the top-level disjuncts in the term.

Example

```
> EVERY_DISJ_CONV BETA_CONV
   (Term`(\x. x /\ p) q \/ (\x. x) r \/ (\y. s /\ y) u`);
val it = ⊢ (λx. x ∧ p) q ∨ (λx. x) r ∨ (λy. s ∧ y) u ⇔ q ∧ p ∨ r ∨ s ∧ u: thm
> EVERY_DISJ_CONV reduceLib.REDUCE_CONV ``3 < x \/ 2 < 3 \/ 2 EXP 1000 < 10``;
val it = ⊢ 3 < x ∨ 2 < 3 ∨ 2 ** 1000 < 10 ⇔ T: thm
```

Useful for applying a conversion to all of the “significant” sub-terms within a term without having to worry about the exact structure of its disjunctive skeleton.

See also

Conv.EVERY_CONJ_CONV, Conv.RATOR_CONV, Conv.RAND_CONV, Conv.LAND_CONV, numLib.REDUCE_CONV

EVERY_LT (Tactical)

```
Tactical.EVERY_LT : (list_tactic list -> list_tactic)
```

Sequentially applies all the list-tactics in a given list of list-tactics.

When applied to a list of list-tactics `[LT1; ... ;LTn]`, and a goal g, the tactical `EVERY_LT` applies each list-tactic in sequence to every subgoal generated by the previous one. This can be represented as:

```
EVERY_LT [LT1;...;LTn] = LT1 THEN_LT ... THEN_LT LTn
```

If the list-tactic list is empty, the resulting list-tactic has no effect.

Failure

The application of `EVERY_LT` to a list-tactic list never fails. The resulting list-tactic fails iff any of the component list-tactics do.

Comments

It is possible to use `EVERY_LT` instead of `THEN_LT`, but probably stylistically inferior. `EVERY_LT` is more useful when applied to a list of list-tactics generated by a function.

See also

Tactical.THEN_LT

EVERY_TCL (Thm_cont)

```
Thm_cont.EVERY_TCL : (thm_tactical list -> thm_tactical)
```

Composes a list of theorem-tacticals.

When given a list of theorem-tacticals and a theorem, EVERY_TCL simply composes their effects on the theorem. The effect is:

```
EVERY_TCL [ttl1;...;ttln] = ttl1 THEN_TCL ... THEN_TCL ttln
```

In other words, if:

```
ttl1 ttac th1 = ttac th2  ...  ttln ttac thn = ttac thn'
```

then:

```
EVERY_TCL [ttl1;...;ttln] ttac th1 = ttac thn'
```

If the theorem-tactical list is empty, the resulting theorem-tactical behaves in the same way as ALL_THEN, the identity theorem-tactical.

Failure

The application to a list of theorem-tacticals never fails.

See also

Thm_cont.FIRST_TCL, Thm_cont.ORELSE_TCL, Thm_cont.REPEAT_TCL, Thm_cont.THEN_TCL

exclude_ssfrags (simpLib)

```
simpLib.exclude_ssfrags : string list -> simpset -> simpset
```

Removes named simpset fragments and records the exclusion in the simpset.

A call to `exclude_ssfrags fragnames simpset` returns a simpset that is the same as `simpset` except that 1. every fragment with a name in `fragnames` has been removed

from the simpset's history (as with `remove_ssfrags`); and 2. the names in `fragnames` are recorded in the simpset's *excluded* set, so that any subsequent attempt to add a fragment with one of those names via the ++ operator is a silent no-op.

Use `force_add` (or, in user-facing thm-list arguments to simplification tactics, the `SF` marker) to opt back in: it adds a fragment and removes its name from the excluded set in one step.

This is the function used to implement the `Proof[exclude_frags = ...]` attribute on theorem proofs.

Failure

Never fails (unlike `remove_ssfrags`, no `Conv.UNCHANGED` is raised when none of the names match a fragment in the simpset).

Example

```
> val base = simpLib.exclude_ssfrags ["ARITH"] (srw_ss());
val base =
   Included fragments (with 1 anonymous fragment [remove using name ""]):
      ABBREV, ARITH_RWTS, ASCIInumbers, BOOL, COMBIN, CONG, ConseqConv,
      Datatype bool$itself, Datatype cv$cv, Datatype example$atree,
      Datatype example$example, Datatype fcp$bit0, Datatype fcp$bit1,
      Datatype fcp$cart, Datatype foo$exp, Datatype foo$foo,
      Datatype foo$point, Datatype integer$int, Datatype list$list,
      Datatype min$bool, Datatype min$fun, Datatype num$num,
      Datatype one$one, Datatype option$option, Datatype pair$prod,
      Datatype patricia$ptree, Datatype string$char, Datatype sum$sum,
      Datatype ternaryComparisons$ordering, GSPEC_SIMP, LABEL_CONG, MOD,
      NORMEQ, NOT, Omega, PURE, REAL_REDUCE, RMULCANON, RMULRELNORM,
      SET_SPEC, UNWIND, While, arithmetic, bag, basicSize, bit, bitstring,
      blast, bool, combin, cooper, cv, cv_prim, cv_rep, cv_type, divides,
      fcp, frac, gcd, hide, hol, hrat, hreal, ind_type, indexedLists,
      intExtension, intReduce, int_arith, integer, integer_word, iterate,
      list, list EQ, logroot, marker, normalForms, normalizer, num, numeral,
      numeral_bit, numpair, numposrep, one, option, pair, patricia,
      patternMatches, patternMatchesSimp, permutes, pred_set, prim_rec,
      primeFactor, quantHeuristics, quotient, rat, real, real_arith, realax,
      reduce, relation, rich_list, sat, sizes, sorting, string, sum, sum_num,
      ternaryComparisons, word arith, word ground, word logic, word shift,
```

```
      word subtract, words
   Rewrites (with 689 anonymous rewrites)
   Other net names/keys:
      .rewrite:ADD_0'.1, .rewrite:ADD_INV_0_EQ'.1, .rewrite:ADD_INV_0_EQ'.2,
      .rewrite:ADD_MONO_LESS_EQ.1, .rewrite:ADD_SUB'.1,
      .rewrite:COND_BOOL_CLAUSES.1, .rewrite:COND_BOOL_CLAUSES.2,
      .rewrite:COND_BOOL_CLAUSES.3, .rewrite:COND_BOOL_CLAUSES.4,
      .rewrite:EQ_MONO_ADD_EQ.1, .rewrite:EXCLUDED_MIDDLE'.1,
[...Output elided...]

> List.exists (fn n => n = "ARITH")
              (simpLib.ssfrag_names_of (base ++ numSimps.ARITH_ss));
val it = false: bool


> List.exists (fn n => n = "ARITH")
              (simpLib.ssfrag_names_of
                 (simpLib.force_add base numSimps.ARITH_ss));
val it = true: bool
```

See also

simpLib.remove_ssfrags, simpLib.force_add, bossLib.SF

EXISTENCE (Conv)

```
Conv.EXISTENCE : (thm -> thm)
```

Deduces existence from unique existence.

When applied to a theorem with a unique-existentially quantified conclusion, EXISTENCE returns the same theorem with normal existential quantification over the same variable.

```
    A |- ?!x. p
   -------------  EXISTENCE
    A |- ?x. p
```

Failure

Fails unless the conclusion of the theorem is unique-existentially quantified.

See also

`Conv.EXISTS_UNIQUE_CONV`

existential (boolSyntax)

```
boolSyntax.existential : term
```

Constant denoting existential quantification.

The ML variable `boolSyntax.existential` is bound to the term `bool$?`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.negation`, `boolSyntax.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`

exists (Lib)

```
Lib.exists : ('a -> bool) -> 'a list -> bool
```

Check if a predicate holds somewhere in a list.

An invocation `exists P l` returns true if `P` holds of some element of `l`. Since there are no elements of `[]`, `exists P []` always returns `false`.

Failure

When searching for an element of `l` that `P` holds of, it may happen that an application of `P` to an element of `l` raises an exception. In that case, `exists P l` raises an exception.

Example

```
> exists (fn i => i mod 2 = 0) [1,3,4];
val it = true: bool

> exists (fn _ => raise Fail "") [];
val it = false: bool

> exists (fn _ => raise Fail "") [1];
Exception- Fail "" raised
```

See also

`Lib.all`, `Lib.first`, `Lib.tryfind`

EXISTS (Thm)

```
Thm.EXISTS : term * term -> thm -> thm
```

Introduces existential quantification given a particular witness.

When applied to a pair of terms and a theorem, the first term an existentially quantified pattern indicating the desired form of the result, and the second a witness whose substitution for the quantified variable gives a term which is the same as the conclusion of the theorem, `EXISTS` gives the desired theorem.

```
   A |- p[u/x]
  -------------  EXISTS (?x. p, u)
   A |- ?x. p
```

Failure

Fails unless the substituted pattern is the same as the conclusion of the theorem.

Example

The following examples illustrate how it is possible to deduce different things from the same theorem:

```
- EXISTS (Term `?x. x=T`,T) (REFL T);
> val it = |- ?x. x = T : thm

- EXISTS (Term `?x:bool. x=x`,T) (REFL T);
> val it = |- ?x. x = x : thm
```

See also

`Thm.CHOOSE`, `Drule.SIMPLE_EXISTS`, `Tactic.EXISTS_TAC`

exists1 (boolSyntax)

```
boolSyntax.exists1 : term
```

Constant denoting the unique existence quantifier.

The ML variable `boolSyntax.exists1` is bound to the term `bool$?!`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.negation`, `boolSyntax.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`

EXISTS_AND_CONV (Conv)

```
Conv.EXISTS_AND_CONV : conv
```

Moves an existential quantification inwards through a conjunction.

When applied to a term of the form `?x. P /\ Q`, where `x` is not free in both `P` and `Q`, `EXISTS_AND_CONV` returns a theorem of one of three forms, depending on occurrences of the variable `x` in `P` and `Q`. If `x` is free in `P` but not in `Q`, then the theorem:

```
|- (?x. P /\ Q) = (?x.P) /\ Q
```

is returned. If `x` is free in `Q` but not in `P`, then the result is:

```
|- (?x. P /\ Q) = P /\ (?x.Q)
```

And if `x` is free in neither `P` nor `Q`, then the result is:

```
|- (?x. P /\ Q) = (?x.P) /\ (?x.Q)
```

Failure

`EXISTS_AND_CONV` fails if it is applied to a term not of the form `?x. P /\ Q`, or if it is applied to a term `?x. P /\ Q` in which the variable `x` is free in both `P` and `Q`.

See also

`Conv.AND_EXISTS_CONV`, `Conv.EXISTS_AND_REORDER_CONV`, `Conv.LEFT_AND_EXISTS_CONV`, `Conv.RIGHT_AND_EXISTS_CONV`

EXISTS_AND_REORDER_CONV (Conv)

```
Conv.EXISTS_AND_REORDER_CONV : conv
```

Moves an existential quantification inwards through a conjunction, sorting the body.

When applied to a term of the form `?x. c1 /\ c2 /\ .. /\ cn`, where `x` is not free in at least one of the conjuncts `ci`, then `EXISTS_AND_REORDER_CONV` returns a theorem of the form

```
|- (?x. ...) = (ci /\ cj /\ ck /\ ...) /\ (?x. cm /\ cn /\ cp /\ ...)
```

where the conjuncts `ci`, `cj` and `ck` do not have the bound variable `x` free, and where the conjuncts `cm`, `cn` and `cp` do.

Failure

EXISTS_AND_REORDER_CONV fails if it is applied to a term that is not an existential. It raises UNCHANGED if the existential's body is not a conjunction, or if the body does not have any conjuncts where the bound variable does not occur, or if none of the body's conjuncts have free occurrences of the bound variable.

Comments

The conjuncts in the resulting term are kept in the same relative order as in the input term, but will all be right-associated in the two groups (because they are re-assembled with list_mk_conj), possibly destroying structure that existed in the original.

See also

Conv.EXISTS_AND_CONV

EXISTS_ARITH_CONV (Arith)

Arith.EXISTS_ARITH_CONV : conv

Partial decision procedure for non-universal Presburger natural arithmetic.

EXISTS_ARITH_CONV is a partial decision procedure for formulae of Presburger natural arithmetic which are in prenex normal form and have all variables existentially quantified. Presburger natural arithmetic is the subset of arithmetic formulae made up from natural number constants, numeric variables, addition, multiplication by a constant, the relations <, <=, =, >=, > and the logical connectives ~, /\, \/, ==>, = (if-and-only-if), ! ('forall') and ? ('there exists'). Products of two expressions which both contain variables are not included in the subset, but the function SUC which is not normally included in a specification of Presburger arithmetic is allowed in this HOL implementation.

Given a formula in the specified subset, the function attempts to prove that it is equal to T (true). The procedure is incomplete; it is not able to prove all formulae in the subset.

Failure

The function can fail in two ways. It fails if the argument term is not a formula in the specified subset, and it also fails if it is unable to prove the formula. The failure strings are different in each case.

Example

```
#EXISTS_ARITH_CONV "?m n. m < n";;
|- (?m n. m < n) = T

#EXISTS_ARITH_CONV "?m n. (2 * m) + (3 * n) = 10";;
|- (?m n. (2 * m) + (3 * n) = 10) = T
```

See also

`Arith.NEGATE_CONV`, `Arith.FORALL_ARITH_CONV`, `numLib.ARITH_CONV`

EXISTS_CONSEQ_CONV (ConseqConv)

```
ConseqConv.EXISTS_CONSEQ_CONV : (conseq_conv -> conseq_conv)
```

Applies a consequence conversion to the body of an existentially quantified term.

If `c` is a consequence conversion that maps a term ``` ``t x`` ``` to a theorem `|- t x = t' x`, `|- t' x ==> t x` or `|- t x ==> t' x`, then `EXISTS_CONSEQ_CONV c` maps ``` ``?x. t x`` ``` to `|- ?x. t x = ?x. t' x`, `|- ?x. t' x ==> ?x. t x` or `|- ?x. t x ==> ?x. t' x`, respectively.

Failure

`EXISTS_CONSEQ_CONV c t` fails, if `t` is not an existentially quantified term or if `c` fails on the body of `t`.

See also

`Conv.QUANT_CONV`, `ConseqConv.FORALL_CONSEQ_CONV`, `ConseqConv.QUANT_CONSEQ_CONV`

EXISTS_DEL1_CONV (unwindLib)

`unwindLib.EXISTS_DEL1_CONV : conv`

Deletes one existential quantifier.

`EXISTS_DEL1_CONV "?x. t"` returns the theorem:

```
|- (?x. t) = t
```

provided `x` is not free in `t`.

Failure

Fails if the argument term is not an existential quantification or if `x` is free in `t`.

See also

`unwindLib.EXISTS_DEL_CONV`, `unwindLib.PRUNE_ONCE_CONV`

EXISTS_DEL_CONV (unwindLib)

`unwindLib.EXISTS_DEL_CONV : conv`

Deletes existential quantifiers.

`EXISTS_DEL_CONV "?x1 ... xn. t"` returns the theorem:

```
|- (?x1 ... xn. t) = t
```

provided `x1`,...,`xn` are not free in `t`.

Failure

Fails if any of the x's appear free in t. The function does not perform a partial deletion; for example, if x1 and x2 do not appear free in t but x3 does, the function will fail; it will not return:

```
|- ?x1 ... xn. t = ?x3 ... xn. t
```

See also

unwindLib.EXISTS_DEL1_CONV, unwindLib.PRUNE_CONV

EXISTS_EQ (Drule)

```
Drule.EXISTS_EQ : (term -> thm -> thm)
```

Existentially quantifies both sides of an equational theorem.

When applied to a variable x and a theorem whose conclusion is equational, A |- t1 = t2, the inference rule EXISTS_EQ returns the theorem A |- (?x. t1) = (?x. t2), provided the variable x is not free in any of the assumptions.

```
        A |- t1 = t2
------------------------  EXISTS_EQ "x"      [where x is not free in A]
 A |- (?x.t1) = (?x.t2)
```

Failure

Fails unless the theorem is equational with both sides having type bool, or if the term is not a variable, or if the variable to be quantified over is free in any of the assumptions.

See also

Thm.AP_TERM, Drule.EXISTS_IMP, Drule.FORALL_EQ, Drule.MK_EXISTS, Drule.SELECT_-EQ

EXISTS_EQ___CONSEQ_CONV (ConseqConv)

```
ConseqConv.EXISTS_EQ___CONSEQ_CONV : conseq_conv
```

Given a term of the form `(?x. P x) = (?x. Q x)` this consequence conversion returns the theorem `|- (!x. (P x = Q x)) ==> ((?x. P x) = (?x. Q x))`.

See also

`ConseqConv.conseq_conv`

EXISTS_EQN_CONV (unwindLib)

```
unwindLib.EXISTS_EQN_CONV : conv
```

Proves the existence of a line that has a non-recursive equation.
`EXISTS_EQN_CONV "?l. !y1 ... ym. l x1 ... xn = t"` returns the theorem:

```
|- (?l. !y1 ... ym. l x1 ... xn = t) = T
```

provided `l` is not free in `t`. Both `m` and `n` may be zero.

Failure

Fails if the argument term is not of the specified form or if `l` appears free in `t`.

See also

`unwindLib.PRUNE_ONCE_CONV`

EXISTS_IMP (Drule)

```
Drule.EXISTS_IMP : (term -> thm -> thm)
```

Existentially quantifies both the antecedent and consequent of an implication.

When applied to a variable `x` and a theorem `A |- t1 ==> t2`, the inference rule `EXISTS_IMP` returns the theorem `A |- (?x. t1) ==> (?x. t2)`, provided `x` is not free in the assumptions.

```
        A |- t1 ==> t2
--------------------------  EXISTS_IMP "x"   [where x is not free in A]
 A |- (?x.t1) ==> (?x.t2)
```

Failure

Fails if the theorem is not implicative, or if the term is not a variable, or if the term is a variable but is free in the assumption list.

See also

`Drule.EXISTS_EQ`

EXISTS_IMP_CONV (Conv)

```
Conv.EXISTS_IMP_CONV : conv
```

Moves an existential quantification inwards through an implication.

When applied to a term of the form `?x. P ==> Q`, where `x` is not free in both `P` and `Q`, `EXISTS_IMP_CONV` returns a theorem of one of three forms, depending on occurrences of the variable `x` in `P` and `Q`. If `x` is free in `P` but not in `Q`, then the theorem:

```
|- (?x. P ==> Q) = (!x.P) ==> Q
```

is returned. If `x` is free in `Q` but not in `P`, then the result is:

```
|- (?x. P ==> Q) = P ==> (?x.Q)
```

And if `x` is free in neither `P` nor `Q`, then the result is:

```
|- (?x. P ==> Q) = (!x.P) ==> (?x.Q)
```

Failure

`EXISTS_IMP_CONV` fails if it is applied to a term not of the form `?x. P ==> Q`, or if it is applied to a term `?x. P ==> Q` in which the variable `x` is free in both `P` and `Q`.

See also

`Conv.LEFT_IMP_FORALL_CONV`, `Conv.RIGHT_IMP_EXISTS_CONV`

EXISTS_INTRO_IMP (ConseqConv)

```
ConseqConv.EXISTS_INTRO_IMP : term -> thm -> thm
```

Existentially quantifies both sides of an implication in the conclusion of a theorem.

When applied to a term `x` and a theorem `A |- t1 ==> t2`, the inference rule `EXISTS_INTRO_IMP` returns the theorem `A |- (?x. t1) ==> (?x. t2)`, provided `x` is a variable not free in any of the assumptions. There is no compulsion that `x` should be free in `t1` or `t2`.

```
        A |- (t1 ==> t2)
---------------------------     EXISTS_INTRO_IMP x      [where x is not free in A]
 A |- (?x. t1) ==> (?x. t2)
```

Failure

Fails if `x` is not a variable, the conclusion of the theorem is not an implication, or if `x` is free in any of the assumptions.

Example

```
- val thm0 = mk_thm ([], Term `P (x:'a) ==> Q x`);
> val thm0 =  |- P (x :'a) ==> Q x : thm

- val thm1 = EXISTS_INTRO_IMP (Term `x:'a`) thm0;
> val thm1 =  |- (?x. P x) ==> (?x. Q x)
```

See also

Thm.GEN, ConseqConv.GEN_IMP

EXISTS_LEFT (Drule)

```
Drule.EXISTS_LEFT : term list -> thm -> thm
```

Existentially quantifes hypotheses of a theorem.

In this example, assume that h1 and h3 (only) involve the free variable x.

```
   h1, h2, h3 |- t
-------------------- EXISTS_LEFT [``x``]
?x. h1 /\ h3, h2 |- t
```

Failure

EXISTS_LEFT will fail if the term list supplied does not consist only of free variables

Example

Where th is [p, q, g x, h y, f x y] |- r, and fvx and fvy are ``x`` and ``y``,

EXISTS_LEFT [fvx, fvy] th is [p, q, ?y. (?x. g x /\ f x y) /\ h y] |- r

EXISTS_LEFT [fvy, fvx] th is [p, q, ?x. (?y. h y /\ f x y) /\ g x] |- r

Where EQ_TRANS is [] |- !x y z. (x = y) /\ (y = z) ==> (x = z) and the current goal is a = b, the tactic MATCH_MP_TAC EQ_TRANS gives a new goal ?y. (a = y) /\ (y = b) by virtue of the smart features built into MATCH_MP_TAC.

Where `trans_thm` is `[] |- !x y z. (x = y) ==> (y = z) ==> (x = z)` the same result could of course be achieved by rewriting it with `AND_IMP_INTRO`. But more generally, `EXISTS_LEFT` could be used as a building-block for a more flexible tactic. In this instance, one might start with

```
val trans_thm_h = UNDISCH_ALL (SPEC_ALL trans_thm) ;
EXISTS_LEFT (thm_frees trans_thm_h) trans_thm_h ;
```

giving `[?y. (x = y) /\ (y = z)] |- x = z`

See also

`Drule.EXISTS_LEFT1`, `Thm.CHOOSE`, `Thm.EXISTS`, `Tactic.CHOOSE_TAC`, `Tactic.EXISTS_TAC`

EXISTS_LEFT1 (Drule)

```
Drule.EXISTS_LEFT1 : term -> thm -> thm
```

Existentially quantifes hypotheses of a theorem.

In this example, assume that `h1` and `h3` (only) involve the free variable `x`.

```
   h1, h2, h3 |- t
--------------------- EXISTS_LEFT1 ``x``
?x. h1 /\ h3, h2 |- t
```

Failure

`EXISTS_LEFT1` will fail unless the term supplied is a free variable which appears in one or more hypotheses but not the conclusion of the given theorem

Example

Where `th` is `[p, q, g x, h y, f x y] |- r`, and `fvx` and `fvy` are ``x`` and ``y``,

`EXISTS_LEFT1 fvx th` is `[p, q, h y, ?x. g x /\ f x y] |- r`

`EXISTS_LEFT1 fvy th` is `[p, q, g x, ?y. h y /\ f x y] |- r`

Comments

EXISTS_LEFT1 fv is just like EXISTS_LEFT [fv] except that EXISTS_LEFT1 fv fails where EXISTS_LEFT [fv] returns the theorem unchanged

See EXISTS_LEFT for further discussion

See also

Drule.EXISTS_LEFT, Thm.CHOOSE, Thm.EXISTS, Tactic.CHOOSE_TAC, Tactic.EXISTS_-TAC

EXISTS_NOT_CONV (Conv)

```
Conv.EXISTS_NOT_CONV : conv
```

Moves an existential quantification inwards through a negation.

When applied to a term of the form ?x.~P, the conversion EXISTS_NOT_CONV returns the theorem:

```
|- (?x.~P) = ~(!x. P)
```

Failure

Fails if applied to a term not of the form ?x.~P.

See also

Conv.FORALL_NOT_CONV, Conv.NOT_EXISTS_CONV, Conv.NOT_FORALL_CONV

EXISTS_OR_CONV (Conv)

```
Conv.EXISTS_OR_CONV : conv
```

Moves an existential quantification inwards through a disjunction.

When applied to a term of the form `?x. P \/ Q`, the conversion `EXISTS_OR_CONV` returns the theorem:

```
|- (?x. P \/ Q) = (?x.P) \/ (?x.Q)
```

Failure

Fails if applied to a term not of the form `?x. P \/ Q`.

See also

`Conv.OR_EXISTS_CONV`, `Conv.LEFT_OR_EXISTS_CONV`, `Conv.RIGHT_OR_EXISTS_CONV`

EXISTS_TAC (Tactic)

```
Tactic.EXISTS_TAC : (term -> tactic)
```

Reduces existentially quantified goal to one involving a specific witness.

When applied to a term `u` and a goal `?x. t`, the tactic `EXISTS_TAC` reduces the goal to `t[u/x]` (substituting `u` for all free instances of `x` in `t`, with variable renaming if necessary to avoid free variable capture).

```
 A ?- ?x. t
============  EXISTS_TAC "u"
 A ?- t[u/x]
```

Failure

Fails unless the goal's conclusion is existentially quantified and the term supplied has the same type as the quantified variable in the goal.

Example

The goal:

```
?- ?x. x=T
```

can be solved by:

```
EXISTS_TAC ``T`` THEN REFL_TAC
```

See also

`Thm.EXISTS`

exists_tyvar (Type)

```
Type.exists_tyvar : (hol_type -> bool) -> hol_type -> bool
```

Checks if a type variable satisfying a given condition exists in a type.

An invocation `exists_tyvar P ty` searches `ty` for a type variable satisfying the predicate `P`. The value `true` is returned if the search is successful; otherwise `false` is the result.

Failure

If `P` fails when applied to a type variable encountered in the course of searching `ty`.

Example

```
> exists_tyvar (equal beta) (alpha --> beta --> bool);
val it = true: bool
```

Comments

This function is more efficient, in some cases, than `exists P o type_vars`.

EXISTS_UNIQUE_CONV (Conv)

```
Conv.EXISTS_UNIQUE_CONV : conv
```

Expands with the definition of unique existence.

Given a term of the form `"?!x.P[x]"`, the conversion `EXISTS_UNIQUE_CONV` proves that this assertion is equivalent to the conjunction of two statements, namely that there exists at least one value `x` such that `P[x]`, and that there is at most one value `x` for which `P[x]` holds. The theorem returned is:

```
|- (?! x. P[x]) = (?x. P[x]) /\ (!x x'. P[x] /\ P[x'] ==> (x = x'))
```

where `x'` is a primed variant of `x` that does not appear free in the input term. Note that the quantified variable `x` need not in fact appear free in the body of the input term. For example, `EXISTS_UNIQUE_CONV "?!x.T"` returns the theorem:

```
|- (?! x. T) = (?x. T) /\ (!x x'. T /\ T ==> (x = x'))
```

Failure

`EXISTS_UNIQUE_CONV tm` fails if `tm` does not have the form `"?!x.P"`.

See also

`Conv.EXISTENCE`

exn_to_string (Feedback)

```
Feedback.exn_to_string : exn -> string
```

Map an exception into a string.

The function `exn_to_string` maps an exception to a string. However, in the case of the `Interrupt` exception, it is not mapped to a string, but is instead raised. This avoids the possibility of suppressing the propagation of `Interrupt` to the top level.

Failure

Never fails.

Example

```
> exn_to_string Interrupt;
Exception- Interrupt raised

> exn_to_string Div;
val it = "Div": string

> exn_to_string (mk_HOL_ERR "Foo" "bar" "incomprehensible input");
val it = "\nException raised at Foo.bar: incomprehensible input\n": string
```

See also

Feedback, Feedback.HOL_ERR, Feedback.ERR_to_string

EXP_CONV (reduceLib)

```
reduceLib.EXP_CONV : conv
```

Calculates by inference the result of raising one numeral to the power of another.

If m and n are numerals (e.g. 0, 1, 2, 3,...), then EXP_CONV "m EXP n" returns the theorem:

```
|- m EXP n = s
```

where s is the numeral that denotes the result of raising the natural number denoted by m to the power of the natural number denoted by n.

Failure

EXP_CONV tm fails unless tm is of the form "m EXP n", where m and n are numerals.

Example

```
#EXP_CONV "0 EXP 0";;
|- 0 EXP 0 = 1

#EXP_CONV "15 EXP 0";;
|- 15 EXP 0 = 1

#EXP_CONV "12 EXP 1";;
|- 12 EXP 1 = 12

#EXP_CONV "2 EXP 6";;
|- 2 EXP 6 = 64
```

expand (proofManagerLib)

```
proofManagerLib.expand : tactic -> proof
```

Applies a tactic to the current goal, stacking the resulting subgoals.

The function `expand` is part of the subgoal package. It may be abbreviated by the function `e`. It applies a tactic to the current goal to give a new proof state. The previous state is stored on the backup list. If the tactic produces subgoals, the new proof state is formed from the old one by removing the current goal from the goal stack and adding a new level consisting of its subgoals. The corresponding justification is placed on the justification stack. The new subgoals are printed. If more than one subgoal is produced, they are printed from the bottom of the stack so that the new current goal is printed last.

If a tactic solves the current goal (returns an empty subgoal list), then its justification is used to prove a corresponding theorem. This theorem is incorporated into the justification of the parent goal and printed. If the subgoal was the last subgoal of the level, the level is removed and the parent goal is proved using its (new) justification. This process is repeated until a level with unproven subgoals is reached. The next goal on the goal stack then becomes the current goal. This goal is printed. If all the subgoals are proved, the resulting proof state consists of the theorem proved by the justifications.

The tactic applied is a validating version of the tactic given. It ensures that the justification of the tactic does provide a proof of the goal from the subgoals generated by the tactic. It will cause failure if this is not so. The tactical `VALID` performs this validation.

For a description of the subgoal package, see `set_goal`.

Failure

`expand tac` fails if the tactic `tac` fails for the top goal. It will diverge if the tactic diverges for the goal. It will fail if there are no unproven goals. This could be because no goal has been set using `set_goal` or because the last goal set has been completely proved. It will also fail in cases when the tactic is invalid.

Example

```
> g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

> expand CONJ_TAC;
OK..
2 subgoals:
val it =

   TL [1; 2; 3] = [2; 3]

   HD [1; 2; 3] = 1

> expand (REWRITE_TAC[listTheory.HD]);
OK..

Goal proved.
⊢ HD [1; 2; 3] = 1

Remaining subgoals:
val it =

   TL [1; 2; 3] = [2; 3]

> expand (REWRITE_TAC[listTheory.TL]);
```

```
OK..

Goal proved.
⊢ TL [1; 2; 3] = [2; 3]
val it =
   Initial goal proved.
   ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]: proof
```

In the following example an invalid tactic is used. It is invalid because it assumes something that is not on the assumption list of the goal. The justification adds this assumption to the assumption list so the justification would not prove the goal that was set.

```
> g `1=2`;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        1 = 2
> expand (REWRITE_TAC[ASSUME (Term `1=2`)]);
Exception- OK..
HOL_ERR
  (at Tactical.VALID: Invalid tactic: theorem has bad hypothesis 1 = 2) raised
```

Note that an invalid tactic may "succeed". Thus, where `tac1` is invalid, and `tac2` is valid (and both succeed), `FIRST [tac1, tac2]` is invalid. For example, where theorem `uth` is `[p] |- q` and `uth'` is `[p'] |- q`

```
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
         0.  p
        ------------------------------------
             q

> e (FIRST (map ACCEPT_TAC [uth', uth])) ;
Exception- OK..
HOL_ERR (at Tactical.VALID: Invalid tactic: theorem has bad hypothesis p') raised
```

```
> e (FIRST (map (VALID o ACCEPT_TAC) [uth', uth])) ;
OK..
val it =
   Initial goal proved.
    [.] ⊢ q: proof
```

Doing a step in an interactive goal-directed proof.

See also

proofManagerLib.set_goal, proofManagerLib.restart, proofManagerLib.backup, proofManagerLib.redo, proofManagerLib.restore, proofManagerLib.save, proofManagerLib.s backup, proofManagerLib.expand, proofManagerLib.expandf, proofManagerLib.flatn, proofManagerLib.p, proofManagerLib.top_thm, proofManagerLib.top_goal, Tactical.VALID, Tactical.VALIDATE

EXPAND_ALL_BUT_CONV (unwindLib)

```
unwindLib.EXPAND_ALL_BUT_CONV : (string list -> thm list -> conv)
```

Unfolds, then unwinds all lines (except those specified) as much as possible, then prunes the unwound lines.

EXPAND_ALL_BUT_CONV [`li(k+1)`;...;`lim`] thl when applied to the following term:

```
"?l1 ... lm. t1 /\ ... /\ ui1 /\ ... /\ uik /\ ... /\ tn"
```

returns a theorem of the form:

```
B |- (?l1 ... lm. t1 /\ ... /\ ui1 /\ ... /\ uik /\ ... /\ tn) =
     (?li(k+1) ... lim. t1' /\ ... /\ tn')
```

where each ti' is the result of rewriting ti with the theorems in thl. The set of assumptions B is the union of the instantiated assumptions of the theorems used for rewriting. If none of the rewrites are applicable to a conjunct, it is unchanged. Those

conjuncts that after rewriting are equations for the lines `li1,...,lik` (they are denoted by `ui1,...,uik`) are used to unwind and the lines `li1,...,lik` are then pruned.

The `li`'s are related by the equation:

```
{{li1,...,lik}} u {{li(k+1),...,lim}} = {{l1,...,lm}}
```

Failure

The function may fail if the argument term is not of the specified form. It will also fail if the unwound lines cannot be pruned. It is possible for the function to attempt unwinding indefinitely (to loop).

Example

```
#EXPAND_ALL_BUT_CONV [`l1`]
# [ASSUME "!in out. INV (in,out) = !(t:num). out t = ~(in t)"]
# "?l1 l2.
#   INV (l1,l2) /\ INV (l2,out) /\ (!(t:num). l1 t = l2 (t-1) \/ out (t-1))";;
. |- (?l1 l2.
       INV(l1,l2) /\ INV(l2,out) /\ (!t. l1 t = l2(t - 1) \/ out(t - 1))) =
     (?l1.
       (!t. out t = ~~l1 t) /\ (!t. l1 t = ~l1(t - 1) \/ ~~l1(t - 1)))
```

See also

`unwindLib.EXPAND_AUTO_CONV`, `unwindLib.EXPAND_ALL_BUT_RIGHT_RULE`, `unwindLib.EXPAND_AUTO_RIGHT_RULE`, `unwindLib.UNFOLD_CONV`, `unwindLib.UNWIND_ALL_BUT_CONV`, `unwindLib.PRUNE_SOME_CONV`

EXPAND_ALL_BUT_RIGHT_RULE (unwindLib)

```
unwindLib.EXPAND_ALL_BUT_RIGHT_RULE : (string list -> thm list -> thm -> thm)
```

Unfolds, then unwinds all lines (except those specified) as much as possible, then prunes the unwound lines.

`EXPAND_ALL_BUT_RIGHT_RULE [`li(k+1)`;...;`lim`] thl` behaves as follows:

```
    A |- !z1 ... zr.
          t = ?l1 ... lm. t1 /\ ... /\ ui1 /\ ... /\ uik /\ ... /\ tn
   ------------------------------------------------------------------
       B u A |- !z1 ... zr. t = ?li(k+1) ... lim. t1' /\ ... /\ tn'
```

where each `ti'` is the result of rewriting `ti` with the theorems in `thl`. The set of assumptions `B` is the union of the instantiated assumptions of the theorems used for rewriting. If none of the rewrites are applicable to a conjunct, it is unchanged. Those conjuncts that after rewriting are equations for the lines `li1,...,lik` (they are denoted by `ui1,...,uik`) are used to unwind and the lines `li1,...,lik` are then pruned.

The `li`'s are related by the equation:

```
{{li1,...,lik}} u {{li(k+1),...,lim}} = {{l1,...,lm}}
```

Failure

The function may fail if the argument theorem is not of the specified form. It will also fail if the unwound lines cannot be pruned. It is possible for the function to attempt unwinding indefinitely (to loop).

Example

```
#EXPAND_ALL_BUT_RIGHT_RULE [`l1`]
# [ASSUME "!in out. INV (in,out) = !(t:num). out t = ~(in t)"]
# (ASSUME
#   "!(in:num->bool) out.
#     DEV(in,out) =
#      ?l1 l2.
#       INV (l1,l2) /\ INV (l2,out) /\ (!(t:num). l1 t = in t \/ out (t-1))");;
.. |- !in out.
       DEV(in,out) =
       (?l1. (!t. out t = ~~l1 t) /\ (!t. l1 t = in t \/ ~~l1(t - 1)))
```

See also

`unwindLib.EXPAND_AUTO_RIGHT_RULE`, `unwindLib.EXPAND_ALL_BUT_CONV`, `unwindLib.EXPAND_AUTO_CONV`, `unwindLib.UNFOLD_RIGHT_RULE`, `unwindLib.UNWIND_ALL_BUT_RIGHT_RULE`, `unwindLib.PRUNE_SOME_RIGHT_RULE`

EXPAND_AUTO_CONV (unwindLib)

```
unwindLib.EXPAND_AUTO_CONV : (thm list -> conv)
```

Unfolds, then unwinds as much as possible, then prunes the unwound lines.

`EXPAND_AUTO_CONV thl` when applied to the following term:

```
"?l1 ... lm. t1 /\ ... /\ ui1 /\ ... /\ uik /\ ... /\ tn"
```

returns a theorem of the form:

```
B |- (?l1 ... lm. t1 /\ ... /\ ui1 /\ ... /\ uik /\ ... /\ tn) =
     (?li(k+1) ... lim. t1' /\ ... /\ tn')
```

where each `ti'` is the result of rewriting `ti` with the theorems in `thl`. The set of assumptions `B` is the union of the instantiated assumptions of the theorems used for rewriting. If none of the rewrites are applicable to a conjunct, it is unchanged. After rewriting, the function decides which of the resulting terms to use for unwinding, by performing a loop analysis on the graph representing the dependencies of the lines.

Suppose the function decides to unwind `li1,...,lik` using the terms `ui1',...,uik'` respectively. Then, after unwinding, the lines `li1,...,lik` are pruned (provided they have been eliminated from the right-hand sides of the conjuncts that are equations, and from the whole of any other conjuncts) resulting in the elimination of `ui1',...,uik'`.

The `li`'s are related by the equation:

```
{{li1,...,lik}} u {{li(k+1),...,lim}} = {{l1,...,lm}}
```

The loop analysis allows the term to be unwound as much as possible without the risk of looping. The user is left to deal with the recursive equations.

Failure

The function may fail if the argument term is not of the specified form. It also fails if there is more than one equation for any line variable.

Example

```
#EXPAND_AUTO_CONV
# [ASSUME "!in out. INV (in,out) = !(t:num). out t = ~(in t)"]
# "?l1 l2.
#   INV (l1,l2) /\ INV (l2,out) /\ (!(t:num). l1 t = l2 (t-1) \/ out (t-1))";;
. |- (?l1 l2.
       INV(l1,l2) /\ INV(l2,out) /\ (!t. l1 t = l2(t - 1) \/ out(t - 1))) =
     (?l2.
       (!t. l2 t = ~(l2(t - 1) \/ ~l2(t - 1))) /\ (!t. out t = ~l2 t))
```

See also

unwindLib.EXPAND_ALL_BUT_CONV, unwindLib.EXPAND_AUTO_RIGHT_RULE, unwindLib.EXPAND_-ALL_BUT_RIGHT_RULE, unwindLib.UNFOLD_CONV, unwindLib.UNWIND_AUTO_CONV, unwindLib.PRUNE_SOME_CONV

EXPAND_AUTO_RIGHT_RULE (unwindLib)

```
unwindLib.EXPAND_AUTO_RIGHT_RULE : (thm list -> thm -> thm)
```

Unfolds, then unwinds as much as possible, then prunes the unwound lines.

`EXPAND_AUTO_RIGHT_RULE thl` behaves as follows:

```
    A |- !z1 ... zr.
          t = ?l1 ... lm. t1 /\ ... /\ ui1 /\ ... /\ uik /\ ... /\ tn
   ------------------------------------------------------------------
     B u A |- !z1 ... zr. t = ?li(k+1) ... lim. t1' /\ ... /\ tn'
```

where each `ti'` is the result of rewriting `ti` with the theorems in `thl`. The set of assumptions `B` is the union of the instantiated assumptions of the theorems used for rewriting. If none of the rewrites are applicable to a conjunct, it is unchanged. After rewriting, the function decides which of the resulting terms to use for unwinding, by performing a loop analysis on the graph representing the dependencies of the lines.

Suppose the function decides to unwind `li1,...,lik` using the terms `ui1',...,uik'` respectively. Then, after unwinding, the lines `li1,...,lik` are pruned (provided they

have been eliminated from the right-hand sides of the conjuncts that are equations, and from the whole of any other conjuncts) resulting in the elimination of `ui1',...,uik'`.

The `li`'s are related by the equation:

```
{{li1,...,lik}} u {{li(k+1),...,lim}} = {{l1,...,lm}}
```

The loop analysis allows the term to be unwound as much as possible without the risk of looping. The user is left to deal with the recursive equations.

Failure

The function may fail if the argument theorem is not of the specified form. It also fails if there is more than one equation for any line variable.

Example

```
#EXPAND_AUTO_RIGHT_RULE
# [ASSUME "!in out. INV (in,out) = !(t:num). out t = ~(in t)"]
# (ASSUME
#   "!(in:num->bool) out.
#     DEV(in,out) =
#      ?l1 l2.
#       INV (l1,l2) /\ INV (l2,out) /\ (!(t:num). l1 t = in t \/ out (t-1))");;
.. |- !in out. DEV(in,out) = (!t. out t = ~~(in t \/ out(t - 1)))
```

See also

`unwindLib.EXPAND_ALL_BUT_RIGHT_RULE`, `unwindLib.EXPAND_AUTO_CONV`, `unwindLib.EXPAND_ALL_BUT_CONV`, `unwindLib.UNFOLD_RIGHT_RULE`, `unwindLib.UNWIND_AUTO_RIGHT_RULE`, `unwindLib.PRUNE_SOME_RIGHT_RULE`

expand_list (proofManagerLib)

```
proofManagerLib.expand_list : list_tactic -> proof
```

Applies a list-tactic to replace the current goal list.

The function `expand_list` is part of the subgoal package. It may be abbreviated by the function `elt`. It applies a tactic to the current goal list (that is, the list of goals produced by the most recent use of `expand` or `expand_list`) to give a new proof state. The previous state is stored on the backup list. If the list-tactic produces subgoals, the new proof state is formed from the old one by removing the current goal list from the goal stack and replacing it by the list of subgoals produced by the list-tactic. The corresponding justification is modified accordingly, appropriate to the new goal list. The new subgoals are printed. If more than one subgoal is produced, they are printed from the bottom of the stack so that the new current goal is printed last.

If a list-tactic solves the current goal list (returns an empty subgoal list), then its justification is used to prove a corresponding theorem. This theorem is incorporated into the justification of the parent goal and printed. That level of goals is removed and the parent goal is proved using its (new) justification. This process is repeated until a level with unproven subgoals is reached. The next goal on the goal stack then becomes the current goal. This goal is printed. If all the subgoals are proved, the resulting proof state consists of the theorem proved by the justifications.

The list-tactic applied is a validating version of the list-tactic given. It ensures that the justification of the list-tactic does provide a proof of the goals from the subgoals generated by the tactic. It will cause failure if this is not so. The tactical `VALID_LT` performs this validation.

For a description of the subgoal package, see `set_goal`.

Failure

`expand_list ltac` fails if the tactic `ltac` fails for the current goal list. It will diverge if the list-tactic diverges for the goal. It will fail if there are no unproven goals. This could be because no goal has been set using `set_goal` or because the last goal set has been completely proved. It will also fail in cases when the list-tactic is invalid.

Example

```
> g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;
val it =
   Proof manager status: 2 proofs.
   2. Completed goalstack:  [.] ⊢ q

   1. Incomplete goalstack:
```

```
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

> expand CONJ_TAC;
OK..
2 subgoals:
val it =

   TL [1; 2; 3] = [2; 3]

   HD [1; 2; 3] = 1

> expand_list (ALLGOALS (REWRITE_TAC[listTheory.HD,listTheory.TL])) ;
OK..
val it =
   Initial goal proved.
   ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]: proof
```

Doing a step in an interactive goal-directed proof, in particular, a step which affects all the subgoals generated by the preceding step.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.expand_listf`, `proofManagerLib.flatn`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

expand_listf (proofManagerLib)

```
proofManagerLib.expand_listf : (list_tactic -> unit)
```

Applies a list-tactic to the current goal, stacking the resulting subgoals.

The function `expand_listf` is a faster version of `expand_list`. It does not use a validated version of the list-tactic. That is, no check is made that the justification of the list-tactic

does prove the goals from the subgoals it generates. If an invalid list-tactic is used, the theorem ultimately proved may not match the goal originally set. Alternatively, failure may occur when the justifications are applied in which case the theorem would not be proved. For a description of the subgoal package, see under `set_goal`.

Failure

Calling `expand_listf ltac` fails if the list-tactic `ltac` fails for the current goal list. It will diverge if the list-tactic diverges for the goals. It will fail if there are no unproven goals. This could be because no goal has been set using `set_goal` or because the last goal set has been completely proved. If an invalid tactic, whose justification actually fails, has been used earlier in the proof, `expand_listf ltac` may succeed in applying `ltac` and apparently prove the current goals. It may then fail as it applies the justifications of the tactics applied earlier.

Saving CPU time when doing goal-directed proofs, since the extra validation is not done. Redoing proofs quickly that are already known to work.

Comments

The CPU time saved may cause misery later. If an invalid tactic or list-tactic is used, this will only be discovered when the proof has apparently been finished and the justifications are applied.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.s backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.expand_list`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

EXPAND_REDUCE_CONV (wordsLib)

`wordsLib.EXPAND_REDUCE_CONV : conv`

The conversion `EXPAND_REDUCE_CONV` expands out applications of `reduce_and`, `reduce_or`, `reduce_xor`, `reduce_nand`, `reduce_nor` and `reduce_xnor`.

Example

```
> wordsLib.EXPAND_REDUCE_CONV “reduce_and (w: word4)”;
val it =
   ⊢ reduce_and w = (((3 >< 3) w && (2 >< 2) w) && (1 >< 1) w) && (0 >< 0) w:
   thm
```

See also

`wordsLib.WORD_EVAL_CONV`

expandf (proofManagerLib)

```
proofManagerLib.expandf : (tactic -> unit)
```

Applies a tactic to the current goal, stacking the resulting subgoals.

The function `expandf` is a faster version of `expand`. It does not use a validated version of the tactic. That is, no check is made that the justification of the tactic does prove the goal from the subgoals it generates. If an invalid tactic is used, the theorem ultimately proved may not match the goal originally set. Alternatively, failure may occur when the justifications are applied in which case the theorem would not be proved. For a description of the subgoal package, see under `set_goal`.

Failure

Calling `expandf tac` fails if the tactic `tac` fails for the top goal. It will diverge if the tactic diverges for the goal. It will fail if there are no unproven goals. This could be because no goal has been set using `set_goal` or because the last goal set has been completely proved. If an invalid tactic, whose justification actually fails, has been used earlier in the proof, `expandf tac` may succeed in applying `tac` and apparently prove the current goal. It may then fail as it applies the justifications of the tactics applied earlier.

Example

```
   - g `HD[1;2;3] = 1`;
```

```
`HD[1;2;3] = 1`

() : void

- expandf (REWRITE_TAC[HD;TL]);;
OK..
goal proved
|- HD[1;2;3] = 1

Previous subproof:
goal proved
() : void
```

The following example shows how the use of an invalid tactic can yield a theorem which does not correspond to the goal set.

```
- set_goal([], Term `1=2`);
`1 = 2`

() : void

- expandf (REWRITE_TAC[ASSUME (Term`1=2`)]);
OK..
goal proved
. |- 1 = 2

Previous subproof:
goal proved
() : void
```

The proof assumed something which was not on the assumption list. This assumption appears in the assumption list of the theorem proved, even though it was not in the goal. An attempt to perform the proof using `expand` fails. The validated version of the tactic detects that the justification produces a theorem which does not correspond to the goal set. It therefore fails.

Saving CPU time when doing goal-directed proofs, since the extra validation is not done. Redoing proofs quickly that are already known to work.

Comments

The CPU time saved may cause misery later. If an invalid tactic is used, this will only be discovered when the proof has apparently been finished and the justifications are applied.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

export_mono (IndDefLib)

```
IndDefLib.export_mono : string -> unit
```

Records a theorem as a monotonicity theorem for inductive definitions.

A call to `export_mono "thm_name"` causes the theorem of that name to be stored as a monotonicity theorem, to be used when an inductive definition is made. See the DESCRIPTION manual for more on the required form of the theorem being exported in this way.

Failure

Fails if the string argument is not the name of a stored theorem. The name can be qualified (preceded) with the name of an ancestral theory segment and a full-stop, or can be "bare", in which case it must be the name of a theorem in the current segment.

See also

`IndDefLib.Hol_reln`

export_rewrites (BasicProvers)

```
BasicProvers.export_rewrites : string list -> unit
```

Exports theorems so that they merge with the "stateful" rewriter's `simpset`.

A call to `export_rewrites strlist` causes the theorems named by the strings in `strlist` to be merged into the `simpset` value maintained behind the function `srw_ss()`, both in the current session and also when the theory generated by the script file is loaded.

Theorems are named by giving the name of the segment, a full-stop (or period) character, and the name of theorem. If the theorem is in the current segment, the segment can be omitted. Thus, if working in the development of the theory of lists, the following are valid names `list.MAP_GENLIST`, `MAP_GENLIST` and `arithmetic.LESS_TRANS`.

The collection of all the theorems specified in calls to `export_rewrites` can be obtained as a value of type `simpLib.ssfrag` using the `thy_ssfrag` function.

Multiple calls to `export_rewrites` cumulatively add to the list of theorems being exported.

Failure

Fails if any of the strings in the list does not name a theorem in the current context.

Comments

This function is useful for ensuring that the stateful rewriter is augmented as theories are loaded. This in turn means that users of these theories don't need to learn the names of their "obvious" theorems. Because theorems can not be removed from the stateful rewriter's underlying `simpset`, choice of "obvious" theorems needs to be done with care.

See also

`bossLib.augment_srw_ss`, `bossLib.srw_ss`, `bossLib.SRW_TAC`, `BasicProvers.thy_ssfrag`

export_theory (Theory)

```
Theory.export_theory : unit -> unit
```

Write a theory segment to disk.

If the `Globals.interactive` flag is false, an invocation `export_theory()` saves the current theory segment to disk. All parents, definitions, axioms, and stored theorems of the segment are saved in such a way that, when the theory is loaded from disk in a later session, the full theory in place at the time `export_theory` was called is re-instated.

If the `Globals.interactive` flag is true, the call `export_theory()` does nothing, returning unit `()` instantly.

If the current theory segment is named `thy`, then `export_theory()` will create ML files `thyTheory.sig` and `thyTheory.sml`, in the current directory at the time `export_theory` is invoked. These files need to be compiled before they become usable. In the standard way of doing things, the `Holmake` facility will handle this task.

Once a theory segment has been exported and compiled, it is available for use. It can be brought into an interactive proof session via

```
load "thyTheory";
```

When the segment is loaded, its parents, axioms, theorems, and definitions are incorporated into the current theory (recall that this notion is different than the current theory segment).

Failure

A call to `export_theory` may fail if the disk file cannot be opened. A call to `export_theory` will also fail if some bindings are such that the name of the binding is not a valid ML identifier. In that case, `export_theory` will report all such bad names. These can be changed with `set_MLname`, and then `export_theory` may be attempted again.

Example

```
save_thm("foo", REFL “x:bool”);
```

```
Globals.interactive := false;

export_theory();
```

(These calls are not evaluated by polyscripter when this entry is rendered: actually running `export_theory()` freezes the polyscripter session's current theory segment and breaks every subsequent `Definition` / `Theorem` example.)

Comments

Note that `export_theory` exports the state of the theory (which can include other user-definable data in addition to the logical content (theorems, definitions, etc.), but not the state of the SML session.

When theories are developed interactively, the `interactive` flag will typically be set to `true`; preventing `export_theory()` from doing anything in this situation reserves special behaviours for when theories are built with `Holmake`.

See also

`Theory.new_theory`, `Theory.adjoin_to_theory`, `Theory.set_MLname`

EXT (Drule)

```
Drule.EXT : thm -> thm
```

Derives equality of functions from extensional equivalence.

When applied to a theorem `A |- !x. t1 x = t2 x`, the inference rule `EXT` returns the theorem `A |- t1 = t2`.

```
 A |- !x. t1 x = t2 x
---------------------  EXT          [where x is not free in t1 or t2]
     A |- t1 = t2
```

Failure

Fails if the theorem does not have the form indicated above, or if the variable `x` is free in either of the functions `t1` or `t2`.

Comments

This rule is expressed as an equivalence in the theorem `boolTheory.FUN_EQ_THM`.

See also

`Thm.AP_THM`, `Drule.ETA_CONV`, `Conv.FUN_EQ_CONV`

EXT_CONSEQ_REWRITE_CONV (ConseqConv)

```
ConseqConv.EXT_CONSEQ_REWRITE_CONV : (thm list -> conv) list -> thm list ->
                          (thm list * thm list * thm list) ->
                          directed_conseq_conv
```

Applies `CONSEQ_REWRITE_CONV` interleaved with conversions and rewrites.

`CONSEQ_REWRITE_CONV` often results in theorems of the following form

```
|- (!x. T) /\ (T /\ (T /\ T)) /\ (\x. P) y /\ T ==>
   something
```

The problem is that `CONSEQ_REWRITE_CONV` applies consequence conversions, but no normal convs or simplifications. This is changed by `EXT_CONSEQ_REWRITE_CONV`. `EXT_CONSEQ_REWRITE_CONV` gets a list of conversions and a list of rewrite theorems. Moreover there are the parameters of `CONSEQ_REWRITE_CONV`. It then applies these conversions (e.g. `DEPTH_CONV BETA_CONV`) and a `REWRITE_CONV` with the given theorem list interleaved with `CONSEQ_REWRITE_CONV`. As a result the theorem above might look now like

```
|- P y ==> something
```

See also

`ConseqConv.CONSEQ_REWRITE_CONV`

EXT_DEPTH_CONSEQ_CONV (ConseqConv)

```
ConseqConv.EXT_DEPTH_CONSEQ_CONV : conseq_conv_congruence list ->
                        depth_conseq_conv_cache_opt -> int option ->
                        bool ->
                        (bool * int option * (thm list -> directed_conseq_conv)) li
                        thm list ->
                        directed_conseq_conv
```

The general depth consequence conversion of which DEPTH_CONSEQ_CONV, REDEPTH_CONSEQ_CONV, ONCE_DEPTH_CONSEQ_CONV etc are just instantiations.

DEPTH_CONSEQ_CONV and similar conversions are able to apply a consequence conversion by breaking down the structure of a term using lemmata about /\, \/, ~, ==>, if-then-else and quantification. While doing so, these conversions collect various amounts of context information. EXT_DEPTH_CONSEQ_CONV congruence_list cache_opt step_opt redepth convL context is the function used by these other depth conversions. For this purpose, the

cache_opt determines which cache to use: NONE means no caching; a standard cache that stores everything is configured by CONSEQ_CONV_default_cache_opt.

The number of steps taken is determined by step_opt. NONE means arbitrarily many; SOME n means at most n. ONCE_DEPTH_CONSEQ_CONV for example uses SOME 1. The parameter redepth determines whether modified terms should be revisited and convL is a basically a list of directed consequence conversions of the conversions that should be applied at subpositions. Its entries consist of a flag, whether to apply the conversion before or after descending into subterms; the weight (i.e. the number of counted steps) for the conversion, and a function from the context (a list of theorems) to the conversion. context provides additional context that might be used.

The first parameter congruence_list is a list of congruence functions that determine how to break down terms. Each element of this list has to be a function congruence context sys dir t which returns a pair of the number of performed steps and a resulting theorem. sys is a callback that allows to apply the depth conversion recursively to subterms. context represents the context that can be used. If you ignore the slightly different return type, the congruence is otherwise a directed consequence conversion. If the congruence can't be applied, it should either fail or raise an UNCHANGED exception. The callback sys gets the number of already performed steps, a direction and a term. It then returns a accumulated number of steps and a thm option. It never fails. The

number of steps is used to abort if the maximum number of globally allowed steps has been reached. The first call of `sys` should get `0`, then the accumulated number has to be passed. The congruence should return the finally accumulated number of steps.

See also

`ConseqConv.DEPTH_CONSEQ_CONV`, `ConseqConv.REDEPTH_CONSEQ_CONV`, `ConseqConv.ONCE_DEPTH_CONSEQ_CONV`, `ConseqConv.NUM_DEPTH_CONSEQ_CONV`

F (boolSyntax)

```
boolSyntax.F : term
```

Constant denoting falsity.

The ML variable `boolSyntax.F` is bound to the term `bool$F`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.negation`, `boolSyntax.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`

fail (Feedback)

```
Feedback.fail : unit -> 'a
```

Raise a `HOL_ERR`.

The function `fail` raises a `HOL_ERR` with default values. This is useful when detailed error tracking is not necessary.

Failure

Always fails.

Example

```
> fail() handle e => Raise e;
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
HOL_ERR (at ??.??: fail) raised
```

See also

`Feedback`, `Feedback.failwith`, `Feedback.Raise`, `Feedback.HOL_ERR`

FAIL_LT (Tactical)

```
Tactical.FAIL_LT : string -> list_tactic
```

List-tactic which always fails, with the supplied string.

Whatever goal list it is applied to, `FAIL_LT s` always fails with the string `s`.

Failure

The application of `FAIL_LT` to a string never fails; the resulting list-tactic always fails.

See also

`Tactical.FAIL_TAC`, `Tactical.ALL_LT`, `Tactical.NO_LT`

FAIL_TAC (Tactical)

```
Tactical.FAIL_TAC : string -> tactic
```

Tactic which always fails, with the supplied string.

Whatever goal it is applied to, `FAIL_TAC s` always fails with the string `s`.

Failure

The application of `FAIL_TAC` to a string never fails; the resulting tactic always fails.

Example

The following example uses the fact that if a tactic `t1` solves a goal, then the tactic `t1 THEN t2` never results in the application of `t2` to anything, because `t1` produces no subgoals. In attempting to solve the following goal:

```
?- if x then T else T
```

the tactic

```
REWRITE_TAC[] THEN FAIL_TAC "Simple rewriting failed to solve goal"
```

will fail with the message provided, whereas:

```
CONV_TAC COND_CONV THEN FAIL_TAC "Using COND_CONV failed to solve goal"
```

will silently solve the goal because `COND_CONV` reduces it to just `?- T`.

See also

`Tactical.ALL_TAC`, `Tactical.NO_TAC`

failwith (Feedback)

```
Feedback.failwith : string -> 'a
```

Raise a HOL_ERR.

The function failwith raises a HOL_ERR with default values. This is useful when detailed error tracking is not necessary.

failwith differs from fail in that it takes an extra string argument, which is typically used to tell which function failwith is being called from.

Failure

Always fails.

Example

```
> failwith "foo" handle e => Raise e;
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
HOL_ERR (at ??.failwith: foo) raised
```

See also

Feedback, Feedback.fail, Feedback.Raise, Feedback.HOL_ERR

FALSE_CONSEQ_CONV (ConseqConv)

ConseqConv.FALSE_CONSEQ_CONV : conseq_conv

Given a term t of type bool this consequence conversion returns the theorem |- F ==> t.

See also

ConseqConv.TRUE_CONSEQ_CONV, ConseqConv.REFL_CONSEQ_CONV, ConseqConv.TRUE_-FALSE_REFL_CONSEQ_CONV

FAST_ASM_QUANT_INSTANTIATE_TAC(quantHeuristicsLib)

```
quantHeuristicsLib.FAST_ASM_QUANT_INSTANTIATE_TAC : quant_param list -> tactic
```

A fast version of `quantHeuristicsLib.ASM_QUANT_INSTANTIATE_TAC`. It does not preprocess the term in order to minimise the number of variable occurrences.

See also

`quantHeuristicsLib.QUANT_INSTANTIATE_CONV`, `quantHeuristicsLib.ASM_QUANT_INSTANTIATE_TAC`

FAST_QUANT_INST_ss (quantHeuristicsLib)

```
quantHeuristicsLib.FAST_QUANT_INST_ss : quant_param list -> simpLib.ssfrag
```

A simpset fragement corresponding to `FAST_QUANT_INSTANTIATE_CONV`.

See also

`quantHeuristicsLib.FAST_QUANT_INSTANTIATE_CONV`

FAST_QUANT_INSTANTIATE_CONV(quantHeuristicsLib)

```
quantHeuristicsLib.FAST_QUANT_INSTANTIATE_CONV : quant_param list -> conv
```

A fast version of `quantHeuristicsLib.QUANT_INSTANTIATE_CONV`. It does not preprocess the term in order to minimise the number of variable occurrences.

See also

quantHeuristicsLib.QUANT_INSTANTIATE_CONV, quantHeuristicsLib.FAST_QUANT_-INSTANTIATE_TAC

FAST_QUANT_INSTANTIATE_TAC(quantHeuristicsLib)

```
quantHeuristicsLib.FAST_QUANT_INSTANTIATE_TAC : quant_param list -> tactic
```

A fast version of quantHeuristicsLib.QUANT_INSTANTIATE_TAC. It does not preprocess the term in order to minimise the number of variable occurrences.

See also

quantHeuristicsLib.QUANT_INSTANTIATE_CONV, quantHeuristicsLib.QUANT_INSTANTIATE_-TAC

FCP_ss (fcpLib)

```
fcpLib.FCP_ss : ssfrag
```

A simpset fragment for simplifying finite Cartesian product expressions.

Example

```
simpLib.SSFRAG{ac = [], congs = [], convs = [], dprocs = [], filter = NONE,
               rewrs =
                 [|- !i. i < dimindex (:'b) ==> ($FCP g ' i = g i),
                  |- !g. (FCP i. g ' i) = g,
                  |- !x y. (x = y) = !i. i < dimindex (:'b) ==> (x ' i = y ' i)]}
  : ssfrag
```

See also

`wordsLib.WORD_BIT_EQ_ss`

Feedback (structure)

```
structure Feedback
```

Module for messages, warnings, errors, and tracing of HOL functions.

The `Feedback` structure provides facilities for raising and viewing HOL errors, and also for monitoring tools as they run.

fetch (DB)

```
DB.fetch : string -> string -> thm
```

Fetch a theorem by theory and name.

An invocation `fetch thy name` searches through the currently loaded theory segments in an attempt to find a theorem, axiom, or definition stored under `name` in theory `thy`.

Failure

If the specified theorem, axiom, or definition cannot be located.

Example

```
> DB.fetch "bool" "NOT_FORALL_THM";
val it = ⊢ ∀P. ¬(∀x. P x) ⇔ ∃x. ¬P x: thm
```

See also

`DB.thms`, `DB.thy`, `DB.theorems`, `DB.axioms`, `DB.definitions`

filter (Lib)

```
Lib.filter : ('a -> bool) -> 'a list -> 'a list
```

Filters a list to the sublist of elements satisfying a predicate.

`filter P l` applies `P` to every element of `l`, returning a list of those that satisfy `P`, in the order they appeared in the original list.

Failure

If `P x` fails for some element `x` of `l`.

Comments

Identical to `List.filter` from the Standard ML Basis Library.

See also

`Lib.mapfilter`, `Lib.partition`

FILTER_ASM_REWRITE_RULE (Rewrite)

```
Rewrite.FILTER_ASM_REWRITE_RULE : ((term -> bool) -> thm list -> thm -> thm)
```

Rewrites a theorem including built-in rewrites and some of the theorem's assumptions.

This function implements selective rewriting with a subset of the assumptions of the theorem. The first argument (a predicate on terms) is applied to all assumptions, and the ones which return `true` are used (along with the set of basic tautologies and the given theorem list) to rewrite the theorem. See `GEN_REWRITE_RULE` for more information on rewriting.

Failure

FILTER_ASM_REWRITE_RULE does not fail. Using FILTER_ASM_REWRITE_RULE may result in a diverging sequence of rewrites. In such cases FILTER_ONCE_ASM_REWRITE_RULE may be used.

This rule can be applied when rewriting with all assumptions results in divergence. Typically, the predicate can model checks as to whether a certain variable appears on the left-hand side of an equational assumption, or whether the assumption is in disjunctive form.

Another use is to improve performance when there are many assumptions which are not applicable. Rewriting, though a powerful method of proving theorems in HOL, can result in a reduced performance due to the pattern matching and the number of primitive inferences involved.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.FILTER_ONCE_ASM_REWRITE_RULE, Rewrite.FILTER_PURE_ASM_REWRITE_RULE, Rewrite.FILTER_PURE_ONCE_ASM_REWRITE_RULE, Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.REWRITE_RULE

FILTER_ASM_REWRITE_TAC (Rewrite)

```
Rewrite.FILTER_ASM_REWRITE_TAC : ((term -> bool) -> thm list -> tactic)
```

Rewrites a goal including built-in rewrites and some of the goal's assumptions.

This function implements selective rewriting with a subset of the assumptions of the goal. The first argument (a predicate on terms) is applied to all assumptions, and the ones which return true are used (along with the set of basic tautologies and the given theorem list) to rewrite the goal. See GEN_REWRITE_TAC for more information on rewriting.

Failure

FILTER_ASM_REWRITE_TAC does not fail, but it can result in an invalid tactic if the rewrite is invalid. This happens when a theorem used for rewriting has assumptions which are

not alpha-convertible to assumptions of the goal. Using FILTER_ASM_REWRITE_TAC may result in a diverging sequence of rewrites. In such cases FILTER_ONCE_ASM_REWRITE_TAC may be used.

This tactic can be applied when rewriting with all assumptions results in divergence, or in an unwanted proof state. Typically, the predicate can model checks as to whether a certain variable appears on the left-hand side of an equational assumption, or whether the assumption is in disjunctive form. Thus it allows choice of assumptions to rewrite with in a position-independent fashion.

Another use is to improve performance when there are many assumptions which are not applicable. Rewriting, though a powerful method of proving theorems in HOL, can result in a reduced performance due to the pattern matching and the number of primitive inferences involved.

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.FILTER_ONCE_ASM_REWRITE_TAC, Rewrite.FILTER_PURE_ASM_REWRITE_TAC, Rewrite.FILTER_PURE_ONCE_ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.ONCE_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC

FILTER_CONV (listLib)

```
listLib.FILTER_CONV : conv -> conv
```

Computes by inference the result of applying a predicate to the elements of a list.

FILTER_CONV takes a conversion conv and a term tm in the following form:

```
FILTER P [x0;...xn]
```

It returns the theorem

```
|- FILTER P [x0;...xn] = [...xi...]
```

where for every xi occurring in the right-hand side of the resulting theorem, conv "P xi" returns a theorem |- P xi = T.

Failure

`FILTER_CONV conv tm` fails if `tm` is not of the form described above.

Example

Evaluating

```
FILTER_CONV bool_EQ_CONV “FILTER ($= T) [T;F;T]”;
```

returns the following theorem:

```
|- FILTER($= T)[T;F;T] = [T;T]
```

In general, if the predicate `P` is an explicit lambda abstraction `(\x. P x)`, the conversion should be in the form

```
(BETA_CONV THENC conv')
```

See also

`listLib.FOLDL_CONV`, `listLib.FOLDR_CONV`, `listLib.list_FOLD_CONV`

FILTER_DISCH_TAC (Tactic)

```
Tactic.FILTER_DISCH_TAC : (term -> tactic)
```

Conditionally moves the antecedent of an implicative goal into the assumptions.

`FILTER_DISCH_TAC` will move the antecedent of an implication into the assumptions, provided its parameter does not occur in the antecedent.

```
 A ?- u ==> v
==============  FILTER_DISCH_TAC w
 A u {u} ?- v
```

Note that DISCH_TAC treats ~u as u ==> F. Unlike DISCH_TAC, the antecedent will be STRIPed into its various components before being ASSUMEd. This stripping includes generating multiple goals for case-analysis of disjunctions. Also, unlike DISCH_TAC, should any component of the discharged antecedent directly imply or contradict the goal, then this simplification will also be made. Again, unlike DISCH_TAC, FILTER_DISCH_TAC will not duplicate identical or alpha-equivalent assumptions.

Failure

FILTER_DISCH_TAC will fail if a term which is identical, or alpha-equivalent to w occurs free in the antecedent, or if the theorem is not an implication or a negation.

Comments

FILTER_DISCH_TAC w behaves like FILTER_DISCH_THEN STRIP_ASSUME_TAC w.

See also

Thm.DISCH, Drule.DISCH_ALL, Tactic.DISCH_TAC, Thm_cont.DISCH_THEN, Tactic.FILTER_DISCH_THEN, Drule.NEG_DISCH, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC

FILTER_DISCH_THEN (Tactic)

```
Tactic.FILTER_DISCH_THEN : (thm_tactic -> term -> tactic)
```

Conditionally gives to a theorem-tactic the antecedent of an implicative goal.

If FILTER_DISCH_THEN's second argument, a term, does not occur in the antecedent, then FILTER_DISCH_THEN removes the antecedent and then creates a theorem by ASSUMEing it. This new theorem is passed to FILTER_DISCH_THEN's first argument, which is subsequently expanded. For example, if

```
    A ?- t
========  f (ASSUME u)
    B ?- v
```

then

```
 A ?- u ==> t
==============  FILTER_DISCH_THEN f
    B ?- v
```

Note that FILTER_DISCH_THEN treats ~u as u ==> F.

Failure

FILTER_DISCH_THEN will fail if a term which is identical, or alpha-equivalent to w occurs free in the antecedent. FILTER_DISCH_THEN will also fail if the theorem is an implication or a negation.

Comments

FILTER_DISCH_THEN is most easily understood by first understanding DISCH_THEN.

For preprocessing an antecedent before moving it to the assumptions, or for using antecedents and then throwing them away.

See also

Thm.DISCH, Drule.DISCH_ALL, Tactic.DISCH_TAC, Thm_cont.DISCH_THEN, Tactic.FILTER_DISCH_TAC, Drule.NEG_DISCH, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC

FILTER_GEN_TAC (Tactic)

```
Tactic.FILTER_GEN_TAC : (term -> tactic)
```

Strips off a universal quantifier, but fails for a given quantified variable.

When applied to a term s and a goal A ?- !x. t, the tactic FILTER_GEN_TAC fails if the quantified variable x is the same as s, but otherwise advances the goal in the same way as GEN_TAC, i.e. returns the goal A ?- t[x'/x] where x' is a variant of x chosen to avoid clashing with any variables free in the goal's assumption list. Normally x' is just x.

```
  A ?- !x. t
==============  FILTER_GEN_TAC "s"
 A ?- t[x'/x]
```

Failure

Fails if the goal's conclusion is not universally quantified or the quantified variable is equal to the given term.

See also

Thm.GEN, Tactic.GEN_TAC, Thm.GENL, Drule.GEN_ALL, Thm.SPEC, Drule.SPECL, Drule.SPEC_ALL, Tactic.SPEC_TAC, Tactic.STRIP_TAC

FILTER_ONCE_ASM_REWRITE_RULE (Rewrite)

```
Rewrite.FILTER_ONCE_ASM_REWRITE_RULE : ((term -> bool) -> thm list -> thm -> thm)
```

Rewrites a theorem once including built-in rewrites and some of its assumptions.

The first argument is a predicate applied to the assumptions. The theorem is rewritten with the assumptions for which the predicate returns `true`, the given list of theorems, and the tautologies stored in `implicit_rewrites`. It searches the term of the theorem once, without applying rewrites recursively. Thus it avoids the divergence which can result from the application of FILTER_ASM_REWRITE_RULE. For more information on rewriting rules, see GEN_REWRITE_RULE.

Failure

Never fails.

This function is useful when rewriting with a subset of assumptions of a theorem, allowing control of the number of rewriting passes.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.FILTER_ASM_REWRITE_RULE, Rewrite.FILTER_PURE_ASM_REWRITE_RULE, Rewrite.FILTER_PURE_ONCE_ASM_REWRITE_RULE, Rewrite.GEN_

REWRITE_RULE, Rewrite.ONCE_ASM_REWRITE_RULE, Conv.ONCE_DEPTH_CONV, Rewrite.PURE_ASM_REWRITE_RULE, Rewrite.PURE_ONCE_ASM_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.REWRITE_RULE

FILTER_ONCE_ASM_REWRITE_TAC (Rewrite)

```
Rewrite.FILTER_ONCE_ASM_REWRITE_TAC : ((term -> bool) -> thm list -> tactic)
```

Rewrites a goal once including built-in rewrites and some of its assumptions.

The first argument is a predicate applied to the assumptions. The goal is rewritten with the assumptions for which the predicate returns `true`, the given list of theorems, and the tautologies stored in `implicit_rewrites`. It searches the term of the goal once, without applying rewrites recursively. Thus it avoids the divergence which can result from the application of FILTER_ASM_REWRITE_TAC. For more information on rewriting tactics, see GEN_REWRITE_TAC.

Failure

Never fails.

This function is useful when rewriting with a subset of assumptions of a goal, allowing control of the number of rewriting passes.

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.FILTER_ASM_REWRITE_TAC, Rewrite.FILTER_PURE_ASM_REWRITE_TAC, Rewrite.FILTER_PURE_ONCE_ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.ONCE_ASM_REWRITE_TAC, Conv.ONCE_DEPTH_CONV, Rewrite.PURE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_ASM_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC

FILTER_PGEN_TAC (PairRules)

```
PairRules.FILTER_PGEN_TAC : (term -> tactic)
```

Strips off a paired universal quantifier, but fails for a given quantified pair.

When applied to a term `q` and a goal `A ?- !p. t`, the tactic `FILTER_PGEN_TAC` fails if the quantified pair `p` is the same as `p`, but otherwise advances the goal in the same way as `PGEN_TAC`, i.e. returns the goal `A ?- t[p'/p]` where `p'` is a variant of `p` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `p'` is just `p`.

```
  A ?- !p. t
============  FILTER_PGEN_TAC "q"
 A ?- t[p'/p]
```

Failure

Fails if the goal's conclusion is not a paired universal quantifier or the quantified pair is equal to the given term.

See also

`Tactic.FILTER_GEN_TAC`, `PairRules.PGEN`, `PairRules.PGEN_TAC`, `PairRules.PGENL`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`, `PairRules.PSTRIP_TAC`

FILTER_PSTRIP_TAC (PairRules)

```
PairRules.FILTER_PSTRIP_TAC : (term -> tactic)
```

Conditionally strips apart a goal by eliminating the outermost connective.

Stripping apart a goal in a more careful way than is done by `PSTRIP_TAC` may be necessary when dealing with quantified terms and implications. `FILTER_PSTRIP_TAC` behaves like `PSTRIP_TAC`, but it does not strip apart a goal if it contains a given term.

If `u` is a term, then `FILTER_PSTRIP_TAC u` is a tactic that removes one outermost occurrence of one of the connectives `!`, `==>`, `~` or `/\` from the conclusion of the goal `t`, provided the term being stripped does not contain `u`. `FILTER_PSTRIP_TAC` will strip

paired universal quantifications. A negation `~t` is treated as the implication `t ==> F`. `FILTER_PSTRIP_TAC` also breaks apart conjunctions without applying any filtering.

If `t` is a universally quantified term, `FILTER_PSTRIP_TAC u` strips off the quantifier:

```
    A ?- !p. v
================  FILTER_PSTRIP_TAC "u"       [where p is not u]
  A ?- v[p'/p]
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the assumptions `A`. If `t` is a conjunction, no filtering is done and `FILTER_PSTRIP_TAC` simply splits the conjunction:

```
    A ?- v /\ w
=================  FILTER_PSTRIP_TAC "u"
 A ?- v   A ?- w
```

If `t` is an implication and the antecedent does not contain a free instance of `u`, then `FILTER_PSTRIP_TAC u` moves the antecedent into the assumptions and recursively splits the antecedent according to the following rules (see `PSTRIP_ASSUME_TAC`):

```
 A ?- v1 /\ ... /\ vn ==> v          A ?- v1 \/ ... \/ vn ==> v
============================       ================================
    A u {v1,...,vn} ?- v            A u {v1} ?- v ... A u {vn} ?- v

  A ?- (?p. w) ==> v
====================
 A u {w[p'/p]} ?- v
```

where `p'` is a variant of the pair `p`.

Failure

`FILTER_PSTRIP_TAC u (A,t)` fails if `t` is not a universally quantified term, an implication, a negation or a conjunction; or if the term being stripped contains `u` in the sense described above (conjunction excluded).

`FILTER_PSTRIP_TAC` is used when stripping outer connectives from a goal in a more delicate way than `PSTRIP_TAC`. A typical application is to keep stripping by using the tactic `REPEAT (FILTER_PSTRIP_TAC u)` until one hits the term `u` at which stripping is to stop.

See also

PairRules.PGEN_TAC, PairRules.PSTRIP_GOAL_THEN, PairRules.FILTER_PSTRIP_-THEN, PairRules.PSTRIP_TAC, Tactic.FILTER_STRIP_TAC

FILTER_PSTRIP_THEN (PairRules)

```
PairRules.FILTER_PSTRIP_THEN : (thm_tactic -> term -> tactic)
```

Conditionally strips a goal, handing an antecedent to the theorem-tactic.

Given a theorem-tactic ttac, a term u and a goal (A,t), FILTER_STRIP_THEN ttac u removes one outer connective (!, ==>, or ~) from t, if the term being stripped does not contain a free instance of u. Note that FILTER_PSTRIP_THEN will strip paired universal quantifiers. A negation ~t is treated as the implication t ==> F. The theorem-tactic ttac is applied only when stripping an implication, by using the antecedent stripped off. FILTER_PSTRIP_THEN also breaks conjunctions.

FILTER_PSTRIP_THEN behaves like PSTRIP_GOAL_THEN, if the term being stripped does not contain a free instance of u. In particular, FILTER_PSTRIP_THEN PSTRIP_ASSUME_TAC behaves like FILTER_PSTRIP_TAC.

Failure

FILTER_PSTRIP_THEN ttac u (A,t) fails if t is not a paired universally quantified term, an implication, a negation or a conjunction; or if the term being stripped contains the term u (conjunction excluded); or if the application of ttac fails, after stripping the goal.

FILTER_PSTRIP_THEN is used to manipulate intermediate results using theorem-tactics, after stripping outer connectives from a goal in a more delicate way than PSTRIP_GOAL_-THEN.

See also

PairRules.PGEN_TAC, PairRules.PSTRIP_GOAL_THEN, Tactic.FILTER_STRIP_THEN, PairRules.PSTRIP_TAC, PairRules.FILTER_PSTRIP_TAC

FILTER_PURE_ASM_REWRITE_RULE (Rewrite)

```
Rewrite.FILTER_PURE_ASM_REWRITE_RULE : ((term -> bool) -> thm list -> thm ->thm)
```

Rewrites a theorem with some of the theorem's assumptions.

This function implements selective rewriting with a subset of the assumptions of the theorem. The first argument (a predicate on terms) is applied to all assumptions, and the ones which return `true` are used to rewrite the goal. See `GEN_REWRITE_RULE` for more information on rewriting.

Failure

`FILTER_PURE_ASM_REWRITE_RULE` does not fail. Using `FILTER_PURE_ASM_REWRITE_RULE` may result in a diverging sequence of rewrites. In such cases `FILTER_PURE_ONCE_ASM_REWRITE_RULE` may be used.

This rule can be applied when rewriting with all assumptions results in divergence. Typically, the predicate can model checks as to whether a certain variable appears on the left-hand side of an equational assumption, or whether the assumption is in disjunctive form.

Another use is to improve performance when there are many assumptions which are not applicable. Rewriting, though a powerful method of proving theorems in HOL, can result in a reduced performance due to the pattern matching and the number of primitive inferences involved.

See also

`Rewrite.ASM_REWRITE_RULE`, `Rewrite.FILTER_ASM_REWRITE_RULE`, `Rewrite.FILTER_ONCE_ASM_REWRITE_RULE`, `Rewrite.FILTER_PURE_ONCE_ASM_REWRITE_RULE`, `Rewrite.GEN_REWRITE_RULE`, `Rewrite.ONCE_REWRITE_RULE`, `Rewrite.PURE_REWRITE_RULE`, `Rewrite.REWRITE_RULE`

FILTER_PURE_ASM_REWRITE_TAC (Rewrite)

```
Rewrite.FILTER_PURE_ASM_REWRITE_TAC : ((term -> bool) -> thm list -> tactic)
```

Rewrites a goal with some of the goal's assumptions.

This function implements selective rewriting with a subset of the assumptions of the goal. The first argument (a predicate on terms) is applied to all assumptions, and the ones which return `true` are used to rewrite the goal. See `GEN_REWRITE_TAC` for more information on rewriting.

Failure

`FILTER_PURE_ASM_REWRITE_TAC` does not fail, but it can result in an invalid tactic if the rewrite is invalid. This happens when a theorem used for rewriting has assumptions which are not alpha-convertible to assumptions of the goal. Using `FILTER_PURE_ASM_REWRITE_TAC` may result in a diverging sequence of rewrites. In such cases `FILTER_PURE_ONCE_ASM_REWRITE_TAC` may be used.

This tactic can be applied when rewriting with all assumptions results in divergence, or in an unwanted proof state. Typically, the predicate can model checks as to whether a certain variable appears on the left-hand side of an equational assumption, or whether the assumption is in disjunctive form. Thus it allows choice of assumptions to rewrite with in a position-independent fashion.

Another use is to improve performance when there are many assumptions which are not applicable. Rewriting, though a powerful method of proving theorems in HOL, can result in a reduced performance due to the pattern matching and the number of primitive inferences involved.

See also

`Rewrite.ASM_REWRITE_TAC`, `Rewrite.FILTER_ASM_REWRITE_TAC`, `Rewrite.FILTER_ONCE_ASM_REWRITE_TAC`, `Rewrite.FILTER_PURE_ONCE_ASM_REWRITE_TAC`, `Rewrite.GEN_REWRITE_TAC`, `Rewrite.ONCE_REWRITE_TAC`, `Rewrite.PURE_REWRITE_TAC`, `Rewrite.REWRITE_TAC`

FILTER_PURE_ONCE_ASM_REWRITE_RULE(Rewrite)

```
Rewrite.FILTER_PURE_ONCE_ASM_REWRITE_RULE : ((term -> bool) -> thm list -> thm -> t
```

Rewrites a theorem once using some of its assumptions.

The first argument is a predicate applied to the assumptions. The theorem is rewritten with the assumptions for which the predicate returns `true` and the given list of theorems. It searches the term of the theorem once, without applying rewrites recursively. Thus it avoids the divergence which can result from the application of `FILTER_PURE_ASM_REWRITE_RULE`. For more information on rewriting rules, see `GEN_REWRITE_RULE`.

Failure

Never fails.

This function is useful when rewriting with a subset of assumptions of a theorem, allowing control of the number of rewriting passes.

See also

`Rewrite.ASM_REWRITE_RULE`, `Rewrite.FILTER_ASM_REWRITE_RULE`, `Rewrite.FILTER_ONCE_ASM_REWRITE_RULE`, `Rewrite.FILTER_PURE_ASM_REWRITE_RULE`, `Rewrite.GEN_REWRITE_RULE`, `Rewrite.ONCE_ASM_REWRITE_RULE`, `Conv.ONCE_DEPTH_CONV`, `Rewrite.PURE_ASM_REWRITE_RULE`, `Rewrite.PURE_ONCE_ASM_REWRITE_RULE`, `Rewrite.PURE_REWRITE_RULE`, `Rewrite.REWRITE_RULE`

FILTER_PURE_ONCE_ASM_REWRITE_TAC (Rewrite)

```
Rewrite.FILTER_PURE_ONCE_ASM_REWRITE_TAC : ((term -> bool) -> thm list -> tactic)
```

Rewrites a goal once using some of its assumptions.

The first argument is a predicate applied to the assumptions. The goal is rewritten with the assumptions for which the predicate returns `true` and the given list of theorems. It searches the term of the goal once, without applying rewrites recursively. Thus it avoids the divergence which can result from the application of `FILTER_PURE_ASM_REWRITE_TAC`. For more information on rewriting tactics, see `GEN_REWRITE_TAC`.

Failure

Never fails.

This function is useful when rewriting with a subset of assumptions of a goal, allowing control of the number of rewriting passes.

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.FILTER_ASM_REWRITE_TAC, Rewrite.FILTER_ONCE_ASM_REWRITE_TAC, Rewrite.FILTER_PURE_ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.ONCE_ASM_REWRITE_TAC, Conv.ONCE_DEPTH_CONV, Rewrite.PURE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_ASM_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC

FILTER_STRIP_TAC (Tactic)

```
Tactic.FILTER_STRIP_TAC : term -> tactic
```

Conditionally strips apart a goal by eliminating the outermost connective.

Stripping apart a goal in a more careful way than is done by STRIP_TAC may be necessary when dealing with quantified terms and implications. FILTER_STRIP_TAC behaves like STRIP_TAC, but it does not strip apart a goal if it contains a given term.

If u is a term, then FILTER_STRIP_TAC u is a tactic that removes one outermost occurrence of one of the connectives !, ==>, ~ or /\ from the conclusion of the goal t, provided the term being stripped does not contain u. A negation ~t is treated as the implication t ==> F. FILTER_STRIP_TAC u also breaks apart conjunctions without applying any filtering.

If t is a universally quantified term, FILTER_STRIP_TAC u strips off the quantifier:

```
    A ?- !x.v
================  FILTER_STRIP_TAC ``u``       [where x is not u]
  A ?- v[x'/x]
```

where x' is a primed variant that does not appear free in the assumptions A. If t is a conjunction, no filtering is done and FILTER_STRIP_TAC u simply splits the conjunction:

```
    A ?- v /\ w
================  FILTER_STRIP_TAC ``u``
 A ?- v   A ?- w
```

If `t` is an implication and the antecedent does not contain a free instance of `u`, then `FILTER_STRIP_TAC u` moves the antecedent into the assumptions and recursively splits the antecedent according to the following rules (see `STRIP_ASSUME_TAC`):

```
 A ?- v1 /\ ... /\ vn ==> v
============================
    A u {v1,...,vn} ?- v
```

```
   A ?- v1 \/ ... \/ vn ==> v
================================
 A u {v1} ?- v ... A u {vn} ?- v
```

```
  A ?- ?x.w ==> v
====================
 A u {w[x'/x]} ?- v
```

where `x'` is a variant of `x`.

Failure

`FILTER_STRIP_TAC u (A,t)` fails if `t` is not a universally quantified term, an implication, a negation or a conjunction; or if the term being stripped contains `u` in the sense described above (conjunction excluded).

Example

When trying to solve the goal

```
?- !n. m <= n /\ n <= m ==> (m = n)
```

the universally quantified variable `n` can be stripped off by using

```
FILTER_STRIP_TAC ``m:num``
```

and then the implication can be stripped apart by using

```
FILTER_STRIP_TAC ``m:num = n``
```

`FILTER_STRIP_TAC` is used when stripping outer connectives from a goal in a more delicate way than `STRIP_TAC`. A typical application is to keep stripping by using the tactic `REPEAT (FILTER_STRIP_TAC u)` until one hits the term `u` at which stripping is to stop.

See also

Tactic.CONJ_TAC, Tactic.FILTER_DISCH_TAC, Tactic.FILTER_DISCH_THEN, Tactic.FILTER_GEN_TAC, Tactic.STRIP_ASSUME_TAC, Tactic.STRIP_TAC

FILTER_STRIP_THEN (Tactic)

```
Tactic.FILTER_STRIP_THEN : (thm_tactic -> term -> tactic)
```

Conditionally strips a goal, handing an antecedent to the theorem-tactic.

Given a theorem-tactic `ttac`, a term `u` and a goal `(A,t)`, `FILTER_STRIP_THEN ttac u` removes one outer connective (`!`, `==>`, or `~`) from `t`, if the term being stripped does not contain a free instance of `u`. A negation `~t` is treated as the implication `t ==> F`. The theorem-tactic `ttac` is applied only when stripping an implication, by using the antecedent stripped off. `FILTER_STRIP_THEN` also breaks conjunctions.

`FILTER_STRIP_THEN` behaves like `STRIP_GOAL_THEN`, if the term being stripped does not contain a free instance of `u`. In particular, `FILTER_STRIP_THEN STRIP_ASSUME_TAC` behaves like `FILTER_STRIP_TAC`.

Failure

`FILTER_STRIP_THEN ttac u (A,t)` fails if `t` is not a universally quantified term, an implication, a negation or a conjunction; or if the term being stripped contains the term `u` (conjunction excluded); or if the application of `ttac` fails, after stripping the goal.

Example

When solving the goal

```
?- (n = 1) ==> (n * n = n)
```

the application of `FILTER_STRIP_THEN SUBST1_TAC "m:num"` results in the goal

```
?- 1 * 1 = 1
```

`FILTER_STRIP_THEN` is used when manipulating intermediate results using theorem-tactics, after stripping outer connectives from a goal in a more delicate way than `STRIP_GOAL_THEN`.

See also

`Tactic.CONJ_TAC`, `Tactic.FILTER_DISCH_TAC`, `Tactic.FILTER_DISCH_THEN`, `Tactic.FILTER_GEN_TAC`, `Tactic.FILTER_STRIP_TAC`, `Tactic.STRIP_ASSUME_TAC`, `Tactic.STRIP_GOAL_THEN`

find (DB)

```
DB.find : string -> data list
```

Search for theory element by name fragment.

An invocation `DB.find s` returns a list of theory elements which have been stored with a name containing a substring matching the regular expression `s`, ignoring case distinctions. All currently loaded theory segments are searched. The regular expression notation allows parentheses, dot (`.`) to match any character, Kleene star (`*`), alternation (`|`) and a special form of intersection (`~`).

The tilde form `r~s` is defined to be equal to `(.*r.*)&(.*s.*)`, where `&` is regular expression intersection. This allows one to require multiple sub-string matches: in a string such as `s1~s2`, matches will be found if the name contains both `s1` and `s2`, in either order.

Failure

Never fails. If nothing suitable can be found, the empty list is returned.

Example

```
> DB.find "inc";
val it =
   [(("arithmetic", "MULT_INCREASES"),
     (⊢ ∀m n. 1 < m ∧ 0 < n ⇒ SUC n ≤ m * n, Thm,
      Located
       {exact = true, linenum = 3572, scriptpath =
        "$(HOLDIR)/src/num/theories/arithmeticScript.sml"})),
    (("arithmetic", "STRICTLY_INCREASING_ONE_ONE"),
```

```
     (⊢ ∀f. (∀n. f n < f (SUC n)) ⇒ ONE_ONE f, Thm,
      Located
       {exact = true, linenum = 4282, scriptpath =
        "$(HOLDIR)/src/num/theories/arithmeticScript.sml"})),
    (("arithmetic", "STRICTLY_INCREASING_TC"),
     (⊢ ∀f. (∀n. f n < f (SUC n)) ⇒ ∀m n. m < n ⇒ f m < f n, Thm,
      Located
       {exact = true, linenum = 4275, scriptpath =
[...Output elided...]
```

Finding theorems in interactive proof sessions.

See also

DB.find_in, DB.match, DB.apropos, DB.selectDB, DB.thy, DB.theorems

find_consts (DB)

```
DB.find_consts : hol_type -> term list
```

Searches the current theory and its ancestors for constants matching given type.

Given a type ty searches the current theory and its ancestors for constants whose type matches the (“pattern”) type ty.

Failure

Never fails.

Example

If we run

```
> find_consts ``:'a -> 'a set -> bool``;
val it = [“$IN”]: term list
```

and with

```
> find_consts ``:num -> num -> num``;
val it =
   [“napp”, “ncons”, “$*,”, “internal_mult”, “numeral$onecount”,
    “numeral$texp_help”, “$*”, “$+”, “$-”, “ABS_DIFF”, “$DIV”,
    “$**”, “MAX”, “MIN”, “$MOD”, “ind_type$NUMPAIR”]: term list
```

The fact that type-matching is performed is apparent with this call:

```
> find_consts ``:'a -> 'a``;
val it =
   [“TL_T”, “common_prefixes”, “FRONT”, “REVERSE”, “TL”, “nub”,
    “COMPL”, “REST”, “tri$^{-1}$”, “nfst”, “nlen”, “nmap”, “nsnd”,
    “tri”, “numeral$exactlog”, “numeral$iDUB”, “numeral$iSQR”,
    “numeral$iZ”, “numeral$iiSUC”, “BIT1”, “BIT2”, “DIV2”, “FACT”,
    “NUMERAL”, “$&”, “PRE”, “SUC”, “SUC_REP”, “Abbrev”, “Cong”,
    “stmarker”, “unint”, “BOUNDED”, “LET”, “literal_case”, “$~”,
    “EQC”, “RC”, “RCOMPL”, “RTC”, “SC”, “STRORD”, “TC”, “I”,
    “NUMFST”, “NUMRIGHT”, “NUMSND”]: term list
```

where both `SUC` and `$~` (boolean negation) are among the list returned.

See also

`bossLib.find_consts_thy`, `DB.apropos`, `DB.find`

find_consts_thy (DB)

```
DB.find_consts_thy : string list -> hol_type -> term list
```

Searches in the theories in list `thl` for a constant matching given type `ty`.

A call to `find_consts_thy thl ty` searches the theories with names from `thl` for constants whose types match type `ty`, and returns that list.

Failure

Never fails.

Example

If we run

```
> find_consts_thy ["bool"] ``:'a -> 'a set -> bool``;
val it = [“$IN”]: term list
```

and

```
> find_consts_thy ["arithmetic"] ``:num -> num -> num``;
val it = [“$*”, “$+”, “$-”, “ABS_DIFF”, “$DIV”, “$**”, “MAX”, “MIN”,
          “$MOD”]: term list
```

See also

bossLib.find_consts, DB.apropos, DB.find

find_in (DB)

```
DB.find_in : string -> data list -> data list
```

Search for theory element by name fragment, among a given list.

An invocation `DB.find_in s data_list` selects from `data_list` those theory elements which have been stored with a name in which `s` occurs as a proper substring, ignoring case distinctions.

Failure

Never fails. If nothing suitable can be found, the empty list is returned.

Example

```
> DB.find "inc";
val it =
   [(("arithmetic", "MULT_INCREASES"),
```

```
     (⊢ ∀m n. 1 < m ∧ 0 < n ⇒ SUC n ≤ m * n, Thm,
      Located
       {exact = true, linenum = 3572, scriptpath =
        "$(HOLDIR)/src/num/theories/arithmeticScript.sml"})),
    (("arithmetic", "STRICTLY_INCREASING_ONE_ONE"),
     (⊢ ∀f. (∀n. f n < f (SUC n)) ⇒ ONE_ONE f, Thm,
      Located
       {exact = true, linenum = 4282, scriptpath =
        "$(HOLDIR)/src/num/theories/arithmeticScript.sml"})),
    (("arithmetic", "STRICTLY_INCREASING_TC"),
     (⊢ ∀f. (∀n. f n < f (SUC n)) ⇒ ∀m n. m < n ⇒ f m < f n, Thm,
      Located
       {exact = true, linenum = 4275, scriptpath =
[...Output elided...]

> DB.find_in "sum" it;
val it =
   [(("sum", "sum_distinct"),
     (⊢ ∀x y. INL x ≠ INR y, Thm,
      Located
       {exact = true, linenum = 259, scriptpath =
        "$(HOLDIR)/src/coretypes/sumScript.sml"})),
    (("sum", "sum_distinct1"),
     (⊢ ∀x y. INR y ≠ INL x, Thm,
      Located
       {exact = true, linenum = 269, scriptpath =
        "$(HOLDIR)/src/coretypes/sumScript.sml"}))]:
   DB_dtype.public_data_value named list
```

Finding theorems in interactive proof sessions. The second argument will normally be the result of a previous call to `DB.find`, `DB.match`, `DB.apropos`, `DB.listDB`, `DB.thy` etc.

See also

`DB.find`, `DB.match`, `DB.apropos`, `DB.listDB`, `DB.thy`, `DB.theorems`

find_term (HolKernel)

```
HolKernel.find_term : (term -> bool) -> term -> term
```

Finds a sub-term satisfying a predicate.

A call to `find_term P t` returns a sub-term of `t` that satisfies the predicate `P` if there is one. Otherwise it raises a `HOL_ERR` exception. The search is done in a top-down, left-to-right order (i.e., rators of combs are examined before rands).

Failure

Fails if the predicate fails when applied to any of the sub-terms of the term argument. Also fails if there is no sub-term satisfying the predicate.

Example

```
> find_term Literal.is_numeral ``2 + x + 3``;
val it = “2”: term

> find_term Literal.is_numeral ``x + y``;
Exception- HOL_ERR at HolKernel.find_term: raised
```

See also

`HolKernel.bvk_find_term`, `HolKernel.find_terms`

find_terms (HolKernel)

```
HolKernel.find_terms : (term -> bool) -> term -> term list
```

Traverses a term, returning a list of sub-terms satisfying a predicate.

A call to `find_terms P t` returns a list of sub-terms of `t` that satisfy `P`. The resulting list is ordered as if the traversal had been bottom-up and right-to-left (i.e., the rands of combs visited before their rators). The term `t` is itself considered a possible sub-term of `t`.

Failure

Only fails if the predicate fails on one of the term's sub-terms.

Example

```
> find_terms (fn _ => true) ``x + y``;
val it = [“y”, “x”, “$+”, “$+ x”, “x + y”]: term list

> find_terms Literal.is_numeral ``x + y``;
val it = []: term list

> find_terms Literal.is_numeral ``1 + x + 2 + y``;
val it = [“2”, “1”]: term list
```

See also

`HolKernel.bvk_find_term`, `HolKernel.find_term`

FINITE_CONV (pred_setLib)

```
pred_setLib.FINITE_CONV : conv
```

Proves finiteness of sets of the form `{t1;...;tn}`.

The conversion `FINITE_CONV` expects its term argument to be an assertion of the form `FINITE {t1;...;tn}`. Given such a term, the conversion returns the theorem

```
   |- FINITE {t1;...;tn} = T
```

Example

```
> pred_setLib.FINITE_CONV ``FINITE {1;2;3}``;
val it = ⊢ FINITE {1; 2; 3} ⇔ T: thm

> pred_setLib.FINITE_CONV ``FINITE ({}:num->bool)``;
val it = ⊢ FINITE ∅ ⇔ T: thm
```

Failure

Fails if applied to a term not of the form `FINITE {t1;...;tn}`.

first (Lib)

```
Lib.first : ('a -> bool) -> 'a list -> 'a
```

Return first element in list that predicate holds of.

An invocation `first P [x1,...,xk,...xn]` returns `xk` if `P xk` returns `true` and `P xi (1 <= i < k)` equals `false`.

Failure

If `P xi` is `false` for every element in `list`, then `first P list` raises an exception. When searching for an element of `list` that `P` holds of, it may happen that an application of `P` to an element of `list` raises an exception `e`. In that case, `first P list` also raises `e`.

Example

```
> first (fn i => i mod 2 = 0) [1,3,4,5];
val it = 4: int

> first (fn i => i mod 2 = 0) [1,3,5,7];
Exception- HOL_ERR (at Lib.first: unsatisfied predicate) raised
```

```
> first (fn _ => raise Fail "") [1];
Exception- Fail "" raised
```

See also

`Lib.exists`, `Lib.tryfind`, `Lib.all`

FIRST (Tactical)

```
Tactical.FIRST : (tactic list -> tactic)
```

Applies the first tactic in a tactic list which succeeds.

When applied to a list of tactics `[T1;...;Tn]`, and a goal `g`, the tactical `FIRST` tries applying the tactics to the goal until one succeeds. If the first tactic which succeeds is `Tm`, then the effect is the same as just `Tm`. Thus `FIRST` effectively behaves as follows:

```
FIRST [T1;...;Tn] = T1 ORELSE ... ORELSE Tn
```

Failure

The application of `FIRST` to a tactic list never fails. The resulting tactic fails iff all the component tactics do when applied to the goal, or if the tactic list is empty.

See also

`Tactical.EVERY`, `Tactical.ORELSE`

FIRST_ASSUM (Tactical)

```
Tactical.FIRST_ASSUM : (thm_tactic -> tactic)
```

Maps a theorem-tactic over the assumptions, applying first successful tactic.

The tactic

```
FIRST_ASSUM ttac ([A1; ...; An], g)
```

has the effect of applying the first tactic which can be produced by `ttac` from the `ASSUME`d assumptions `(A1 |- A1)`, ..., `(An |- An)` and which succeeds when applied to the goal. Failures of `ttac` to produce a tactic are ignored.

Failure

Fails if `ttac (Ai |- Ai)` fails for every assumption `Ai`, or if the assumption list is empty, or if all the tactics produced by `ttac` fail when applied to the goal.

Example

The tactic

```
FIRST_ASSUM (fn asm => CONTR_TAC asm  ORELSE  ACCEPT_TAC asm)
```

searches the assumptions for either a contradiction or the desired conclusion. The tactic

```
FIRST_ASSUM MATCH_MP_TAC
```

searches the assumption list for an implication whose conclusion matches the goal, reducing the goal to the antecedent of the corresponding instance of this implication.

Comments

By default, the assumption list is printed in reverse order, with the head of the list printed last. To process the assumption list in the opposite order, use `LAST_ASSUM`

See also

`Tactical.FIRST_X_ASSUM`, `Tactical.LAST_ASSUM`, `Tactical.ASSUM_LIST`, `Tactical.EVERY`, `Tactical.EVERY_ASSUM`, `Tactical.FIRST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`

FIRST_CONSEQ_CONV (ConseqConv)

```
ConseqConv.FIRST_CONSEQ_CONV : (conseq_conv list -> conseq_conv)
```

Apply the first of the conversions in a given list that succeeds.

See also

`ConseqConv.ORELSE_CONSEQ_CONV`, `Conv.FIRST_CONV`

FIRST_CONV (Conv)

```
Conv.FIRST_CONV : conv list -> conv
```

Apply the first of the conversions in a given list that succeeds.

`FIRST_CONV [c1,...,cn] t` returns the result of applying to the term `t` the first conversion `ci` that succeeds (or raises `UNCHANGED`) when applied to `t`. The conversions are tried in the order in which they are given in the list.

Failure

`FIRST_CONV [c1,...,cn] t` fails if all the conversions `c1`, `...`, `cn` fail when applied to the term `t`. `FIRST_CONV cs t` also fails if `cs` is the empty list.

See also

`Conv.ORELSEC`, `Conv.UNCHANGED`

first_fv_term (boolLib)

```
boolLib.first_fv_term : (term -> tactic) -> tactic
```

Applies a term-tactic to goal's first free variable that makes it succeed

A call to `first_fv_term tmtac` applies the function `tmtac` to all of a goal's free variables. This generates a list of tactics, which is then applied to the goal tactic-by-tactic (this is the action of the tactical `MAP_FIRST`). The first application that succeeds (doesn't raise a `HOL_ERR`) is taken as the result. Later applications are not attempted.

Failure

Fails if there is no free variable `v` in the goal `A ?- g` that makes the application `tmtac v (A?-g)` succeed.

See also

`Tactical.MAP_FIRST`

FIRST_LT (Tactical)

```
Tactical.FIRST_LT : tactic -> list_tactic
```

Applies a tactic to the first goal in the goal-list that works.

Given a list of goals `gl`, an application of `FIRST_LT tac` to `gl` will try to apply `tac` to each goal in `gl` in turn. If no goal lets `tac` succeed, the whole application fails also. Otherwise, the first goal on which `tac` succeeds will generate a (possibly empty) list of new sub-goals. These new sub-goals are pushed onto the front of the rest of `gl`.

Failure

The application of `FIRST_LT` to a tactic never fails. The resulting list-tactic fails if the goal list is empty, or if argument `tac` fails on each goal in the list `gl`.

Example

```
> FIRST_LT CONJ_TAC [([], “p ⇒ q”), ([“a ∨ b”], “p /\ q”)]
```

```
val it = ([(["a ∨ b"], "p"), (["a ∨ b"], "q"), ([], "p ⇒ q")], fn):
   goal list * list_validation
```

See also

Tactical.THEN_LT, Tactical.HEADGOAL

FIRST_PROVE (Tactical)

```
Tactical.FIRST_PROVE : (tactic list -> tactic)
```

Applies the first tactic in a tactic list which completely proves the goal.

When applied to a list of tactics `[T1;...;Tn]`, and a goal `g`, the tactical `FIRST_PROVE` tries applying the tactics to the goal until one proves the goal. If the first tactic which proves the goal is `Tm`, then the effect is the same as just `Tm`. Thus `FIRST_PROVE` effectively behaves as follows:

```
FIRST_PROVE [T1;...;Tn] = (T1 THEN NO_TAC) ORELSE ... ORELSE (Tn THEN NO_TAC)
```

Failure

The application of `FIRST_PROVE` to a tactic list never fails. The resulting tactic fails iff none of the supplied tactics completely proves the goal by itself, or if the tactic list is empty.

See also

Tactical.EVERY, Tactical.ORELSE, Tactical.FIRST

FIRST_TCL (Thm_cont)

```
Thm_cont.FIRST_TCL : (thm_tactical list -> thm_tactical)
```

Applies the first theorem-tactical in a list which succeeds.

When applied to a list of theorem-tacticals, a theorem-tactic and a theorem, FIRST_TCL returns the tactic resulting from the application of the first theorem-tactical to the theorem-tactic and theorem which succeeds. The effect is the same as:

```
FIRST_TCL [ttl1;...;ttln] = ttl1 ORELSE_TCL ... ORELSE_TCL ttln
```

Failure

FIRST_TCL fails iff each tactic in the list fails when applied to the theorem-tactic and theorem. This is trivially the case if the list is empty.

See also

Thm_cont.EVERY_TCL, Thm_cont.ORELSE_TCL, Thm_cont.REPEAT_TCL, Thm_cont.THEN_TCL

FIRST_X_ASSUM (Tactical)

```
Tactical.FIRST_X_ASSUM : thm_tactic -> tactic
```

Maps a theorem-tactic over the assumptions, applying first successful tactic and removing the assumption that gave rise to the successful tactic.

The tactic

```
FIRST_X_ASSUM ttac ([A1; ...; An], g)
```

has the effect of applying the first tactic which can be produced by `ttac` from the ASSUMEd assumptions `(A1 |- A1)`, ..., `(An |- An)` and which succeeds when applied to the goal. The assumption which produced the successful theorem-tactic is removed from the assumption list (before `ttac` is applied). Failures of `ttac` to produce a tactic are ignored.

Failure

Fails if `ttac (Ai |- Ai)` fails for every assumption `Ai`, or if the assumption list is empty, or if all the tactics produced by `ttac` fail when applied to the goal.

Example

The tactic

```
FIRST_X_ASSUM SUBST_ALL_TAC
```

searches the assumptions for an equality and causes its right hand side to be substituted for its left hand side throughout the goal and assumptions. It also removes the equality from the assumption list. Using `FIRST_ASSUM` above would leave an equality on the assumption list of the form `x = x`. The tactic

```
FIRST_X_ASSUM MATCH_MP_TAC
```

searches the assumption list for an implication whose conclusion matches the goal, reducing the goal to the antecedent of the corresponding instance of this implication and removing the implication from the assumption list.

Comments

The "X" in the name of this tactic is a mnemonic for the "crossing out" or removal of the assumption found.

By default, the assumption list is printed in reverse order, with the head of the list printed last. To process the assumption list in the opposite order, use `LAST_X_ASSUM`

See also

`Tactical.FIRST_ASSUM`, `Tactical.LAST_X_ASSUM`, `Tactical.ASSUM_LIST`, `Tactical.EVERY`, `Tactical.PAT_ASSUM`, `Tactical.EVERY_ASSUM`, `Tactical.FIRST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`, `Thm_cont.UNDISCH_THEN`

FIRSTN_CONV (listLib)

`listLib.FIRSTN_CONV : conv`

Computes by inference the result of taking the initial n elements of a list.

For any object language list of the form "`[x0;...x(n-k);...;x(n-1)]`" , the result of evaluating

```
FIRSTN_CONV “FIRSTN k [x0;...x(n-k);...;x(n-1)]”
```

is the theorem

```
|- FIRSTN k [x0;...;x(n-k);...;x(n-1)] = [x0;...;x(n-k)]
```

Failure

`FIRSTN_CONV tm` fails if `tm` is not of the form described above, or `k` is greater than the length of the list.

FLAT_CONV (listLib)

```
listLib.FLAT_CONV : conv
```

Computes by inference the result of flattening a list of lists.

`FLAT_CONV` takes a term `tm` in the following form:

```
FLAT [[x00;...x0n]; ...; [xm0;...xmn]]
```

It returns the theorem

```
|- FLAT [[x00;...x0n];...;[xm0;...xmn]] = [x00;...x0n;...;xm0;...xmn]
```

Failure

`FLAT_CONV tm` fails if `tm` is not of the form described above.

Example

Evaluating

```
FLAT_CONV “FLAT [[0;2;4];[0;1;2;3;4]]”;
```

returns the following theorem:

```
|- FLAT[[0;2;4];[0;1;2;3;4]] = [0;2;4;0;1;2;3;4]
```

See also

listLib.FOLDL_CONV, listLib.FOLDR_CONV, listLib.list_FOLD_CONV

flatn (proofManagerLib)

```
proofManagerLib.flatn : int -> unit
```

Flattens the tree structure of subgoals on the subgoal package goal stack.

The function `flatn` is part of the subgoal package. For a general description of the subgoal package, see `set_goal`.

The `flatn` function's basic step of operation is to take the the current list of sub-goals and concatenate it with the previous list of subgoals (excluding the first of that list, from which the current list was obtained). The numeric argument passed to `flatn` specifies how many times this operation is to be performed.

Failure

Raises the `NO_PROOFS` exception if there is no current proof manipulated by the subgoal package.

If `n` is too large, or negative, the result will be a flat list of all subgoals.

To view, reorder, or attack simultaneously, current and previously obtained subgoals.

See also

proofManagerLib.set_goal, proofManagerLib.restart, proofManagerLib.backup, proofManagerLib.redo, proofManagerLib.restore, proofManagerLib.save, proofManagerLib.set_backup, proofManagerLib.expand, proofManagerLib.expandf, proofManagerLib.p, proofManagerLib.top_thm, proofManagerLib.top_goal

flatten (Lib)

```
Lib.flatten : 'a list list -> 'a list
```

Removes one level of bracketing from a list.

An invocation `flatten [[x11,...,x1k1],...,[xn1,...,xnkn]]` yields the list `[x1,...,x1k1,...,`

Failure

Never fails.

Example

```
> flatten [[1,2,3],[],[4,5]];
val it = [1, 2, 3, 4, 5]: int list

> flatten ([[[]]] : int list list list);
val it = [[]]: int list list
```

FLATTEN_CONJ_CONV (unwindLib)

```
unwindLib.FLATTEN_CONJ_CONV : conv
```

Flattens a 'tree' of conjunctions.

`FLATTEN_CONJ_CONV "t1 /\ ... /\ tn"` returns a theorem of the form:

```
   |- t1 /\ ... /\ tn = u1 /\ ... /\ un
```

where the right-hand side of the equation is a flattened version of the left-hand side.

Failure

Never fails.

Example

```
#FLATTEN_CONJ_CONV "(a /\ (b /\ c)) /\ ((d /\ e) /\ f)";;
|- (a /\ b /\ c) /\ (d /\ e) /\ f = a /\ b /\ c /\ d /\ e /\ f
```

FOLDL_CONV (listLib)

```
listLib.FOLDL_CONV : conv -> conv
```

Computes by inference the result of applying a function to the elements of a list.

`FOLDL_CONV` takes a conversion `conv` and a term `tm` in the following form:

```
FOLDL f e [x0;...xn]
```

It returns the theorem

```
|- FOLDL f e [x0;...xn] = tm'
```

where `tm'` is the result of applying the function `f` iteratively to the successive elements of the list and the result of the previous application starting from the tail end of the list. During each iteration, an expression `f ei xi` is evaluated. The user supplied conversion `conv` is used to derive a theorem

```
|- f ei xi = e(i+1)
```

which is used in the next iteration.

Failure

`FOLDL_CONV conv tm` fails if `tm` is not of the form described above.

Example

To sum the elements of a list, one can use `FOLDL_CONV` with `REDUCE_CONV` from the library `numLib`.

```
- FOLDL_CONV numLib.REDUCE_CONV ``FOLDL $+ 0 [0;1;2;3]``;
val it = |- FOLDL $+ 0 [0;1;2;3] = 6 : thm
```

In general, if the function `f` is an explicit lambda abstraction `(\x x'. t[x,x'])`, the conversion should be in the form

```
((RATOR_CONV BETA_CONV) THENC BETA_CONV THENC conv'))
```

where `conv'` applied to `t[x,x']` returns the theorem

```
|-t[x,x'] = e''.
```

See also

`listLib.FOLDR_CONV`, `listLib.list_FOLD_CONV`

FOLDR_CONV (listLib)

```
listLib.FOLDR_CONV : conv -> conv
```

Computes by inference the result of applying a function to the elements of a list.

`FOLDR_CONV` takes a conversion `conv` and a term `tm` in the following form:

```
FOLDR f e [x0;...xn]
```

It returns the theorem

```
|- FOLDR f e [x0;...xn] = tm'
```

where `tm'` is the result of applying the function `f` iteratively to the successive elements of the list and the result of the previous application starting from the tail end of the list. During each iteration, an expression `f xi ei` is evaluated. The user supplied conversion `conv` is used to derive a theorem

```
|- f xi ei = e(i+1)
```

which is used in the next iteration.

Failure

`FOLDR_CONV conv tm` fails if `tm` is not of the form described above.

Example

To sum the elements of a list, one can use `FOLDR_CONV` with `REDUCE_CONV` from the library `numLib`.

```
- FOLDR_CONV numLib.REDUCE_CONV ``FOLDR $+ 0 [0;1;2;3]``;
val it = |- FOLDR $+ 0[0;1;2;3] = 6 : thm
```

In general, if the function `f` is an explicit lambda abstraction `(\x x'. t[x,x'])`, the conversion should be in the form

```
((RATOR_CONV BETA_CONV) THENC BETA_CONV THENC conv'))
```

where `conv'` applied to `t[x,x']` returns the theorem

```
|-t[x,x'] = e''.
```

See also

`listLib.FOLDL_CONV`, `listLib.list_FOLD_CONV`

for (Lib)

```
Lib.for : int -> int -> (int -> 'a) -> 'a list
```

Functional ‘for’ loops.

An application `for b t f` is equal to `[f b, f (b+1), ..., f t]`. If `b` is greater than `t`, the empty list is returned.

Failure

If `f i` fails for `b <= i <= t`.

Example

```
> for 97 122 Char.chr;
val it =
   [#"a", #"b", #"c", #"d", #"e", #"f", #"g", #"h", #"i", #"j", #"k", #"l",
    #"m", #"n", #"o", #"p", #"q", #"r", #"s", #"t", #"u", #"v", #"w", #"x",
    #"y", #"z"]: char list
```

See also

`Lib.for_se`

for_se (Lib)

```
Lib.for_se : int -> int -> (int -> unit) -> unit
```

Side-effecting 'for' loops.

An application `for_se b t f` is equal to `(f b; f (b+1); ...; f t)`. If `b` is greater than `t`, then `for_se b t f` does no evaluation, in particular `f b` is not evaluated.

Failure

If `f i` fails for `b <= i <= t`.

Example

```
> let val A = Array.array(26,#" ")
  in
    for_se 0 25 (fn i => Array.update(A,i, Char.chr (i+97)))
  ; for_se 0 25 (Feedback.HOL_INFO o Char.toString o curry Array.sub A)
  ; Feedback.HOL_INFO "\n"
  end;
abcdefghijklmnopqrstuvwxyz
val it = (): unit
```

See also

`Lib.for`

FORALL_AND_CONV (Conv)

```
Conv.FORALL_AND_CONV : conv
```

Moves a universal quantification inwards through a conjunction.

When applied to a term of the form `!x. P /\ Q`, the conversion `FORALL_AND_CONV` returns the theorem:

```
|- (!x. P /\ Q) = (!x.P) /\ (!x.Q)
```

Failure

Fails if applied to a term not of the form `!x. P /\ Q`.

See also

`Conv.AND_FORALL_CONV`, `Conv.LEFT_AND_FORALL_CONV`, `Conv.RIGHT_AND_FORALL_CONV`

FORALL_ARITH_CONV (Arith)

```
Arith.FORALL_ARITH_CONV : conv
```

Partial decision procedure for non-existential Presburger natural arithmetic.

`FORALL_ARITH_CONV` is a partial decision procedure for formulae of Presburger natural arithmetic which are in prenex normal form and have all variables either free or universally quantified. Presburger natural arithmetic is the subset of arithmetic formulae made up from natural number constants, numeric variables, addition, multiplication by a constant, the relations <, <=, =, >=, > and the logical connectives ~, /\, \/, ==>, =

(if-and-only-if), ! ('forall') and ? ('there exists'). Products of two expressions which both contain variables are not included in the subset, but the function SUC which is not normally included in a specification of Presburger arithmetic is allowed in this HOL implementation.

Given a formula in the specified subset, the function attempts to prove that it is equal to T (true). The procedure only works if the formula would also be true of the non-negative rationals; it cannot prove formulae whose truth depends on the integral properties of the natural numbers.

Failure

The function can fail in two ways. It fails if the argument term is not a formula in the specified subset, and it also fails if it is unable to prove the formula. The failure strings are different in each case.

Example

```
#FORALL_ARITH_CONV "m < SUC m";;
|- m < (SUC m) = T

#FORALL_ARITH_CONV "!m n p q. m <= p /\ n <= q ==> (m + n) <= (p + q)";;
|- (!m n p q. m <= p /\ n <= q ==> (m + n) <= (p + q)) = T

#FORALL_ARITH_CONV "!m n. ~(SUC (2 * m) = 2 * n)";;
evaluation failed     FORALL_ARITH_CONV -- cannot prove formula
```

See also

Arith.NEGATE_CONV, Arith.EXISTS_ARITH_CONV, numLib.ARITH_CONV, Arith.ARITH_FORM_NORM_CONV, Arith.DISJ_INEQS_FALSE_CONV

FORALL_CONJ_CONV (unwindLib)

unwindLib.FORALL_CONJ_CONV : conv

Moves universal quantifiers down through a tree of conjunctions.

`FORALL_CONJ_CONV "!x1 ... xm. t1 /\ ... /\ tn"` returns the theorem:

```
|- !x1 ... xm. t1 /\ ... /\ tn =
   (!x1 ... xm. t1) /\ ... /\ (!x1 ... xm. tn)
```

where the original term can be an arbitrary tree of conjunctions. The structure of the tree is retained in both sides of the equation.

Failure

Never fails.

Example

```
#FORALL_CONJ_CONV "!(x:*) (y:*) (z:*). (a /\ b) /\ c";;
|- (!x y z. (a /\ b) /\ c) = ((!x y z. a) /\ (!x y z. b)) /\ (!x y z. c)

#FORALL_CONJ_CONV "T";;
|- T = T

#FORALL_CONJ_CONV "!(x:*) (y:*) (z:*). T";;
|- (!x y z. T) = (!x y z. T)
```

See also

`unwindLib.CONJ_FORALL_CONV`, `unwindLib.FORALL_CONJ_ONCE_CONV`, `unwindLib.CONJ_FORALL_ONCE_CONV`, `unwindLib.FORALL_CONJ_RIGHT_RULE`, `unwindLib.CONJ_FORALL_RIGHT_RULE`

FORALL_CONJ_ONCE_CONV (unwindLib)

`unwindLib.FORALL_CONJ_ONCE_CONV : conv`

Moves a single universal quantifier down through a tree of conjunctions.

`FORALL_CONJ_ONCE_CONV "!x. t1 /\ ... /\ tn"` returns the theorem:

```
|- !x. t1 /\ ... /\ tn = (!x. t1) /\ ... /\ (!x. tn)
```

where the original term can be an arbitrary tree of conjunctions. The structure of the tree is retained in both sides of the equation.

Failure

Fails if the argument term is not of the required form. The body of the term need not be a conjunction.

Example

```
#FORALL_CONJ_ONCE_CONV "!x. ((x \/ a) /\ (x \/ b)) /\ (x \/ c)";;
|- (!x. ((x \/ a) /\ (x \/ b)) /\ (x \/ c)) =
   ((!x. x \/ a) /\ (!x. x \/ b)) /\ (!x. x \/ c)

#FORALL_CONJ_ONCE_CONV "!x. x \/ a";;
|- (!x. x \/ a) = (!x. x \/ a)

#FORALL_CONJ_ONCE_CONV "!x. ((x \/ a) /\ (y \/ b)) /\ (x \/ c)";;
|- (!x. ((x \/ a) /\ (y \/ b)) /\ (x \/ c)) =
   ((!x. x \/ a) /\ (!x. y \/ b)) /\ (!x. x \/ c)
```

See also

unwindLib.CONJ_FORALL_ONCE_CONV, unwindLib.FORALL_CONJ_CONV, unwindLib.CONJ_FORALL_CONV, unwindLib.FORALL_CONJ_RIGHT_RULE, unwindLib.CONJ_FORALL_RIGHT_RULE

FORALL_CONJ_RIGHT_RULE (unwindLib)

```
unwindLib.FORALL_CONJ_RIGHT_RULE : (thm -> thm)
```

Moves universal quantifiers down through a tree of conjunctions.

```
    A |- !z1 ... zr. t = ?y1 ... yp. !x1 ... xm. t1 /\ ... /\ tn
------------------------------------------------------------------------
 A |- !z1 ... zr.
       t = ?y1 ... yp. (!x1 ... xm. t1) /\ ... /\ (!x1 ... xm. tn)
```

Failure

Fails if the argument theorem is not of the required form, though either or both of `r` and `p` may be zero.

See also

`unwindLib.CONJ_FORALL_RIGHT_RULE`, `unwindLib.FORALL_CONJ_CONV`, `unwindLib.CONJ_FORALL_CONV`, `unwindLib.FORALL_CONJ_ONCE_CONV`, `unwindLib.CONJ_FORALL_ONCE_CONV`

FORALL_CONSEQ_CONV (ConseqConv)

```
ConseqConv.FORALL_CONSEQ_CONV : (conseq_conv -> conseq_conv)
```

Applies a consequence conversion to the body of a universally-quantified term.

If `c` is a consequence conversion that maps a term ``` ``t x`` ``` to a theorem `|- t x = t' x`, `|- t' x ==> t x` or `|- t x ==> t' x`, then `FORALL_CONSEQ_CONV c` maps ``` ``!x. t x`` ``` to `|- !x. t x = !x. t' x`, `|- !x. t' x ==> !x. t x` or `|- !x. t x ==> !x. t' x`, respectively.

Failure

`FORALL_CONSEQ_CONV c t` fails, if `t` is not an all-quantified term or if `c` fails on the body of `t`.

See also

`Conv.QUANT_CONV`, `ConseqConv.EXISTS_CONSEQ_CONV`, `ConseqConv.QUANT_CONSEQ_CONV`

FORALL_EQ (Drule)

```
Drule.FORALL_EQ : (term -> thm -> thm)
```

Universally quantifies both sides of an equational theorem.

When applied to a variable `x` and a theorem `A |- t1 = t2`, whose conclusion is an equation between boolean terms, `FORALL_EQ` returns the theorem `A |- (!x. t1) = (!x. t2)`, unless the variable `x` is free in any of the assumptions.

```
        A |- t1 = t2
------------------------  FORALL_EQ "x"      [where x is not free in A]
 A |- (!x.t1) = (!x.t2)
```

Failure

Fails if the theorem is not an equation between boolean terms, or if the supplied term is not simply a variable, or if the variable is free in any of the assumptions.

See also

`Thm.AP_TERM`, `Drule.EXISTS_EQ`, `Drule.SELECT_EQ`

FORALL_EQ___CONSEQ_CONV (ConseqConv)

```
ConseqConv.FORALL_EQ___CONSEQ_CONV : conseq_conv
```

Given a term of the form `(!x. P x) = (!x. Q x)` this consequence conversion returns the theorem `|- (!x. (P x = Q x)) ==> ((!x. P x) = (!x. Q x))`.

See also

`ConseqConv.conseq_conv`

FORALL_IMP_CONV (Conv)

```
Conv.FORALL_IMP_CONV : conv
```

Moves a universal quantification inwards through an implication.

When applied to a term of the form `!x. P ==> Q`, where `x` is not free in both `P` and `Q`, `FORALL_IMP_CONV` returns a theorem of one of three forms, depending on occurrences of the variable `x` in `P` and `Q`. If `x` is free in `P` but not in `Q`, then the theorem:

```
|- (!x. P ==> Q) = (?x.P) ==> Q
```

is returned. If `x` is free in `Q` but not in `P`, then the result is:

```
|- (!x. P ==> Q) = P ==> (!x.Q)
```

And if `x` is free in neither `P` nor `Q`, then the result is:

```
|- (!x. P ==> Q) = (?x.P) ==> (!x.Q)
```

Failure

`FORALL_IMP_CONV` fails if it is applied to a term not of the form `!x. P ==> Q`, or if it is applied to a term `!x. P ==> Q` in which the variable `x` is free in both `P` and `Q`.

See also

`Conv.LEFT_IMP_EXISTS_CONV`, `Conv.RIGHT_IMP_FORALL_CONV`

FORALL_NOT_CONV (Conv)

```
Conv.FORALL_NOT_CONV : conv
```

Moves a universal quantification inwards through a negation.

When applied to a term of the form `!x.~P`, the conversion `FORALL_NOT_CONV` returns the theorem:

```
|- (!x.~P) = ~(?x. P)
```

Failure

Fails if applied to a term not of the form `!x.~P`.

See also

`Conv.EXISTS_NOT_CONV`, `Conv.NOT_EXISTS_CONV`, `Conv.NOT_FORALL_CONV`

FORALL_OR_CONV (Conv)

```
Conv.FORALL_OR_CONV : conv
```

Moves a universal quantification inwards through a disjunction.

When applied to a term of the form `!x. P \/ Q`, where `x` is not free in both `P` and `Q`, `FORALL_OR_CONV` returns a theorem of one of three forms, depending on occurrences of the variable `x` in `P` and `Q`. If `x` is free in `P` but not in `Q`, then the theorem:

```
|- (!x. P \/ Q) = (!x.P) \/ Q
```

is returned. If `x` is free in `Q` but not in `P`, then the result is:

```
|- (!x. P \/ Q) = P \/ (!x.Q)
```

And if `x` is free in neither `P` nor `Q`, then the result is:

```
|- (!x. P \/ Q) = (!x.P) \/ (!x.Q)
```

Failure

`FORALL_OR_CONV` fails if it is applied to a term not of the form `!x. P \/ Q`, or if it is applied to a term `!x. P \/ Q` in which the variable `x` is free in both `P` and `Q`.

See also

`Conv.OR_FORALL_CONV`, `Conv.LEFT_OR_FORALL_CONV`, `Conv.RIGHT_OR_FORALL_CONV`

force_add (simpLib)

```
simpLib.force_add : simpset -> ssfrag -> simpset
```

Adds a simpset fragment, overriding any active exclusion of its name.

A call to `force_add simpset frag` returns a simpset that is `simpset` augmented with `frag` (as if by ++). In addition, if `frag` has a name that is currently in the simpset's *excluded* set (because of an earlier `exclude_ssfrags` call), that name is removed from the excluded set so that subsequent ++ of a fragment with the same name will also succeed.

This function is the override mechanism for `exclude_ssfrags`: while ++ is silent when the incoming fragment is currently excluded, `force_add` lifts the exclusion for that name and performs the addition.

`force_add` is what underlies the behaviour of the `SF` marker in thm-list arguments to simplification tactics: writing `simp[SF FRAG_ss]` inside a `Proof[exclude_frags = FRAG]` body re-enables `FRAG_ss` for that simp call.

Failure

Never fails.

See also

`simpLib.exclude_ssfrags`, `simpLib.++`, `bossLib.SF`

forget_history (proofManagerLib)

```
proofManagerLib.forget_history : unit -> unit
```

Clears the proof history.

The function `forget_history` is part of the subgoal package. A call to `forget_history` clears the history of saved proof states. Subsequent calls to `backup` or `restart` will behave as if the initial goal was the state at the time of the call to `forget_history`. For a description of the subgoal package, see `set_goal`.

Failure

The function `forget_history` only fails if no goalstack is being managed.

Hiding an automatic preprocessing phase of a proof before handing it to the user.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.s` `backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

FORK_CONV (Conv)

```
Conv.FORK_CONV : (conv * conv) -> conv
```

Applies a pair of conversions to the arguments of a binary operator.

If the conversion `c1` maps a term `t1` to the theorem `|- t1 = t1'`, and the conversion `c2` maps `t2` to `|- t2 = t2'`, then the conversion `FORK_CONV (c1,c2)` maps terms of the form `f t1 t2` to theorems of the form `|- f t1 t2 = f t1' t2'`.

Failure

`FORK_CONV (c1,c2) t` will fail if `t` is not of the general form `f t1 t2`, or if `c1` fails when applied to `t1`, or if `c2` fails when applied to `t2`, or if `c1` or `c2` aren't really conversions, and thereby fail to return appropriate equational theorems.

Example

```
> FORK_CONV (BETA_CONV,reduceLib.REDUCE_CONV) (Term`(\x. x + 1)y * (10 DIV 3)`);
val it = ⊢ (λx. x + 1) y * (10 DIV 3) = (y + 1) * 3: thm
```

See also

Conv.BINOP_CONV, Conv.LAND_CONV, Conv.RAND_CONV, Conv.RATOR_CONV, numLib.REDUCE_CONV

format_ERR (Feedback)

```
Feedback.format_ERR : error_record -> string
```

Maps argument record of HOL_ERR to a string.

The format_ERR function maps the argument of an application of HOL_ERR to a string. It is the default function used by ERR_to_string.

Failure

Never fails.

Example

```
> print
  (format_ERR {origin_structure = "Foo",
               origin_function = "bar",
               message = "incomprehensible input"});
Exception- Type error in function application.
   Function: format_ERR : hol_error -> string
   Argument:
      {origin_structure = "Foo", origin_function = "bar",
        message = "incomprehensible input"} :
      {message: string,
```

```
        origin_function: string, origin_structure: string}
   Reason:
      Can't unify hol_error to
         {message: string,
           origin_function: string, origin_structure: string}
         (Incompatible types)
Fail "Static Errors" raised
```

See also

Feedback, Feedback.ERR_to_string, Feedback.format_MESG, Feedback.format_WARNING

format_MESG (Feedback)

```
Feedback.format_MESG : string -> string
```

Maps argument of HOL_MESG to a string.

The format_MESG function maps a string to a string. Usually, the input string is the argument of an invocation of HOL_MESG. format_MESG is the default function used by MESG_to_string.

Failure

Never fails.

Example

```
> format_MESG "Hello world.";
val it = "<<HOL message: Hello world.>>\n": string
```

See also

Feedback, Feedback.MESG_to_string, Feedback.format_ERR, Feedback.format_WARNING

format_WARNING (Feedback)

```
Feedback.format_WARNING : string -> string -> string -> string
```

Maps arguments of `HOL_WARNING` to a string.

The `format_WARNING` function maps three strings to a string. Usually, the input strings are the arguments to an invocation of `HOL_WARNING`. `format_WARNING` is the default function used by `WARNING_to_string`.

Failure

Never fails.

Example

```
> format_WARNING "Module" "function" "Gadzooks!";
val it = "<<HOL warning: Module.function: Gadzooks!>>\n": string
```

See also

`Feedback`, `Feedback.WARNING_to_string`, `Feedback.format_ERR`, `Feedback.format_MESG`

free_in (Term)

```
Term.free_in : term -> term -> bool
```

Tests if one term is free in another.

When applied to two terms `t1` and `t2`, the function `free_in` returns `true` if `t1` is free in `t2`, and `false` otherwise. It is not necessary that `t1` be simply a variable. A term `M` occurs free in `N` when there is some occurrence of `M` in `N` such that each free variable of `M` in that occurrence is not bound by a binder in `N`.

Failure

Never fails.

Example

In the following example `free_in` returns `false` because the `x` in `SUC x` in the second term is bound:

```
- free_in ``SUC x`` ``!x. SUC x = x + 1``;
> val it = false : bool
```

whereas the following call returns `true` because the first instance of `x` in the second term is free, even though there is also a bound instance:

```
- free_in ``x:bool`` ``!y. x /\ ?x. x = y``;
> val it = true : bool
```

See also

`Term.free_vars`, `Term.FVL`

free_vars (Term)

```
Term.free_vars : term -> term list
```

Returns the set of free variables in a term.

An invocation `free_vars tm` returns a list representing the set of term variables occurring in `tm`.

Failure

Never fails.

Example

```
> free_vars (Term `x /\ y /\ y ==> x`);
val it = [“y”, “x”]: term list
```

Comments

Code should not depend on how elements are arranged in the result of `free_vars`.

`free_vars` is not efficient for large terms with many free variables. Demanding applications should be coded with `FVL`.

See also

`Term.FVL`, `Term.free_vars_lr`, `Term.free_varsl`, `Term.empty_varset`, `Type.type_vars`

free_vars_lr (Term)

```
Term.free_vars_lr : term -> term list
```

Returns the set of free variables in a term, in order.

An invocation `free_vars_lr ty` returns a list representing the set of type variables occurring in `ty`. The list will be in order of variable occurrence when scanning the parse tree of the term from left to right. This is usually, but need not be, the textual order when the term is printed.

Failure

Never fails.

Example

```
> free_vars_lr (Term `x /\ y /\ y ==> z`);
val it = [“x”, “y”, “z”]: term list
```

Comments

`free_vars_lr` is not efficient for large terms with many free variables. More strenuous applications should use high performance set implementations available in the Standard ML Basis Library.

`free_vars_lr` can be used to build pleasing quantifier prefixes.

See also

`Term.FVL`, `Term.free_vars`, `Term.empty_varset`, `Type.type_vars`

free_varsl (Term)

```
Term.free_varsl : term list -> term list
```

Returns the set of free variables in a list of terms.

An invocation `free_varsl [t1,...,tn]` returns a list representing the set of free term variables occurring in `t1,...,tn`.

Failure

Never fails.

Example

```
> free_varsl [Term `x /\ y /\ y ==> x`,
              Term `!x. x ==> p ==> y`];
val it = [“x”, “y”, “p”]: term list
```

Comments

Code should not depend on how elements are arranged in the result of `free_varsl`.

`free_varsl` is not efficient for large terms with many free variables. Demanding applications should be coded with `FVL`.

See also

`Term.FVL`, `Term.free_vars_lr`, `Term.free_vars`, `Term.empty_varset`, `Type.type_vars`

frees (hol88Lib)

```
hol88Lib.frees : term -> term list
```

Returns a list of the variables which are free in a term.

`frees` is equivalent to `rev o Term.free_vars`.

Failure

Never fails.

Comments

Superseded by `Term.free_vars`.

See also

`hol88Lib.freesl`, `Term.free_vars`

FREEZE_THEN (Tactic)

```
Tactic.FREEZE_THEN : thm_tactical
```

'Freezes' a theorem to prevent instantiation of its free variables.

`FREEZE_THEN` expects a tactic-generating function `f:thm->tactic` and a theorem `(A1 |- w)` as arguments. The tactic-generating function `f` is applied to the theorem `(w |- w)`. If this tactic generates the subgoal:

```
 A0 ?- t
=========  f (w |- w)
  A ?- t1
```

then applying `FREEZE_THEN f (A1 |- w)` to the goal `(A0 ?- t)` produces the subgoal:

```
          A0 ?- t
==================  FREEZE_THEN f (A1 |- w)
 A - {w}, A1 ?- t1
```

Since the term `w` is a hypothesis of the argument to the function `f`, none of the free variables present in `w` may be instantiated or generalized. The hypothesis is discharged by `PROVE_HYP` upon the completion of the proof of the subgoal.

Failure

Failures may arise from the tactic-generating function. An invalid tactic arises if the hypotheses of the theorem are not alpha-convertible to assumptions of the goal.

Example

Given the goal `([``b < c``, ``a < b``], ``SUC a <= c``)`, and the specialized variant of the theorem `LESS_TRANS`:

```
th = |- !p. a < b /\ b < p ==> a < p
```

`IMP_RES_TAC th` will generate several unneeded assumptions:

```
{b < c, a < b, a < c, !p. c < p ==> b < p, !a'. a' < a ==> a' < b}
    ?- SUC a <= c
```

which can be avoided by first 'freezing' the theorem, using the tactic

```
FREEZE_THEN IMP_RES_TAC th
```

This prevents the variables `a` and `b` from being instantiated.

```
{b < c, a < b, a < c} ?- SUC a <= c
```

Used in serious proof hacking to limit the matches achievable by resolution and rewriting.

See also

`Thm.ASSUME`, `Tactic.IMP_RES_TAC`, `Drule.PROVE_HYP`, `Tactic.RES_TAC`, `Conv.REWR_CONV`

front_last (Lib)

```
Lib.front_last : 'a list -> 'a list * 'a
```

Takes a non-empty list `L` and returns a pair `(front,last)` such that `front @ [last] = L`.

Failure

Fails if the list is empty.

Example

```
> front_last [1];
val it = ([], 1): int list * int

> front_last [1,2,3];
val it = ([1, 2], 3): int list * int
```

See also

`Lib.butlast`, `Lib.last`

fst (Lib)

```
Lib.fst : ('a * 'b) -> 'a
```

Extracts the first component of a pair.

`fst (x,y)` returns `x`.

Failure

Never fails. However, notice that `fst (x,y,z)` fails to typecheck, since `(x,y,z)` is not a pair.

Example

```
> fst (1, "foo");
val it = 1: int

> fst (1, "foo", []);
Exception- Type error in function application.
   Function: fst : 'a * 'b -> 'a
   Argument: (1, "foo", []) : int * string * 'a list
   Reason:
      Can't unify 'a * 'b to int * string * 'a list
         (Different number of fields)
Fail "Static Errors" raised

> fst (1, ("foo", []));
val it = 1: int
```

See also

`Lib.snd`

ftyvar (Type)

`Type.ftyvar : hol_type`

Common type variable.

The ML variable `Type.ftyvar` is bound to the type variable `'f`.

See also

`Type.alpha`, `Type.beta`, `Type.gamma`, `Type.delta`, `Type.etyvar`, `Type.bool`

FULL_GEN_TYVARIFY (Drule)

```
Drule.FULL_GEN_TYVARIFY : thm -> thm
```

Replace a theorem's type variables with fresh versions

A call to `FULL_GEN_TYVARIFTY th` replaces (with `INST_TYPE`) the type variables occurring in theorem `th`. The new type variables are generated by successive calls to `gen_tyvar`, so should not have been seen before.

Failure

Never fails.

Comments

The derived rule `GEN_TYVARIFY` will only instantiate those type variables that are exclusively found in the conclusion. This is reasonable when handling theorems derived from a tactic goal's assumptions.

See also

`Drule.GEN_TYVARIFY`, `Thm.INST_TYPE`

FULL_SIMP_TAC (bossLib)

```
bossLib.FULL_SIMP_TAC : simpset -> thm list -> tactic
```

Also exported as `simpLib.FULL_SIMP_TAC`.

Simplifies the goal (assumptions as well as conclusion) with the given simpset.

`FULL_SIMP_TAC` is a powerful simplification tactic that simplifies all of a goal. It proceeds by applying simplification to each assumption of the goal in turn, accumulating simplified assumptions as it goes. These simplified assumptions are used to simplify further

assumptions, and all of the simplified assumptions are used as additional rewrites when the conclusion of the goal is simplified.

In addition, simplified assumptions are added back onto the goal using the equivalent of STRIP_ASSUME_TAC and this causes automatic skolemization of existential assumptions, case splits on disjunctions, and the separate assumption of conjunctions. If an assumption is simplified to TRUTH, then this is left on the assumption list. If an assumption is simplified to falsity, this proves the goal.

Failure

FULL_SIMP_TAC never fails, but it may diverge.

Example

Here FULL_SIMP_TAC is used to prove a goal:

```
> FULL_SIMP_TAC arith_ss [] (map Term [`x = 3`, `x < 2`],
                             Term `?y. x * y = 51`)
- val it = ([], fn) : tactic_result
```

Using LESS_OR_EQ |- !m n. m <= n = m < n \/ (m = n), a useful case split can be induced in the next goal:

```
> FULL_SIMP_TAC bool_ss [LESS_OR_EQ] (map Term [`x <= y`, `x < z`],
                                       Term `x + y < z`);
- val it =
    ([([`x < y`, `x < z`], `x + y < z`),
      ([`x = y`, `x < z`], `y + y < z`)], fn)
    : tactic_result
```

Note that the equality x = y is not used to simplify the subsequent assumptions, but is used to simplify the conclusion of the goal.

Comments

The application of STRIP_ASSUME_TAC to simplified assumptions means that FULL_SIMP_TAC can cause unwanted case-splits and other undesirable transformations to occur in one's assumption list. If one wants to apply the simplifier to assumptions without this occurring, the best approach seems to be the use of RULE_ASSUM_TAC and SIMP_RULE.

Each assumption is used to rewrite lower-numbered assumptions. To get the opposite effect, where each assumption is used to rewrite higher-numbered assumptions, use `REV_FULL_SIMP_TAC`.

See also

`bossLib.REV_FULL_SIMP_TAC`, `bossLib.ASM_SIMP_TAC`, `bossLib.SIMP_CONV`, `bossLib.SIMP_RULE`, `bossLib.SIMP_TAC`, `BasicProvers.VAR_EQ_TAC`

FULL_SIMP_TAC (simpLib)

```
simpLib.FULL_SIMP_TAC : simpset -> thm list -> tactic
```

Re-exported from `bossLib.FULL_SIMP_TAC`. See that entry for full documentation.

FULL_STRUCT_CASES_TAC (Tactic)

```
Tactic.FULL_STRUCT_CASES_TAC : thm_tactic
```

A form of `STRUCT_CASES_TAC` that also applies the case analysis to the assumption list.

See `STRUCT_CASES_TAC`.

Failure

Fails unless provided with a theorem that is a conjunction of (possibly multiply existentially quantified) terms which assert the equality of a variable with some given terms.

Example

Suppose we have the goal:

```
~(l:(*)list = []) ?- (LENGTH l) > 0
```

then we can get rid of the universal quantifier from the inbuilt list theorem `list_CASES`:

```
list_CASES = !l. (l = []) \/ (?t h. l = CONS h t)
```

and then use `FULL_STRUCT_CASES_TAC`. This amounts to applying the following tactic:

```
FULL_STRUCT_CASES_TAC (SPEC_ALL list_CASES)
```

which results in the following two subgoals:

```
~(CONS h t = []) ?- (LENGTH(CONS h t)) > 0
```

```
~([] = []) ?- (LENGTH[]) > 0
```

Note that this is a rather simple case, since there are no constraints, and therefore the resulting subgoals have no extra assumptions.

Generating a case split from the axioms specifying a structure.

See also

`Tactic.ASM_CASES_TAC`, `Tactic.BOOL_CASES_TAC`, `Tactic.COND_CASES_TAC`, `Tactic.DISJ_-CASES_TAC`, `Tactic.STRUCT_CASES_TAC`

FUN_EQ_CONV (Conv)

```
Conv.FUN_EQ_CONV : conv
```

Equates normal and extensional equality for two functions.

The conversion `FUN_EQ_CONV` embodies the fact that two functions are equal precisely when they give the same results for all values to which they can be applied. When supplied with a term argument of the form `f = g`, where `f` and `g` are functions of type `ty1->ty2`, `FUN_EQ_CONV` returns the theorem:

```
|- (f = g) = (!x. f x = g x)
```

where `x` is a variable of type `ty1` chosen by the conversion.

Failure

`FUN_EQ_CONV tm` fails if `tm` is not an equation `f = g`, where `f` and `g` are functions.

Used for proving equality of functions.

See also

`Drule.EXT`, `Conv.X_FUN_EQ_CONV`

funpow (Lib)

```
Lib.funpow : int -> ('a -> 'a) -> 'a -> 'a
```

Iterates a function a fixed number of times.

`funpow n f x` applies `f` to `x`, `n` times, giving the result `f (f ... (f x)...)` where the number of `f`'s is `n`. If `n` is not positive, the result is `x`.

Failure

`funpow n f x` fails if any of the `n` applications of `f` fail.

Example

Apply `tl` three times to a list:

```
- funpow 3 tl [1,2,3,4,5];
> [4, 5] : int list
```

Apply `tl` zero times:

```
- funpow 0 tl [1,2,3,4,5];
> [1; 2; 3; 4; 5] : int list
```

Apply `tl` six times to a list of only five elements:

```
- funpow 6 tl [1,2,3,4,5];
! Uncaught exception:
! List.Empty
```

See also

`Lib.repeat`

fv_term (boolLib)

```
boolLib.fv_term : (term -> tactic) -> tactic
```

Applies a term-tactic to a goal's "first" free variable

Applying `fv_term tmtac` to a goal `A ?- g`, finds the first free variable in the goal, and passes that variable to the function `tmtac`, generating a tactic, which is then applied to the goal. The first free variable is the first returned by successive calls to `free_vars` applied to first `g` and then each assumption in `A` in turn.

Failure

Fails if a goal does not have any free variables, or if `tmtac v` fails when applied to the goal, with `v` the "first" free variable as defined above.

Example

```
          ?- 0 < f (n:num)
================================== fv_term (C tmCases_on ["", "j"])
  ?- 0 < f 0    ?- 0 < f (SUC j)
```

See also

`boolLib.first_fv_term`

FVL (Term)

```
Term.FVL : term list -> term set -> term set
```

Efficient computation of the set of free variables in a list of terms.

An invocation `FVL [t1,...,tn] V` adds the set of free variables found in `t1,...,tn` to the accumulator `V`.

Failure

Never fails.

Example

```
> FVL [Term `v1 /\ v2 ==> v2 \/ v3`] empty_varset;
val it = HOLset{“v1”, “v2”, “v3”}: term set

> HOLset.listItems it;
val it = [“v1”, “v2”, “v3”]: term list
```

Comments

Preferable to `free_varsl` when the number of free variables becomes large.

See also

`HOLset`, `Term.empty_varset`, `Term.free_varsl`, `Term.free_vars`

g (proofManagerLib)

```
proofManagerLib.g : term quotation -> proofs
```

Initializes the subgoal package with a new goal which has no assumptions.

The call

```
g `tm`
```

is equivalent to

```
set_goal([],Term`tm`)
```

and clearly more convenient if a goal has no assumptions. For a description of the subgoal package, see `set_goal`.

Failure

Fails unless the argument term has type `bool`.

Example

```
> g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;
val it =
   Proof manager status: 3 proofs.
   3. Completed goalstack:  [.] ⊢ q

   2. Completed goalstack: ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

   1. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]
```

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.s backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

gamma (Type)

```
Type.gamma : hol_type
```

Common type variable.

The ML variable `Type.gamma` is bound to the type variable `'c`.

See also

`Type.alpha`, `Type.beta`, `Type.delta`, `Type.bool`

GE_CONV (reduceLib)

`reduceLib.GE_CONV : conv`

Proves result of less-than-or-equal-to ordering on two numerals.

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...), then `GE_CONV "m >= n"` returns the theorem:

```
|- (m >= n) = T
```

if the natural number denoted by `m` is greater than or equal to that denoted by `n`, or

```
|- (m >= n) = F
```

otherwise.

Failure

`GE_CONV tm` fails unless `tm` is of the form `"m >= n"`, where `m` and `n` are numerals.

Example

```
#GE_CONV "15 >= 14";;
|- 15 >= 14 = T

#GE_CONV "100 >= 100";;
|- 100 >= 100 = T

#GE_CONV "0 >= 107";;
|- 0 >= 107 = F
```

GEN (Thm)

```
Thm.GEN : term -> thm -> thm
```

Generalizes the conclusion of a theorem.

When applied to a term `x` and a theorem `A |- t`, the inference rule `GEN` returns the theorem `A |- !x. t`, provided `x` is a variable not free in any of the assumptions. There is no compulsion that `x` should be free in `t`.

```
   A |- t
------------  GEN x                [where x is not free in A]
 A |- !x. t
```

Failure

Fails if `x` is not a variable, or if it is free in any of the assumptions.

Example

The following example shows how the above side-condition prevents the derivation of the theorem `x=T |- !x. x=T`, which is clearly invalid.

```
> show_types := true;
val it = (): unit

> val t = ASSUME “x=T”;
val t =  [.] ⊢ (x :bool) ⇔ T: thm

> try (GEN “x:bool”) t;
Exception- HOL_ERR (at Thm.GEN: variable occurs free in hypotheses) raised
```

See also

`Thm.GENL`, `Drule.GEN_ALL`, `Tactic.GEN_TAC`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`, `Tactic.SPEC_TAC`, `ConseqConv.GEN_ASSUM`, `ConseqConv.GEN_IMP`, `ConseqConv.GEN_EQ`

GEN_ABS (Thm)

```
Thm.GEN_ABS : term option -> term list -> thm -> thm
```

Rule of inference for building binder-equations.

The `GEN_ABS` function is, semantically at least, a derived rule that combines applications of the primitive rules `ABS` and `MK_COMB`. When the first argument, a term option, is the value `NONE`, the effect is an iterated application of the rule `ABS` (as per `List.foldl`. Thus,

```
                G |- x = y
------------------------------------------  GEN_ABS NONE [v1,v2,...,vn]
 G |- (\v1 v2 .. vn. x) = (\v1 v2 .. vn. y)
```

If the first argument is `SOME b` for some term `b`, this term `b` is to be a binder, usually of polymorphic type `:('a -> bool) -> bool`. Then the effect is to interleave the effect of `ABS` and a call to `AP_TERM`. For every variable `v` in the list, the following theorem transformation will occur

```
        G |- x = y
  ----------------------- ABS v
   G |- (\v. x) = (\v. y)
---------------------------- AP_TERM b'
 G |- b (\v. x) = b (\v. x)
```

where `b'` is a version of `b` that has been instantiated to match the type of the term to which it is applied (`AP_TERM` doesn't do this).

Example

```
> val th = REWRITE_CONV [] ``t /\ u /\ u``
val th = ⊢ t ∧ u ∧ u ⇔ t ∧ u: thm

> GEN_ABS (SOME ``$!``) [``t:bool``, ``u:bool``] th;
val it = ⊢ (∀t u. t ∧ u ∧ u) ⇔ ∀t u. t ∧ u: thm
```

Failure

Fails if the theorem argument is not an equality. Fails if the second argument (the list of terms) does not consist of variables. Fails if any of the variables in the list appears in the hypotheses of the theorem. Fails if the first argument is `SOME b` and the type of `b` is either not of type `:('a -> bool) -> bool`, or some `:(ty -> bool) -> bool` where all the variables have type `ty`.

Comments

Though semantically a derived rule, a HOL kernel may implement this as part of its core for reasons of efficiency.

See also

`Thm.ABS`, `Thm.AP_TERM`, `Thm.MK_COMB`

GEN_ALL (Drule)

```
Drule.GEN_ALL : thm -> thm
```

Generalizes the conclusion of a theorem over its own free variables.

When applied to a theorem `A |- t`, the inference rule `GEN_ALL` returns the theorem `A |- !x1...xn. t`, where the `xi` are all the variables, if any, which are free in `t` but not in the assumptions.

```
      A |- t
------------------  GEN_ALL
 A |- !x1...xn. t
```

Failure

Never fails.

Comments

Sometimes people write code that depends on the order of the quantification. They shouldn't.

See also

`Thm.GEN`, `Thm.GENL`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`, `Tactic.SPEC_TAC`

GEN_ALL (hol88Lib)

```
hol88Lib.GEN_ALL : thm -> thm
```

Generalizes the conclusion of a theorem over its own free variables.

When applied to a theorem `A |- t`, the inference rule `GEN_ALL` returns the theorem `A |- !x1...xn. t`, where the `xi` are all the variables, if any, which are free in `t` but not in the assumptions.

```
       A |- t
------------------  GEN_ALL
 A |- !x1...xn. t
```

Failure

Never fails.

Comments

Superseded by `Drule.GEN_ALL`, which, however, may return a different result. That is why `GEN_ALL` is in `hol88Lib`. Sometimes people write code that depends on the order of the quantification. They shouldn't.

See also

`Drule.GEN_ALL`

GEN_ALPHA_CONV (Drule)

```
Drule.GEN_ALPHA_CONV : term -> conv
```

Renames the bound variable of an abstraction, a quantified term, or other binder application.

The conversion `GEN_ALPHA_CONV` provides alpha conversion for lambda abstractions of the form `\y.t`, quantified terms of the forms `!y.t`, `?y.t` or `?!y.t`, and epsilon terms of the form `@y.t`. In general, if `B` is a binder constant, then `GEN_ALPHA_CONV` implements alpha conversion for applications of the form `B y.t`.

If `tm` is an abstraction `\y.t` or an application of a binder to an abstraction `B y.t`, where the bound variable `y` has type `ty`, and if `x` is a variable also of type `ty`, then `GEN_ALPHA_CONV x tm` returns one of the theorems:

```
|- (\y.t)  = (\x'. t[x'/y])
|- (B y.t)  = (B x'. t[x'/y])
```

depending on whether the input term is `\y.t` or `B y.t` respectively. The variable `x':ty` in the resulting theorem is a primed variant of `x` chosen so as not to be free in the term provided as the second argument to `GEN_ALPHA_CONV`.

Failure

`GEN_ALPHA_CONV x tm` fails if `x` is not a variable, or if `tm` does not have one of the forms `\y.t` or `B y.t`, where `B` is a binder. `GEN_ALPHA_CONV x tm` also fails if `tm` does have one of these forms, but types of the variables `x` and `y` differ.

See also

`Thm.ALPHA`, `Drule.ALPHA_CONV`, `boolSyntax.new_binder_definition`

GEN_ASSUM (ConseqConv)

```
ConseqConv.GEN_ASSUM : term -> thm -> thm
```

Generalizes the conclusion of a theorem and the hypotheses containing the same variable.

When applied to a term `x` and a theorem `[A1, A2] |- t`, where `x` occurs in `A1` but not in `A2`, the inference rule `GEN_ASSUM` returns the theorem `[!x. A1 x, A2 |- !x. t`. There is no compulsion that `x` should be free in `t`.

`GEN_ASSUM` is a generalisation of `GEN`. While `GEN` fails, if `x` is free in an assumption, `GEN_ASSUM` succeeds.

```
     A1, A2 |- t
---------------------    GEN_ASSUM x   [where x is free in A1, but not in A2]
(!x. A1), A2 |- !x. t
```

Failure

Fails if `x` is not a variable.

See also

`Thm.GEN`

GEN_BETA_CONV (PairedLambda)

```
PairedLambda.GEN_BETA_CONV : conv
```

Beta-reduces single or paired beta-redexes, creating a paired argument if needed.

The conversion `GEN_BETA_CONV` will perform beta-reduction of simple beta-redexes in the manner of `BETA_CONV`, or of tupled beta-redexes in the manner of `PAIRED_BETA_CONV`. Unlike the latter, it will force through a beta-reduction by introducing arbitrarily nested pair destructors if necessary. The following shows the action for one level of pairing; others are similar.

```
GEN_BETA_CONV "(\(x,y). t) p" = t[(FST p)/x, (SND p)/y]
```

Failure

GEN_BETA_CONV tm fails if tm is neither a simple nor a tupled beta-redex.

Example

The following examples show the action of GEN_BETA_CONV on tupled redexes. In the following, it acts in the same way as PAIRED_BETA_CONV:

```
- pairLib.GEN_BETA_CONV (Term `(\(x,y). x + y) (1,2)`);
val it = |- (\(x,y). x + y)(1,2) = 1 + 2 : thm
```

whereas in the following, the operand of the beta-redex is not a pair, so FST and SND are introduced:

```
- pairLib.GEN_BETA_CONV (Term `(\(x,y). x + y) numpair`);
> val it = |- (\(x,y). x + y) numpair = FST numpair + SND numpair : thm
```

The introduction of FST and SND will be done more than once as necessary:

```
- pairLib.GEN_BETA_CONV (Term `(\(w,x,y,z). w + x + y + z) (1,triple)`);
> val it =
    |- (\(w,x,y,z). w + x + y + z) (1,triple) =
       1 + FST triple + FST (SND triple) + SND (SND triple) : thm
```

See also

Thm.BETA_CONV, PairedLambda.PAIRED_BETA_CONV

GEN_EXISTS_TAC (bossLib)

```
bossLib.GEN_EXISTS_TAC : string -> Parse.term Lib.frag list -> tactic
```

Instantiate a quantifier at subposition.

GEN_EXISTS_TAC v_name i tries to instantiate a quantifier for a variable with name v_name with i. It is short for quantHeuristicsLib.QUANT_TAC [(v, i, [])]. It can be seen as a generalisation of Q.EXISTS_TAC.

See also

Tactic.EXISTS_TAC, quantHeuristicsLib.QUANT_TAC

gen_find_term (HolKernel)

```
HolKernel.gen_find_term : (term list * term -> 'a option) -> term -> 'a option
```

Finds first value in range of partial function mapped over sub-terms of a term.

If a call to `gen_find_term f t` returns `SOME v`, then that result is the first value returned by a call of function `f` to one of the sub-terms of term `t`. The function `f` is successively passed sub-terms of `t` starting with `t` itself and then proceeding in a top-down, left-to-right traversal.

The additional list of terms passed to the function `f` is the list of bound variables "governing" the sub-term in question, with the innermost bound variable first in the list.

Failure

A call to `gen_find_term f t` will fail if `f` fails when applied to any of the sub-terms of `t`.

Example

```
> gen_find_term (fn x => SOME x) ``SUC x``;
val it = SOME ([], “SUC x”): (term list * term) option

> gen_find_term
   (fn (bvs,t) => if null bvs andalso
                     (is_var t orelse numSyntax.is_numeral t)
                  then
                    Lib.total (match_term ``x:num``) t
                  else NONE)
   ``SUC z + (\y. y) 5``;
val it = SOME ([{redex = “x”, residue = “z”}], []):
   ((term, term) Term.subst * (hol_type, hol_type) Term.subst) option
```

Comments

This function is used to implement `bvk_find_term`. This function could itself be approximated by returning the last value in the list returned by `gen_find_terms`. Such an implementation would be less efficient because it would unnecessarily construct a list of all possible results. It would also be semantically different if `f` had side effects.

See also

`HolKernel.bvk_find_term`, `HolKernel.find_term`, `HolKernel.gen_find_terms`

gen_find_terms (HolKernel)

```
HolKernel.gen_find_terms : (term list * term -> 'a option) -> term -> 'a list
```

Maps a partial function over sub-terms of a term.

A call to `gen_find_terms f t` returns a list of values derived from sub-terms of term `t`. The values are those generated by the function `f` when it returns `SOME v`. The function `f` is successively passed sub-terms of `t` starting with `t` itself and then proceeding in a top-down, left-to-right traversal. The list of results returned in the reverse of this order (but the fact that the traversal is done in the described order is observable if `f` has side effects).

The additional list of terms passed to the function `f` is the list of bound variables "governing" the sub-term in question, with the innermost bound variable first in the list.

Failure

A call to `gen_find_terms f t` will fail if `f` fails when applied to any of the sub-terms of `t`.

Example

```
> gen_find_terms (fn x => SOME x) ``SUC x``;
val it = [([], “x”), ([], “SUC”), ([], “SUC x”)]: (term list * term) list
```

```
> gen_find_terms
   (fn (bvs,t) => if null bvs andalso
                     (is_var t orelse numSyntax.is_numeral t)
                  then
                    Lib.total (match_term ``x:num``) t
                  else NONE)
   ``SUC z + (\y. y) 5``;
val it =
   [([{redex = “x”, residue = “5”}], []),
    ([{redex = “x”, residue = “z”}], [])]:
   ((term, term) Term.subst * (hol_type, hol_type) Term.subst) list
```

See also

`HolKernel.bvk_find_term`, `HolKernel.find_term`, `HolKernel.gen_find_term`

GEN_IMP (ConseqConv)

```
ConseqConv.GEN_IMP : term -> thm -> thm
```

Generalizes both sides of an implication in the conclusion of a theorem.

When applied to a term `x` and a theorem `A |- t1 ==> t2`, the inference rule `GEN_IMP` returns the theorem `A |- (!x. t1) ==> (!x. t2)`, provided `x` is a variable not free in any of the assumptions. There is no compulsion that `x` should be free in `t1` or `t2`.

```
        A |- (t1 ==> t2)
   ---------------------------     GEN_IMP x       [where x is not free in A]
    A |- (!x. t1) ==> (!x. t2)
```

Failure

Fails if `x` is not a variable, the conclusion of the theorem is not an implication, or if `x` is free in any of the assumptions.

Example

```
- val thm0 = mk_thm ([], Term `P (x:'a) ==> Q x`);
> val thm0 =  |- P (x :'a) ==> Q x : thm

- val thm1 = GEN_IMP (Term `x:'a`) thm0;
> val thm1 =  |- (!x. P x) ==> (!x. Q x)
```

See also

Thm.GEN, ConseqConv.GEN_EQ

GEN_MESON_TAC (mesonLib)

```
mesonLib.GEN_MESON_TAC : int -> int -> int -> thm list -> tactic
```

Performs first order proof search to prove the goal, using both the given theorems and the assumptions in the search.

GEN_MESON_TAC is the function which provides the underlying implementation of the model elimination solver used by both MESON_TAC and ASM_MESON_TAC. The three integer parameters correspond to various ways in which the search can be tuned.

The first is the minimum depth at which to search. Setting this to a number greater than zero can save time if its clear that there will not be a proof of such a small depth. ASM_MESON_TAC and MESON_TAC always use a value of 0 for this parameter.

The second is the maximum depth to which to search. Setting this low will stop the search taking too long, but may cause the engine to miss proofs it would otherwise find. The setting of this variable for ASM_MESON_TAC and MESON_TAC is done through the reference variable mesonLib.max_depth. This is set to 30 by default, but most proofs do not need anything like this depth.

The third parameter is the increment used to increase the depth of search done by the proof search procedure.

The approach used is iterative deepening, so with a call to

```
GEN_MESON_TAC mn mx inc
```

the algorithm looks for a proof of depth `mn`, then for one of depth `mn + inc`, then at depth `mn + 2 * inc` etc. Once the depth gets greater than `mx`, the proof search stops.

Failure

`GEN_MESON_TAC` fails if it searches to a depth equal to the second integer parameter without finding a proof. Shouldn't fail otherwise.

The construction of tailored versions of `MESON_TAC` and `ASM_MESON_TAC`.

See also

`mesonLib.ASM_MESON_TAC`, `mesonLib.MESON_TAC`

gen_new_specification (Definition)

```
Definition.gen_new_specification : string * thm -> thm
```

Introduce a constant or constants satisfying a given property.

The ML function `gen_new_specification` implements the generalised primitive rule of constant specification for the HOL logic.

Evaluating:

```
gen_new_specification (name, [x1=t1,...,xn=tn] |- t)
```

simultaneously introduces new constants named `x1`,...,`xn` satisfying the property:

```
|- t
```

where the variables `x1`,...,`xn` in `t` are replaced by the new constants.

This theorem is stored, with name `name`, as a definition in the current theory segment. It is also returned by the call to `gen_new_specification`.

Failure

gen_new_specification fails if any of the hypotheses of the input theorem are not of the right form: they must be equations each with a variable on the left-hand side and no free variables on the right-hand side. It also fails if the supplied variables (equivalently, the desired constant names) x1,...,xn are not distinct. Finally, failure occurs if the type of some ti does not contain all the type variables occurring in the term ti itself.

Comments

The generalised version is described in Rob Arthan's ITP 2014 paper, HOL Constant Definition Done Right, available from http://www.lemma-one.com/papers/hcddr.pdf.

See also

Definition.new_specification

GEN_PALPHA_CONV (PairRules)

PairRules.GEN_PALPHA_CONV : term -> conv

Renames the bound pair of a paired abstraction, quantified term, or other binder.

The conversion GEN_PALPHA_CONV provides alpha conversion for lambda abstractions of the form \p.t, quantified terms of the forms !p.t, ?p.t or ?!p.t, and epsilon terms of the form @p.t.

The renaming of pairs is as described for PALPHA_CONV.

Failure

GEN_PALPHA_CONV q tm fails if q is not a variable, or if tm does not have one of the required forms. GEN_ALPHA_CONV q tm also fails if tm does have one of these forms, but types of the variables p and q differ.

See also

Drule.GEN_ALPHA_CONV, PairRules.PALPHA, PairRules.PALPHA_CONV

GEN_REWRITE_CONV (Rewrite)

```
Rewrite.GEN_REWRITE_CONV : ((conv -> conv) -> rewrites -> thm list -> conv)
```

Rewrites a term, selecting terms according to a user-specified strategy.

Rewriting in HOL is based on the use of equational theorems as left-to-right replacements on the subterms of an object theorem. This replacement is mediated by the use of `REWR_CONV`, which finds matches between left-hand sides of given equations in a term and applies the substitution.

Equations used in rewriting are obtained from the theorem lists given as arguments to the function. These are at first transformed into a form suitable for rewriting. Conjunctions are separated into individual rewrites. Theorems with conclusions of the form `"~t"` are transformed into the corresponding equations `"t = F"`. Theorems `"t"` which are not equations are cast as equations of form `"t = T"`.

If a theorem is used to rewrite a term, its assumptions are added to the assumptions of the returned theorem. The matching involved uses variable instantiation. Thus, all free variables are generalized, and terms are instantiated before substitution. Theorems may have universally quantified variables.

The theorems with which rewriting is done are divided into two groups, to facilitate implementing other rewriting tools. However, they are considered in an order-independent fashion. (That is, the ordering is an implementation detail which is not specified.)

The search strategy for finding matching subterms is the first argument to the rule. Matching and substitution may occur at any level of the term, according to the specified search strategy: the whole term, or starting from any subterm. The search strategy also specifies the depth of the search: recursively up to an arbitrary depth until no matches occur, once over the selected subterm, or any more complex scheme.

Failure

`GEN_REWRITE_CONV` fails if the search strategy fails. It may also cause a non-terminating sequence of rewrites, depending on the search strategy used.

This conversion is used in the system to implement all other rewritings conversions, and may provide a user with a method to fine-tune rewriting of terms.

Example

Suppose we have a term of the form:

```
"(1 + 2) + 3 = (3 + 1) + 2"
```

and we would like to rewrite the left-hand side with the theorem ADD_SYM without changing the right hand side. This can be done by using:

```
GEN_REWRITE_CONV (RATOR_CONV o ONCE_DEPTH_CONV) empty_rewrites [ADD_SYM] mythm
```

Other rules, such as ONCE_REWRITE_CONV, would match and substitute on both sides, which would not be the desirable result.

As another example, REWRITE_CONV could be implemented as

```
GEN_REWRITE_CONV TOP_DEPTH_CONV (implicit_rewrites())
```

which specifies that matches should be searched recursively starting from the whole term of the theorem, and implicit_rewrites must be added to the user defined set of theorems employed in rewriting.

See also

Rewrite.ONCE_REWRITE_CONV, Rewrite.PURE_REWRITE_CONV, Conv.REWR_CONV, Rewrite.REWRITE_CONV

GEN_REWRITE_RULE (Rewrite)

```
Rewrite.GEN_REWRITE_RULE : ((conv -> conv) -> rewrites -> thm list -> thm -> thm)
```

Rewrites a theorem, selecting terms according to a user-specified strategy.

Rewriting in HOL is based on the use of equational theorems as left-to-right replacements on the subterms of an object theorem. This replacement is mediated by the use of REWR_CONV, which finds matches between left-hand sides of given equations in a term and applies the substitution.

Equations used in rewriting are obtained from the theorem lists given as arguments to the function. These are at first transformed into a form suitable for rewriting. Conjunctions are separated into individual rewrites. Theorems with conclusions of the form `"~t"` are transformed into the corresponding equations `"t = F"`. Theorems `"t"` which are not equations are cast as equations of form `"t = T"`.

If a theorem is used to rewrite the object theorem, its assumptions are added to the assumptions of the returned theorem, unless they are alpha-convertible to existing assumptions. The matching involved uses variable instantiation. Thus, all free variables are generalized, and terms are instantiated before substitution. Theorems may have universally quantified variables.

The theorems with which rewriting is done are divided into two groups, to facilitate implementing other rewriting tools. However, they are considered in an order-independent fashion. (That is, the ordering is an implementation detail which is not specified.)

The search strategy for finding matching subterms is the first argument to the rule. Matching and substitution may occur at any level of the term, according to the specified search strategy: the whole term, or starting from any subterm. The search strategy also specifies the depth of the search: recursively up to an arbitrary depth until no matches occur, once over the selected subterm, or any more complex scheme.

Failure

`GEN_REWRITE_RULE` fails if the search strategy fails. It may also cause a non-terminating sequence of rewrites, depending on the search strategy used.

This rule is used in the system to implement all other rewriting rules, and may provide a user with a method to fine-tune rewriting of theorems.

Example

Suppose we have a theorem of the form:

```
thm = |- (1 + 2) + 3 = (3 + 1) + 2
```

and we would like to rewrite the left-hand side with the theorem `ADD_SYM` without changing the right hand side. This can be done by using:

```
GEN_REWRITE_RULE (RATOR_CONV o ONCE_DEPTH_CONV) empty_rewrites [ADD_SYM] mythm
```

Other rules, such as `ONCE_REWRITE_RULE`, would match and substitute on both sides, which would not be the desirable result.

As another example, REWRITE_RULE could be implemented as

```
GEN_REWRITE_RULE TOP_DEPTH_CONV (implicit_rewrites())
```

which specifies that matches should be searched recursively starting from the whole term of the theorem, and implicit_rewrites must be added to the user defined set of theorems employed in rewriting.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.FILTER_ASM_REWRITE_RULE, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Conv.REWR_CONV, Rewrite.REWRITE_RULE

GEN_REWRITE_TAC (Rewrite)

```
Rewrite.GEN_REWRITE_TAC : ((conv -> conv) -> rewrites -> thm list -> tactic)
```

Rewrites a goal, selecting terms according to a user-specified strategy.

Distinct rewriting tactics differ in the search strategies used in finding subterms on which to apply substitutions, and the built-in theorems used in rewriting. In the case of REWRITE_TAC, this is a recursive traversal starting from the body of the goal's conclusion part, while in the case of ONCE_REWRITE_TAC, for example, the search stops as soon as a term on which a substitution is possible is found. GEN_REWRITE_TAC allows a user to specify a more complex strategy for rewriting.

The basis of pattern-matching for rewriting is the notion of conversions, through the application of REWR_CONV. Conversions are rules for mapping terms with theorems equating the given terms to other semantically equivalent ones.

When attempting to rewrite subterms recursively, the use of conversions (and therefore rewrites) can be automated further by using functions which take a conversion and search for instances at which they are applicable. Examples of these functions are ONCE_DEPTH_CONV and RAND_CONV. The first argument to GEN_REWRITE_TAC is such a function, which specifies a search strategy; i.e. it specifies how subterms (on which substitutions are allowed) should be searched for.

The second and third arguments are lists of theorems used for rewriting. The order in which these are used is not specified. The theorems need not be in equational form: negated terms, say `"~ t"`, are transformed into the equivalent equational form `"t = F"`, while other non-equational theorems with conclusion of form `"t"` are cast as the corresponding equations `"t = T"`. Conjunctions are separated into the individual components, which are used as distinct rewrites.

Failure

`GEN_REWRITE_TAC` fails if the search strategy fails. It may also cause a non-terminating sequence of rewrites, depending on the search strategy used. The resulting tactic is invalid when a theorem which matches the goal (and which is thus used for rewriting it with) has a hypothesis which is not alpha-convertible to any of the assumptions of the goal. Applying such an invalid tactic may result in a proof of a theorem which does not correspond to the original goal.

Detailed control of rewriting strategy, allowing a user to specify a search strategy.

Example

Given a goal such as:

```
?- a - (b + c) = a - (c + b)
```

we may want to rewrite only one side of it with a theorem, say `ADD_SYM`. Rewriting tactics which operate recursively result in divergence; the tactic `ONCE_REWRITE_TAC [ADD_SYM]` rewrites on both sides to produce the following goal:

```
?- a - (c + b) = a - (b + c)
```

as `ADD_SYM` matches at two positions. To rewrite on only one side of the equation, the following tactic can be used:

```
GEN_REWRITE_TAC (RAND_CONV o ONCE_DEPTH_CONV) empty_rewrites [ADD_SYM]
```

which produces the desired goal:

```
?- a - (c + b) = a - (c + b)
```

As another example, one can write a tactic which will behave similarly to REWRITE_TAC but will also include ADD_CLAUSES in the set of theorems to use always:

```
val ADD_REWRITE_TAC = GEN_REWRITE_TAC TOP_DEPTH_CONV
                          (add_rewrites (implicit_rewrites ())
                                        [ADD_CLAUSES])
```

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Conv.REWR_CONV, Rewrite.REWRITE_TAC

GEN_TAC (Tactic)

```
Tactic.GEN_TAC : tactic
```

Strips the outermost universal quantifier from the conclusion of a goal.

When applied to a goal `A ?- !x. t`, the tactic GEN_TAC reduces it to `A ?- t[x'/x]` where `x'` is a variant of `x` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `x'` is just `x`.

```
   A ?- !x. t
==============  GEN_TAC
 A ?- t[x'/x]
```

Failure

Fails unless the goal's conclusion is universally quantified.

The tactic `REPEAT GEN_TAC` strips away any universal quantifiers, and is commonly used before tactics relying on the underlying term structure.

See also

Tactic.FILTER_GEN_TAC, Thm.GEN, Thm.GENL, Drule.GEN_ALL, Thm.SPEC, Drule.SPECL, Drule.SPEC_ALL, Tactic.SPEC_TAC, Tactic.STRIP_TAC, Tactic.X_GEN_TAC

gen_tyvar (Type)

```
Type.gen_tyvar : unit -> hol_type
```

Generate a fresh type variable.

An invocation `gen_tyvar()` generates a type variable `tyv` not seen in the current session. Furthermore, the concrete syntax of `tyv` is such that `tyv` is not obtainable by `mk_vartype`, or by parsing.

Failure

Never fails.

Example

```
> gen_tyvar();
val it = ":%%gen_tyvar%%30": hol_type

> try Type `:%%gen_tyvar%%1`;
Exception- HOL_ERR
  (at Parse.parse_type: on line 1, characters 11-12:
       Type parsing failure with remaining input: %%gen_tyvar%%1) raised

> try mk_vartype "%%gen_tyvar%%1";
val it = ":%%gen_tyvar%%1": hol_type
```

Comments

In general, the actual name returned by `gen_tyvar` should not be relied on.

Useful for coding some proof procedures.

See also

`Term.genvar`, `Term.variant`

gen_tyvar_sigma (boolSyntax)

```
boolSyntax.gen_tyvar_sigma : hol_type list -> (hol_type,hol_type) Lib.subst
```

Generates an instantiation mapping each type to a fresh type variable

A call to `gen_tyvar_sigma tys` generates an instantiation (a list of `{redex,residue}` pairs) mapping the types in `tys` to fresh type variables (generated in turn with `gen_tyvar`). Standard practice would be to have `tys` be a list of distinct type variables, but this is not checked.

Failure

Never fails.

Example

```
> gen_tyvar_sigma [":'c", ":'a", ":'bob"];
val it =
   [{redex = ":γ", residue = ":%%gen_tyvar%%32"},
    {redex = ":α", residue = ":%%gen_tyvar%%33"},
    {redex = ":'bob", residue = ":%%gen_tyvar%%34"}]:
   (hol_type, hol_type) Lib.subst
```

See also

`Type.gen_tyvar`, `Drule.GEN_TYVARIFY`

GEN_TYVARIFY (Drule)

```
Drule.GEN_TYVARIFY : thm -> thm
```

Replace a theorem's type variables with fresh versions

A call to `GEN_TYVARIFY th` replaces (with `INST_TYPE`) the type variables occurring in theorem `th`'s conclusion that do not also appear in any of `th`'s hypotheses. The new type variables are generated by successive calls to `gen_tyvar`, so should not have been seen before.

Failure

Never fails.

Comments

The derived rule `FULL_GEN_TYVARIFY` will instantiate all of a theorem's type variables, whether or not they appear in the hypotheses.

See also

`Drule.FULL_GEN_TYVARIFY`, `boolSyntax.gen_tyvar_sigma`, `Thm.INST_TYPE`

gen_tyvarify (boolSyntax)

```
boolSyntax.gen_tyvarify : term -> term
```

Instantiates a term with fresh type variables

A call to `gen_tyvarify tm` renames all of the type variables in term `tm` to fresh replacements (generated with `gen_tyvar`).

Failure

Never fails.

Example

```
> show_types := true;
  gen_tyvarify “h::t”;
val it = (): unit
val it = “(h :%%gen_tyvar%%35)::(t :%%gen_tyvar%%35 list)”: term
```

See also

`Type.gen_tyvar`

GEN_VALIDATE (Tactical)

```
Tactical.GEN_VALIDATE : bool -> tactic -> tactic
```

Where a tactic requires assumptions to be in the goal, add those assumptions as new subgoals.

See `VALIDATE`, which is implemented as `GEN_VALIDATE true`.

Suppose `tac` applied to the goal `(asl,g)` produces a justification that creates a theorem `A |- g'`. Then `GEN_VALIDATE false` adds new subgoals for members of `A`, regardless of whether they are present in `asl`.

Failure

Fails by design if `tac`, when applied to a goal, produces a proof which is invalid on account of proving a theorem whose conclusion differs from that of the goal.

Also fails if `tac` fails when applied to the given goal.

Example

For example, where theorem `uthr'` is `[p', q] |- r`

```
[...Lines elided...]
   4. Incomplete goalstack:
        Initial goal:
        ∃R. WF R ∧ (∀rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)) ∧
            ∀rst x ord. R (ord,FILTER ($¬ ∘ ord x) rst) (ord,x::rst)

   3. Incomplete goalstack:
        Initial goal:
        1 + 2 = 2 + 1
```

```
        Current goal:
        ∀(x,y). x + y = y + x

   2. Incomplete goalstack:
        Initial goal:
         0.  p ⇒ q
        ------------------------------------
             r

        Current goal:
        p

   1. Incomplete goalstack:
        Initial goal:
         0.  q
         1.  p
        ------------------------------------
             r


> e (VALIDATE (ACCEPT_TAC uthr')) ;
OK..
1 subgoal:
val it =

    0.  q
    1.  p
   ------------------------------------
        p'
```

but, instead of that,

```
> e (GEN_VALIDATE false (ACCEPT_TAC uthr')) ;
OK..
2 subgoals:
val it =

    0.  q
    1.  p
   ------------------------------------
```

```
    q

 0. q
 1. p
------------------------------------
    p'
```

Use `GEN_VALIDATE false` rather than `VALIDATE` when programming compound tactics if you want to know what the resulting subgoals will be, regardless of what the assumptions of the goal are.

See also

`Tactical.VALID`, `Tactical.VALIDATE`, `Tactical.ADD_SGS_TAC`

GEN_VALIDATE_LT (Tactical)

```
Tactical.GEN_VALIDATE_LT : bool -> list_tactic -> list_tactic
```

Where a list-tactic requires assumptions to be in the goal, add those assumptions as new subgoals.

See `VALIDATE_LT`, which is implemented as `GEN_VALIDATE_LT true`.

When list-tactic `ltac` is applied to a goal list `gl` it produces a new goal list `gl'` and a justification. When the justification is applied to a list `thl'` of theorems which are the new goals `gl'`, proved, it produces a list `thl` of theorems (which, for a valid list-tactic are the goals `gl`, proved).

But `GEN_VALIDATE_LT false ltac` also returns extra subgoals corresponding to the assumptions of `thl`.

See `GEN_VALIDATE` for more details.

Failure

Fails by design if `ltac`, when applied to a goal list, produces a proof which is invalid on account of proving a theorem whose conclusion differs from that of the corresponding goal.

Also fails if `ltac` fails when applied to the given goals.

Compared with `VALIDATE_LT ltac`, `GEN_VALIDATE_LT false ltac` can produces extra, unnecessary, subgoals. But since the subgoals produced are predictable, regardless of the assumptions of the goal, which may be useful when coding compound tactics.

See also

`Tactical.VALID`, `Tactical.VALID_LT`, `Tactical.VALIDATE`, `Tactical.VALIDATE_LT`, `Tactical.GEN_VALIDATE`

GENL (Thm)

```
Thm.GENL : term list -> thm -> thm
```

Generalizes zero or more variables in the conclusion of a theorem.

When applied to a term list `[x1,...,xn]` and a theorem `A |- t`, the inference rule `GENL` returns the theorem `A |- !x1...xn. t`, provided none of the variables `xi` are free in any of the assumptions. It is not necessary that any or all of the `xi` should be free in `t`.

```
       A |- t
------------------  GENL [x1,...,xn]       [where no xi is free in A]
 A |- !x1...xn. t
```

Failure

Fails unless all the terms in the list are variables, none of which are free in the assumption list.

See also

`Thm.GEN`, `Drule.GEN_ALL`, `Tactic.GEN_TAC`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`, `Tactic.SPEC_TAC`

GENLIST_CONV (listLib)

```
listLib.GENLIST_CONV : conv -> conv
```

Computes by inference the result of generating a list from a function.

For an arbitrary function `f`, numeral constant `n` and conversion to evaluate `f conv`, the result of evaluating

```
GENLIST_CONV conv “GENLIST f n”
```

is the theorem

```
|- GENLIST f x = [x0;x1...xi...x(n-1)]
```

where each `xi` is the result of evaluating `conv “f i”`

Example

Evaluating `GENLIST_CONV BETA_CONV “GENLIST (\n . n) 4”` will return the following theorem:

```
|- GENLIST (\n. n) 4 = [0; 1; 2; 3]
```

Failure

`GENLIST_CONV tm` fails if `tm` is not of the form described above, or if any call `conv “f i”` fails.

genvar (Term)

```
Term.genvar : hol_type -> term
```

Returns a variable whose name has not been used previously.

When given a type, `genvar` returns a variable of that type whose name has not been used for a variable or constant in the HOL session so far.

Failure

Never fails.

Example

The following indicates the typical stylized form of the names (this should not be relied on, of course):

```
- genvar bool;
> val it = `%%genvar%%1380` : term

- genvar (Type`:num`);
> val it = `%%genvar%%1381` : term
```

Note that one can anticipate `genvar`:

```
- mk_var("%%genvar%%1382",bool);
> val it = `%%genvar%%1382` : term

- genvar bool;
> val it = `%%genvar%%1382` : term
```

This shortcoming could be guarded against, but it doesn't seem worth it currently. It doesn't seem to affect the soundness of the implementation of HOL; at worst, a proof procedure may fail because it doesn't have a sufficiently fresh variable.

The unique variables are useful in writing derived rules, for specializing terms without having to worry about such things as free variable capture. If the names are to be visible to a typical user, the function `variant` can provide rather more meaningful names.

See also

`Drule.GSPEC`, `Term.variant`

genvars (Term)

```
Term.genvars : hol_type -> int -> term list
```

Generate a specified number of fresh variables.

An invocation `genvars ty n` will invoke `genvar n` times and return the resulting list of variables.

Failure

Never fails. If `n` is less-than-or-equal to zero, the empty list is returned.

Example

```
> genvars alpha 3;
val it =
   [“$var$(%%genvar%%1100)”, “$var$(%%genvar%%1101)”,
    “$var$(%%genvar%%1102)”]: term list
```

See also

`Term.genvar`, `Term.mk_var`

genvarstruct (pairSyntax)

```
pairSyntax.genvarstruct : hol_type -> term
```

Returns a pair structure of variables whose names have not been previously used.

When given a product type, `genvarstruct` returns a paired structure of variables whose names have not been used for variables or constants in the HOL session so far. The structure of the term returned will be identical to the structure of the argument.

Failure

Never fails.

Example

The following example illustrates the behaviour of `genvarstruct`:

```
- genvarstruct (type_of (Term `((1,2),(x:'a,x:'a))`));
> val it = `((%%genvar%%1535,%%genvar%%1536),%%genvar%%1537,%%genvar%%1538)`
    : term
```

Unique variables are useful in writing derived rules, for specializing terms without having to worry about such things as free variable capture. It is often important in such rules to keep the same structure. If not, `genvar` will be adequate. If the names are to be visible to a typical user, the function `pvariant` can provide rather more meaningful names.

See also

`Term.genvar`, `PairRules.GPSPEC`, `pairSyntax.pvariant`

goal_assum (Tactical)

```
Tactical.goal_assum : thm_tactic -> tactic
```

Makes the goal available as a (negated) assumption for a theorem-tactic.

An application of `goal_assum ttac` to the goal `(A,w0)` can be seen as a tactic that first transforms the goal into `(w1::A,F)` (as if starting a proof by contradiction, normalising ¬`w0` into an equivalent `w1`); pops the new assumption `w1` and applies the theorem-tactic `ttac` to this theorem (`w1 ⊢ w1`); and when this completes, renormalises the conclusion of the goal if it has turned into something of the form `w2 ⇒ F`.

The first normalisation phase will turn something of the form ¬`p` into `p⇒F`, and will also flip outermost existential quantifiers into universals. Thus, if the `w0` term was `∃x. P x ∧ Q (f x)`, the `w1` term will be `∀x. P x ∧ Q (f x) ⇒ F`. The second normalisation phase will undo this, so that if the effect of `ttac` is equivalent to a call of `MP_TAC th'` with `th'` a universally quantified implication into falsity, then the goal will again become an existentially quantified conjunction.

Failure

Fails if `ttac` fails when applied to the theorem `w1 ⊢ w1` and the goal `(A,F)`.

See also

Tactical.FIRST_ASSUM

GPSPEC (PairRules)

```
PairRules.GPSPEC : (thm -> thm)
```

Specializes the conclusion of a theorem with unique pairs.

When applied to a theorem `A |- !p1...pn. t`, where the number of universally quantified variables may be zero, GPSPEC returns `A |- t[g1/p1]...[gn/pn]`, where the `gi` is paired structures of the same structure as `pi` and made up of distinct variables , chosen by genvar.

```
      A |- !p1...pn. t
--------------------------  GPSPEC
 A |- t[g1/p1]...[gn/pn]
```

Failure

Never fails.

GPSPEC is useful in writing derived inference rules which need to specialize theorems while avoiding using any variables that may be present elsewhere.

See also

Drule.GSPEC, PairRules.PGEN, PairRules.PGENL, Term.genvar, PairRules.PGEN_TAC, PairRules.PSPEC, PairRules.PSPECL, PairRules.PSPEC_ALL, PairRules.PSPEC_TAC, PairRules.PSPEC_PAIR

gs (bossLib)

```
bossLib.gs : thm list -> tactic
```

Simplifies assumptions and goal conclusion until a normal form is reached.

A call to `gs ths` produces a simplification tactic that repeatedly simplifies with the theorems `ths`, the stateful simpset, the natural number arithmetic decision procedure and normalizer, and let-elimination (as done by `simp`) over both a goal's assumptions and the goal's conclusion.

Assumptions are simplified first, with assumption terms simplified in turn in a context that includes all of the other assumptions. After simplification, if an assumption has been reduced to `T` (truth), it is dropped. Otherwise, it is added back to the assumption list using `STRIP_ASSUME_TAC`. After this process of assumption simplification produces no further change (assessed using `CHANGED_TAC`), the goal's conclusion is also simplified, in a context that assumes all of the (now simplified) asssumptions.

Theorems with restrictions (`Once`, `Ntimes`) passed to the `gs` tactic will not have those restrictions refreshed as invocations of the base simplification procedure are repeated. This means that the restricted theorems will likely only be applied to the first assumption where the left-hand-sides match.

Failure

Never fails, but may loop.

Example

The theorem `SUB_CANCEL` has two preconditions:

```
> arithmeticTheory.SUB_CANCEL;
val it = ⊢ ∀p n m. n ≤ p ∧ m ≤ p ⇒ (p − n = p − m ⇔ n = m): thm
```

If those preconditions are distributed awkwardly in a goal, neither `fs` nor `rfs` (which make passes over the assumptions in a particular order) may be able to apply the rewrite. However, `gs` will make progress:

```
 x ≤ b, b - x = b - y, y ≤ b   ?- x * y < 10
============================================  gs[SUB_CANCEL]
        y ≤ b, x = y          ?- y ** 2 < 10
```

Comments

The accompanying functions gvs, gnvs and gns are similar, but tweak the behaviours slightly. The functions with v in their name eliminate equalities (the x = y in the example above, say), and the functions with n in the name do not use STRIP_ASSUME_TAC when adding assumptions back to the goal. The latter can prevent case-splits.

The rgs variant attacks the assumptions in the reverse order to gs. The latter simplifies older assumptions using newer assumptions, but rgs uses the opposite order. If, for example, the assumption list includes both 0 < n and n ≠ 0, then gs will preserve one of these and rgs will preserve the other.

See also

Tactical.CHANGED_TAC, bossLib.simp

GSPEC (Drule)

```
Drule.GSPEC : (thm -> thm)
```

Specializes the conclusion of a theorem with unique variables.

When applied to a theorem A |- !x1...xn. t, where the number of universally quantified variables may be zero, GSPEC returns A |- t[g1/x1]...[gn/xn], where the gi are distinct variable names of the appropriate type, chosen by genvar.

```
      A |- !x1...xn. t
------------------------  GSPEC
 A |- t[g1/x1]...[gn/xn]
```

Failure

Never fails.

GSPEC is useful in writing derived inference rules which need to specialize theorems while avoiding using any variables that may be present elsewhere.

See also

Thm.GEN, Thm.GENL, Term.genvar, Drule.GEN_ALL, Tactic.GEN_TAC, Thm.SPEC, Drule.SPECL, Drule.SPEC_ALL, Tactic.SPEC_TAC

GSUBST_TAC (Tactic)

```
Tactic.GSUBST_TAC : ((term * term) list -> term -> term) -> thm list -> tactic
```

Makes term substitutions in a goal using a supplied substitution function.

GSUBST_TAC is the basic substitution tactic by means of which other tactics such as SUBST_OCCS_TAC and SUBST_TAC are defined. Given a list `[(v1,w1),...,(vk,wk)]` of pairs of terms and a term `w`, a substitution function replaces occurrences of `wj` in `w` with `vj` according to a specific substitution criterion. Such a criterion may be, for example, to substitute all the occurrences or only some selected ones of each `wj` in `w`.

Given a substitution function `sfn`, `GSUBST_TAC sfn [A1|-t1=u1,...,An|-tn=un] (A,t)` replaces occurrences of `ti` in `t` with `ui` according to `sfn`.

```
            A ?- t
============================  GSUBST_TAC sfn [A1|-t1=u1,...,An|-tn=un]
 A ?- t[u1,...,un/t1,...,tn]
```

The assumptions of the theorems used to substitute with are not added to the assumptions `A` of the goal, while they are recorded in the proof. If any `Ai` is not a subset of `A` (up to alpha-conversion), then `GSUBST_TAC sfn [A1|-t1=u1,...,An|-tn=un]` results in an invalid tactic.

GSUBST_TAC automatically renames bound variables to prevent free variables in `ui` becoming bound after substitution.

Failure

`GSUBST_TAC sfn [th1,...,thn] (A,t)` fails if the conclusion of each theorem in the list is not an equation. No change is made to the goal if the occurrences to be substituted according to the substitution function `sfn` do not appear in `t`.

GSUBST_TAC is used to define substitution tactics such as SUBST_OCCS_TAC and SUBST_TAC. It may also provide the user with a tool for tailoring substitution tactics.

See also

Tactic.SUBST1_TAC, Tactic.SUBST_OCCS_TAC, Tactic.SUBST_TAC

GSYM (Conv)

```
Conv.GSYM : thm -> thm
```

Reverses the first equation(s) encountered in a top-down search.

The inference rule GSYM reverses the first equation(s) encountered in a top-down search of the conclusion of the argument theorem. An equation will be reversed iff it is not a proper subterm of another equation. If a theorem contains no equations, it will be returned unchanged.

```
    A |- ..(s1 = s2)...(t1 = t2)..
   -------------------------------  GSYM
    A |- ..(s2 = s1)...(t2 = t1)..
```

Failure

Never fails, and never loops infinitely.

Example

```
> arithmeticTheory.ADD;
val it = ⊢ (∀n. 0 + n = n) ∧ ∀m n. SUC m + n = SUC (m + n): thm

> GSYM arithmeticTheory.ADD;
val it = ⊢ (∀n. n = 0 + n) ∧ ∀m n. SUC (m + n) = SUC m + n: thm
```

See also

Drule.NOT_EQ_SYM, Thm.REFL, Thm.SYM

GT_CONV (reduceLib)

`reduceLib.GT_CONV : conv`

Proves result of greater-than ordering on two numerals.

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...) of type `:num`, then `GT_CONV "m > n"` returns the theorem:

```
|- (m > n) = T
```

if the natural number denoted by `m` is greater than that denoted by `n`, or

```
|- (m > n) = F
```

otherwise.

Failure

`GT_CONV tm` fails unless `tm` is of the form ``m > n``, where `m` and `n` are numerals of type `:num`.

Example

```
> GT_CONV ``100 > 10``;
val it = |- 100 > 10 <=> T : thm

> GT_CONV ``15 > 15``;
val it = |- 15 > 15 <=> F : thm

> GT_CONV ``11 > 27``;
val it = |- 11 > 27 = F : thm
```

See also

`reduceLib.LT_CONV`

guess_lengths (wordsLib)

```
wordsLib.guess_lengths : unit -> unit
```

Turns on word length guessing.

A call to `guess_lengths` adds a post-prcessing stage to the term parser: the function `inst_word_lengths` is used to instantiate type variables that are the return type of `word_concat` and `word_extract`.

Example

```
> show_types := true;
val it = (): unit

> val t1 = “(7 >< 5) a @@ (4 >< 0) a”;
val t1 =
   “(((((7 :num) >< (5 :num)) (a :δ word) :α word) @@
      (((4 :num) >< (0 :num)) a :β word))
       :γ word)”: term

> wordsLib.guess_lengths();
val it = (): unit

> val t2 = “(7 >< 5) a @@ (4 >< 0) a”;
val t2 =
   “(((((7 :num) >< (5 :num)) (a :δ word) :word3) @@
      (((4 :num) >< (0 :num)) a :word5))
       :word8)”: term
> type_of t2;
val it = “:word8”: hol_type
```

See also

`wordsLib.inst_word_lengths`, `wordsLib.notify_on_word_length_guess`

HALF_MK_ABS (jrhUtils)

```
jrhUtils.HALF_MK_ABS : (thm -> thm)
```

Converts a function definition to lambda-form.

When applied to a theorem `A |- !x. t1 x = t2`, whose conclusion is a universally quantified equation, `HALF_MK_ABS` returns the theorem `A |- t1 = \x. t2`.

```
 A |- !x. t1 x = t2
--------------------  HALF_MK_ABS            [where x is not free in t1]
 A |- t1 = (\x. t2)
```

Failure

Fails unless the theorem is a singly universally quantified equation whose left-hand side is a function applied to the quantified variable, or if the variable is free in that function.

See also

`Drule.ETA_CONV`, `Drule.MK_ABS`, `Thm.MK_COMB`, `Drule.MK_EXISTS`

HALF_MK_PABS (PairRules)

```
PairRules.HALF_MK_PABS : (thm -> thm)
```

Converts a function definition to lambda-form.

When applied to a theorem `A |- !p. t1 p = t2`, whose conclusion is a universally quantified equation, `HALF_MK_PABS` returns the theorem `A |- t1 = (\p. t2)`.

```
 A |- !p. t1 p = t2
--------------------  HALF_MK_PABS            [where p is not free in t1]
 A |- t1 = (\p. t2)
```

Failure

Fails unless the theorem is a singly paired universally quantified equation whose left-hand side is a function applied to the quantified pair, or if any of the variables in the quantified pair is free in that function.

See also

`jrhUtils.HALF_MK_ABS`, `PairRules.PETA_CONV`, `PairRules.MK_PABS`, `PairRules.MK_PEXISTS`

hash (Lib)

```
Lib.hash : int -> string -> int * int -> int
```

Hash function for strings.

An invocation `hash i s (j,k)` takes an integer `i` and uses that to construct a function that, given a string `s`, will produce values approximately equally distributed among the numbers less than `i`. The argument `j` gives an index in the string to start from. The `k` argument is an accumulator, which is useful when hashing a collection of strings.

Failure

Never fails.

Example

```
> hash 13 "ishkabibble" (0,0);
Exception- Type error in function application.
   Function: hash : string -> string
   Argument: 13 : int
   Reason:
      Can't unify int (*In Basis*) with string (*In Basis*)
         (Different type constructors)
Type error in function application.
```

```
   Function: hash 13 : string
   Argument: "ishkabibble" : string
   Reason: Value being applied does not have a function type
Fail "Static Errors" raised
```

Comments

For better results, the `i` parameter should be a prime.

This is probably not an industrial strength hash function.

(Lib)

```
op Lib.## : ('a -> 'b) * ('c -> 'd) -> 'a * 'c -> 'b * 'd
```

Infix combinator for applying two functions to the two projections of a pair.

An application `(f ## g) (x,y)` is equal to `(f x, g y)`.

Failure

If `f x` or `g y` fails.

Example

```
> (I ## dest_imp) (strip_forall (Term `!x y z. x /\ y ==> z /\ p`));
val it = (["x", "y", "z"], ("x ∧ y", "z ∧ p")): term list * (term * term)
```

Comments

The ## combinator can be thought of as a map operation for pairs. It is declared as a right associative infix.

See also

`Lib.pair`

HEADGOAL (Tactical)

```
Tactical.HEADGOAL : tactic -> list_tactic
```

The list-tactic which applies a tactic to the first member of a list of goals.

If `tac` is a tactic, `HEADGOAL tac` is a list-tactic which applies the tactic `tac` to the first member of a list of goals.

Failure

The application of `HEADGOAL` to a tactic never fails. The resulting list-tactic fails the goal list is empty or or finally if `tac` fails when applied to the first member of the goal list.

Applying a tactic to the first subgoal.

Example

Where `tac1` and `tac2` are tactics, `tac1 THEN_LT HEADGOAL tac2` applies `tac1` to a goal, and then applies `tac2` to the first resulting subgoal.

See also

`Tactical.THEN_LT`, `Tactical.NTH_GOAL`, `Tactical.THEN1`, `Tactical.LASTGOAL`

hidden (Parse)

```
Parse.hidden : string -> bool
```

Checks to see if a given name has been hidden.

A call `hidden c` where `c` is the name of a constant, will check to see if the given name had been hidden by a previous call to `Parse.hide`.

Failure

Never fails.

Comments

The hiding of a constant only affects the quotation parser; the constant is still there in a theory.

See also

`Parse.hide`, `Parse.reveal`

hide (Parse)

```
Parse.hide : string -> ({Name : string, Thy : string} list *
                        {Name : string, Thy : string} list)
```

Stops the quotation parser from recognizing a constant.

A call `hide c` where `c` is a string that maps to one or more constants, will prevent the quotation parser from parsing it as such; it will just be parsed as a variable. (A string maps to a set of possible constants because of the possibility of overloading.) The function returns two lists. Both specify constants by way of pairs of strings. The first list is of constants that the string might have mapped to in parsing (specifically, in the `absyn_to_term` stage of parsing), and the second is the list of constants that would have tried to be printed as the string. It is important to note that the two lists need not be the same.

The effect can be reversed by `Parse.update_overload_maps`. The function `reveal` is only the inverse of `hide` if the only constants mapped to by the string all have that string as their names. (These constants will all be in different theories.)

Failure

Never fails.

Comments

The hiding of a constant only affects the quotation parser; the constant is still there in a theory. Further, (re-)defining a string hidden with `hide` will reveal it once more. The `hide` function's effect is temporary; it is not exported with a theory. A more permanent hiding effect is possible with use of the `remove_ovl_mapping` function.

See also

`Parse.hidden`, `Parse.known_constants`, `Parse.remove_ovl_mapping`, `Parse.reveal`, `Parse.set_known_constants`, `Parse.update_overload_maps`

HINT_EXISTS_TAC (Tactic)

```
Tactic.HINT_EXISTS_TAC : tactic
```

Reduces an existentially quantified goal by finding a witness which, from the assumption list, satisfies at least partially the body of the existential.

When applied to a goal `?x. t1 /\ ... /\ tn`, the tactic `HINT_EXISTS_TAC` looks for an assumption of the form `ti[u/x]`, where `i` belongs to `1..n`, and reduces the goal by taking `u` as a witness for `x`.

Failure

Fails unless the goal contains an assumption of the expected form.

Example

- The goal:

```
b = 0, a < 1, c > 0 ?- ?x. x < 1
```

is turned by `HINT_EXISTS_TAC` into:

```
b = 0, a < 1, c > 0 ?- a < 1
```

- However the tactic also allows to make progress if only one conjunct of the existential is satisfied. For instance, the goal:

```
b = 0, a < 1, c > 0 ?- ?x. x < 1 /\ x + x = c
```

is turned by `HINT_EXISTS_TAC` into:

```
b = 0, a < 1, c > 0 ?- a < 1 /\ a + a = c
```

- The location of the conjunct does not matter, the goal:

```
b = 0, a < 1, c > 0 ?- ?x. x + x = c /\ x < 1
```

is turned by `HINT_EXISTS_TAC` into:

```
b = 0, a < 1, c > 0 ?- a + a = c /\ a < 1
```

- It can be convenient to chain the call to `HINT_EXISTS_TAC` with one to `ASM_REWRITE_TAC` in order to remove automatically the satisfied conjunct:

```
b = 0, a < 1, c > 0 ?- ?x. x + x = c /\ x < 1
```

is turned by `HINT_EXISTS_TAC THEN ASM_REWRITE_TAC[]` into:

```
b = 0, a < 1, c > 0 ?- a + a = c
```

Avoid providing a witness explicitly, in order to make the tactic script less fragile.

See also

`Tactic.EXISTS_TAC`

HO_MATCH_ABBREV_TAC (Q)

```
Q.HO_MATCH_ABBREV_TAC : term quotation -> tactic
```

Introduces abbreviations by doing a higher-order match against the goal.

This tactic is just like `Q.MATCH_ABBREV_TAC`, but does a higher-order match against the goal rather than a first order match. See the documentation for `MATCH_ABBREV_TAC` for more details.

Example

The goal

```
?- !n. (n + 1) * (n - 1) = n * n - 1
```

is transformed by `Q.HO_MATCH_ABBREV_TAC `!k. P k`` to

```
Abbrev(P = (\n. (n + 1) * (n - 1) = n * n - 1)) ?-  !k. P k
```

Note how the bound variable changes to match that used in the pattern.

See also

`Q.ABBREV_TAC`, `Q.MATCH_ABBREV_TAC`

HO_MATCH_MP_TAC (Tactic)

```
Tactic.HO_MATCH_MP_TAC : thm_tactic
```

Reduces the goal using a supplied implication, with higher-order matching.

When applied to a theorem of the form

```
A' |- !x1...xn. s ==> t
```

`HO_MATCH_MP_TAC` produces a tactic that reduces a goal whose conclusion `t'` is a substitution and/or type instance of `t` to the corresponding instance of `s`. Any variables free in `s` but not in `t` will be existentially quantified in the resulting subgoal:

```
    A ?- t'
======================  HO_MATCH_MP_TAC (A' |- !x1...xn. s ==> t)
    A ?- ?z1...zp. s'
```

where `z1`, ..., `zp` are (type instances of) those variables among `x1`, ..., `xn` that do not occur free in `t`. Note that this is not a valid tactic unless `A'` is a subset of `A`.

Example

The following goal might be solved by case analysis:

```
> g `!n:num. n <= n * n`;
```

We can "manually" perform induction by using the following theorem:

```
> numTheory.INDUCTION;
- val it : thm = |- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> (!n. P n)
```

which is useful with `HO_MATCH_MP_TAC` because of higher-order matching:

```
> e(HO_MATCH_MP_TAC numTheory.INDUCTION);
- val it : goalstack = 1 subgoal (1 total)

`0 <= 0 * 0 /\ (!n. n <= n * n ==> SUC n <= SUC n * SUC n)`
```

The goal can be finished with `SIMP_TAC arith_ss []`.

Failure

Fails unless the theorem is an (optionally universally quantified) implication whose consequent can be instantiated to match the goal.

See also

`Tactic.MATCH_MP_TAC`, `bossLib.Induct_on`, `Thm.EQ_MP`, `Drule.MATCH_MP`, `Thm.MP`, `Tactic.MP_TAC`, `ConseqConv.CONSEQ_CONV_TAC`

Hol_datatype (bossLib)

```
bossLib.Hol_datatype : hol_type quotation -> unit
```

Define a concrete datatype (deprecated syntax).

The `Hol_datatype` function provides exactly the same definitional power as the `Datatype` function (which see), with a slightly different input syntax, given below:

```
spec    ::= [ <binding> ; ]* <binding>

binding ::= <ident> = [ <clause> | ]* <clause>
          |  <ident> = <| [ <ident> : <type> ; ]* <ident> : <type> |>

clause  ::= <ident>
          |  <ident> of [<type> => ]* <type>
```

Example

For example, what with `Datatype` would be

```
Datatype`btree = Leaf 'a | Node btree 'b btree
```

is

```
Hol_datatype `btree = Leaf of 'a
                    | Node of btree => 'b => btree`
```

when using `Hol_datatype`.

The => notation in the description highlights the fact that, in HOL, constructors are by default curried.

Comments

The `Datatype` function's syntax is easier to write and easier to understand.

See also

`bossLib.Datatype`

Hol_defn (Defn)

```
Defn.Hol_defn : string -> term quotation -> defn
```

Re-exported from `bossLib.Hol_defn`. See that entry for full documentation.

Hol_defn (bossLib)

```
bossLib.Hol_defn : string -> term quotation -> defn
```

Also exported as `Defn.Hol_defn`.

General-purpose function definition facility.

`Hol_defn` allows one to define functions, recursive functions in particular, while deferring termination issues. `Hol_defn` should be used when `Define` or `xDefine` fails, or when the context required by `Define` or `xDefine` is too much.

`Hol_defn` takes the same arguments as `xDefine`.

`Hol_defn s q` automatically constructs termination constraints for the function specified by `q`, defines the function, derives the specified equations, and proves an induction theorem. All these results are packaged up in the returned `defn` value. The `defn` type is best thought of as an intermediate step in the process of deriving the unconstrained equations and induction theorem for the function.

The termination conditions constructed by `Hol_defn` are for a function that takes a single tuple as an argument. This is an artifact of the way that recursive functions are modelled.

A prettyprinter, which prints out a summary of the known information on the results of `Hol_defn`, has been installed in the interactive system.

`Hol_defn` may be found in `bossLib` and also in `Defn`.

Failure

`Hol_defn s q` fails if `s` is not an alphanumeric identifier.

`Hol_defn s q` fails if `q` fails to parse or typecheck.

`Hol_defn` may extract unsatisfiable termination conditions when asked to define a higher-order recursion involving a higher-order function that the termination condition extraction mechanism of `Hol_defn` is unaware of.

Example

Here we attempt to define a quick-sort function `qsort`:

```
- Hol_defn "qsort"
```

```
        `(qsort ___ [] = []) /\
         (qsort ord (x::rst) =
            APPEND (qsort ord (FILTER ($~ o ord x) rst))
              (x :: qsort ord (FILTER (ord x) rst)))`;

<<HOL message: inventing new type variable names: 'a>>
> val it =
    HOL function definition (recursive)

    Equation(s) :
     [...]
    |- (qsort v0 [] = []) /\
       (qsort ord (x::rst) =
        APPEND (qsort ord (FILTER ($~ o ord x) rst))
          (x::qsort ord (FILTER (ord x) rst)))

    Induction :
     [...]
    |- !P.
         (!v0. P v0 []) /\
         (!ord x rst.
            P ord (FILTER ($~ o ord x) rst) /\
            P ord (FILTER (ord x) rst) ==> P ord (x::rst))
           ==> !v v1. P v v1

    Termination conditions :
      0. WF R
      1. !rst x ord. R (ord,FILTER ($~ o ord x) rst) (ord,x::rst)
      2. !rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)
```

In the following we give an example of how to use `Hol_defn` to define a nested recursion. In processing this definition, an auxiliary function `N_aux` is defined. The termination conditions of `N` are phrased in terms of `N_aux` for technical reasons.

```
- Hol_defn "ninety1"
    `N x = if x>100 then x-10
                    else N(N(x+11))`;

> val it =
    HOL function definition (nested recursion)
```

```
Equation(s) :
 [...] |- N x = (if x > 100 then x - 10 else N (N (x + 11)))

Induction :
 [...]
|- !P.
     (!x. (~(x > 100) ==> P (x + 11)) /\
          (~(x > 100) ==> P (N (x + 11))) ==> P x)
     ==>
      !v. P v

Termination conditions :
  0. WF R
  1. !x. ~(x > 100) ==> R (x + 11) x
  2. !x. ~(x > 100) ==> R (N_aux R (x + 11)) x
```

Comments

An invocation of `Hol_defn` is usually the first step in a multi-step process that ends with unconstrained recursion equations for a function, along with an induction theorem. `Hol_defn` is used to construct the function and synthesize its termination conditions; next, one invokes `tgoal` to set up a goal to prove termination of the function. The termination proof usually starts with an invocation of `WF_REL_TAC`. After the proof is over, the desired recursion equations and induction theorem are available for further use.

It is occasionally important to understand, at least in part, how `Hol_defn` constructs termination constraints. In some cases, it is necessary for users to influence this process in order to have correct termination constraints extracted. The process is driven by so-called congruence theorems for particular HOL constants. For example, suppose we were interested in defining a 'destructor-style' version of the factorial function over natural numbers:

```
fact n = if n=0 then 1 else n * fact (n-1).
```

In the absence of a congruence theorem for the 'if-then-else' construct, `Hol_defn` would extract the termination constraints

```
0. WF R
1. !n. R (n - 1) n
```

which are unprovable, because the context of the recursive call has not been taken account of. This example is in fact not a problem for HOL, since the following congruence theorem is known to Hol_defn:

```
|- !b b' x x' y y'.
      (b = b') /\
      (b' ==> (x = x')) /\
      (~b' ==> (y = y')) ==>
      ((if b then x else y) = (if b' then x' else y'))
```

This theorem is interpreted by Hol_defn as an ordered sequence of instructions to follow when the termination condition extractor hits an 'if-then-else'. The theorem is read as follows:

```
When an instance `if B then X else Y` is encountered while the
extractor traverses the function definition, do the following:

  1. Go into B and extract termination conditions TCs(B) from
     any recursive calls in it. This returns a theorem
     TCs(B) |- B = B'.

  2. Assume B' and extract termination conditions from any
     recursive calls in X. This returns a theorem
     TCs(X) |- X = X'. Each element of TCs(X) will have
     the form "B' ==> M".

  3. Assume ~B' and extract termination conditions from any
     recursive calls in Y. This returns a theorem
     TCs(Y) |- Y = Y'. Each element of TCs(Y) will have
     the form "~B' ==> M".

  4. By equality reasoning with (1), (2), and (3), derive

         TCs(B) u TCs(X) u TCs(Y)
          |-
         (if B then X else Y) = (if B' then X' else Y')

  5. Replace "if B then X else Y" by "if B' then X' else Y'".
```

The accumulated termination conditions are propagated until the extraction process finishes, and appear as hypotheses in the final result. In our example, context is properly

accounted for in recursive calls under either branch of an 'if-then-else'. Thus the extracted termination conditions for `fact` are

```
0. WF R
1. !n. ~(n = 0) ==> R (n - 1) n
```

and are easy to prove.

Now we discuss congruence theorems for higher-order functions. A 'higher-order' recursion is one in which a higher-order function is used to apply the recursive function to arguments. In order for the correct termination conditions to be proved for such a recursion, congruence rules for the higher order function must be known to the termination condition extraction mechanism. Congruence rules for common higher-order functions, e.g., `MAP`, `EVERY`, and `EXISTS` for lists, are already known to the mechanism. However, at times, one must manually prove and install a congruence theorem for a higher-order function.

For example, suppose we define a higher-order function `SIGMA` for summing the results of a function in a list. We then use `SIGMA` in the definition of a function for summing the results of a function in an arbitrarily (finitely) branching tree.

```
- Define `(SIGMA f [] = 0) /\
          (SIGMA f (h::t) = f h + SIGMA f t)`;


- Hol_datatype `ltree = Node of 'a => ltree list`;
> val it = () : unit

- Defn.Hol_defn
     "ltree_sigma"     (* higher order recursion *)
     `ltree_sigma f (Node v tl) = f v + SIGMA (ltree_sigma f) tl`;

> val it =
  HOL function definition (recursive)

    Equation(s) :
     [..] |- ltree_sigma f (Node v tl)
               = f v + SIGMA (\a. ltree_sigma f a) tl

    Induction :
     [..] |- !P. (!f v tl. (!a. P f a) ==> P f (Node v tl))
```

```
                  ==> !v v1. P v v1

      Termination conditions :
        0. WF R
        1. !tl v f a. R (f,a) (f,Node v tl) : defn
```

The termination conditions for `ltree_sigma` seem to require finding a well-founded relation `R` such that the pair `(f,a)` is `R`-less than `(f, Node v tl)`. However, this is a hopeless task, since there is no relation between `a` and `Node v tl`, besides the fact that they are both `ltrees`. The termination condition extractor has not performed properly, because it didn't know a congruence rule for `SIGMA`. Such a congruence theorem is the following:

```
SIGMA_CONG =
 |- !l1 l2 f g.
      (l1=l2) /\ (!x. MEM x l2 ==> (f x = g x)) ==>
      (SIGMA f l1 = SIGMA g l2)
```

Once `Hol_defn` has been told about this theorem, via `write_congs`, the termination conditions extracted for the definition are provable, since `a` is a proper subterm of `Node v tl`.

```
- local open DefnBase
  in
  val _ = write_congs (SIGMA_CONG::read_congs())
  end;

- Defn.Hol_defn
     "ltree_sigma"
     `ltree_sigma f (Node v tl) = f v + SIGMA (ltree_sigma f) tl`;

> val it =
    HOL function definition (recursive)

    Equation(s) :  ...  (* as before *)
    Induction :    ...  (* as before *)

    Termination conditions :
      0. WF R
      1. !v f tl a. MEM a tl ==> R (f,a) (f,Node v tl)
```

One final point : for every HOL datatype defined by application of `Hol_datatype`, a congruence theorem is automatically proved for the 'case' constant for that type, and stored in the `TypeBase`. For example, the following congruence theorem for `num_case` is stored in the `TypeBase`:

```
|- !f' f b' b M' M.
     (M = M') /\
     ((M' = 0) ==> (b = b')) /\
     (!n. (M' = SUC n) ==> (f n = f' n))
    ==>
     (num_case b f M = num_case b' f' M')
```

This allows the contexts of recursive calls in branches of 'case' expressions to be tracked.

See also

`Defn.tgoal`, `Defn.tprove`, `bossLib.WF_REL_TAC`, `bossLib.Define`, `bossLib.xDefine`, `bossLib.Hol_datatype`

HOL_ERR (Feedback)

```
HOL_ERR : hol_error -> exn
```

Standard HOL exception.

`HOL_ERR` is the exception that HOL functions are expected to raise when they encounter an anomalous situation.

Example

A `HOL_ERR` value is built from a `hol_error` value, which is typically created with `mk_hol_error`.

```
> val test_exn =
    HOL_ERR (mk_hol_error "Foo" "bar"
                locn.Loc_Unknown "unexpected input")
```

```
> raise test_exn;
Exception- HOL_ERR (at Foo.Bar: unexpected input) raised
```

HOL_ERR values can also be directly constructed by mk_HOL_ERR or mk_HOL_ERRloc.

```
> val test_exn_again =
    mk_HOL_ERR "Foo" "bar" "unexpected input"

> raise test_exn_again;
Exception- HOL_ERR (at Foo.Bar: unexpected input) raised
```

Usage patterns

Constructing backtraces

Information can be added to a HOL_ERR with wrap_exn:

```
> raise wrap_exn "structA" "fnB" test_exn;
Exception-
   HOL_ERR
     (at structA.fnB:
        at Foo.bar: unexpected input) raised
```

Location information can be included with wrap_exn_loc.

A common HOL programming idiom using wrap_exn has the following pattern (assume function bar is being defined in structure Foo):

```
fun bar x y =
  let val z = ...
  in
     ...
  end
  handle e as HOL_ERR _ => raise wrap_exn "Foo" "bar" e
```

If HOL_ERR <holerr> happens to be raised inside an invocation of bar, the handler will extend the origins of holerr with Foo and bar and raise the augmented HOL_ERR.

Scrutinizing and setting the payload

The payload of a `HOL_ERR` can be accessed by pattern matching and the contents accessed by functions over `hol_error` such as `top_structure_of`, `top_function_of`, and `message_of`:

```
> val HOL_ERR holerr = test_exn

> val (s,f,m) = (top_structure_of holerr,
                 top_function_of holerr,
                 message_of holerr)
val f = "bar": string
val m = "unexpected input": string
val s = "Foo": string
```

Portions of the payload can also be set by `set_top_function` and `set_message`.

Branching on the interaction mode

The variable `Globals.interactive` is used by programs to tell whether the HOL4 system is running interactively (i.e. is in the Read-Eval-Print loop) or not (is running in batch mode under `Holmake`). In the REPL, an uncaught `HOL_ERR` propagates to the top level and gets prettyprinted. In batch mode, in contrast, uncaught exceptions are not prettyprinted and can be rendered quite messily. The function `render_exn` can be used to write code that displays `HOL_ERR` properly in either mode.

See also

`Feedback`, `Feedback.mk_hol_error`, `Feedback.mk_HOL_ERR`, `Feedback.mk_HOL_ERRloc`, `Feedback.wrap_exn`, `Feedback.top_structure_of`, `Feedback.top_function_of`, `Feedback.top_location_of`, `Feedback.message_of`, `Feedback.set_top_function`, `Feedback.set_message`, `Globals.interactive`, `Feedback.render_exn`

hol_error (Feedback_dtype)

```
type Feedback.origin =
   {origin_structure:string,
    origin_function:string,
```

```
      source_location : locn.locn}

datatype Feedback.hol_error =
   HOL_ERROR of
     {origins : origin list,
      message : string}
```

Payload of HOL_ERR.

The payload of the HOL_ERR exception, hol_error, is a single constructor datatype that groups a list of "origins" with a message. An origin provides information about where (host structure, calling function, source location) an exception has been raised. Since exceptions can be re-raised, the origin list can be used to construct a useful backtrace to help track down program errors.

Example

A hol_error can be directly constructed but it is preferable to use the higher level entrypoint mk_hol_error:

```
> val holerr =
    mk_hol_error "Struct" "Fn" locn.Loc_Unknown "<Msg>"
# val holerr = at Struct.Fn: <Msg>: hol_error
```

A builtin prettyprinter is used to display hol_error values.

Accessing hol_error components

The message and origins of a hol_error value can be obtained via message_of and origins_of. The subcomponents of the head of the origins list can be obtained with top_structure_of, top_function_of, and top_location_of

Changing hol_error components

To augment an existing hol_error with new origin information, use wrap_hol_error. However, it is usually preferable to use wrap_exn. To set the message to a new value, use set_message, and to set the function component of the head origin, use set_top_function.

Comment

For obscure reasons, `hol_error` is defined in structure `Feedback_dtype` and duplicated in `Feedback`.

See also

`Feedback`, `Feedback.mk_hol_error`, `Feedback.top_structure_of`, `Feedback.top_function_of`, `Feedback.top_location_of`, `Feedback.origins_of`, `Feedback.message_of`, `Feedback.set_message`, `Feedback.set_top_function`, `Feedback.wrap_exn`

HOL_MESG (Feedback)

```
Feedback.HOL_MESG : string -> unit
```

Prints out a message in a special format.

`HOL_MESG` prints out its argument after formatting it a bit. The formatting is controlled by the function held in `MESG_to_string`, which is `format_MESG` by default. The output stream that the message is printed on is controlled by `MESG_outstream`, and is `TextIO.stdOut` by default.

There are three kinds of informative messages that the `Feedback` structure supports: errors, warnings, and messages. Errors are signalled by the raising of an exception built from `HOL_ERR`; warnings, which are printed by `HOL_WARNING`, are less severe than errors, and lead to a warning message being printed; finally, messages have no pejorative weight at all, and may be freely employed, via `HOL_MESG`, to keep users informed in the normal course of processing.

Failure

The invocation fails if the formatting or output routines fail.

Example

```
- HOL_MESG "Ack.";
<<HOL message: Ack.>>
```

See also

Feedback, Feedback.HOL_ERR, Feedback.Raise, Feedback.HOL_WARNING, Feedback.MESG_to_string, Feedback.format_MESG, Feedback.MESG_outstream

Hol_reln (IndDefLib)

```
IndDefLib.Hol_reln : term quotation -> thm * thm * thm
```

Re-exported from bossLib.Hol_reln. See that entry for full documentation.

Hol_reln (bossLib)

```
bossLib.Hol_reln : term quotation -> (thm * thm * thm)
```

Also exported as IndDefLib.Hol_reln.

Defines inductive relations.

The Hol_reln function is used to define inductively characterised relations. It takes a term quotation as input and attempts to define the relations there specified. The input term quotation must parse to a term that conforms to the following grammar:

```
<input-format> ::= <clause> /\ <input-format> | <clause>
<clause>       ::= (!x1 .. xn. <hypothesis> ==> <conclusion>)
                |  (!x1 .. xn. <conclusion>)
<conclusion>   ::= <con> sv1 sv2 ....
<hypothesis>   ::= any term
<con>          ::= a new relation constant
```

The sv1 terms that appear after a constant name are so-called “schematic variables”. The same variables must always follow the same constant name throughout the definition. These variables and the names of the constants-to-be must not be quantified over in each <clause>. Otherwise, a <clause> must not include any free variables. (The universal

quantifiers at the head of the clause can be used to bind free variables, but it is also permissible to use existential quantification in the hypotheses. If a clause has no free variables, it is permissible to have no universal quantification.)

The `Hol_reln` function may be used to define multiple relations. These may or may not be mutually recursive. The clauses for each relation need not be contiguous.

The function returns three theorems. Each is also saved in the current theory segment. The first is a conjunction of implications that will be the same as the input term quotation. This theorem is saved under the name `<stem>_rules`, where `<stem>` is the name of the first relation defined by the function. The second is the induction principle for the relations, saved under the name `<stem>_ind`. The third is the cases theorem for the relations, saved under the name `<stem>_cases`. The cases theorem is of the form

```
(!a0 .. an.  R1 a0 .. an = <R1's first rule possibility> \/
                           <R1's second rule possibility> \/ ...)
                /\
(!a0 .. am.  R2 a0 .. am = <R2's first rule possibility> \/
                           <R2's second rule possibility> \/ ...)
                /\
...
```

Failure

The `Hol_reln` function will fail if the provided quotation does not parse to a term of the specified form. It will also fail if a clause's only free variables do not follow a relation name, or if a relation name is followed by differing schematic variables. If the definition principle can not prove that the characterisation is inductive (as would happen if a hypothesis included a negated occurence of one of the relation names), then the same theorems are returned, but with extra assumptions stating the required inductive property.

If the name of the new constants are such that they will produce invalid SML identifiers when bound in a theory file, using `export_theory` will fail, and suggest the use of `set_MLname` to fix the problem.

Example

Defining `ODD` and `EVEN`:

```
- Hol_reln`EVEN 0 /\
```

```
            (!n. ODD n ==> EVEN (n + 1)) /\
            (!n. EVEN n ==> ODD (n + 1))`;
> val it =
    (|- EVEN 0 /\ (!n. ODD n ==> EVEN (n + 1)) /\
        !n. EVEN n ==> ODD (n + 1),

     |- !EVEN' ODD'.
          EVEN' 0 /\ (!n. ODD' n ==> EVEN' (n + 1)) /\
          (!n. EVEN' n ==> ODD' (n + 1)) ==>
          (!a0. EVEN a0 ==> EVEN' a0) /\ !a1. ODD a1 ==> ODD' a1,

     |- (!a0. EVEN a0 = (a0 = 0) \/
                        ?n. (a0 = n + 1) /\ ODD n) /\
        !a1. ODD a1 = ?n. (a1 = n + 1) /\ EVEN n)

   : thm * thm * thm
```

Defining reflexive and transitive closure, using a schematic variable. This is appropriate because it is `RTC R` that has the inductive characterisation, not `RTC` itself.

```
- Hol_reln `(!x. RTC R x x) /\
            (!x z. (?y. R x y /\ RTC R y z) ==> RTC R x z)`;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    (|- !R. (!x. RTC R x x) /\
            !x z. (?y. R x y /\ RTC R y z) ==> RTC R x z,

     |- !R RTC'.
          (!x. RTC' x x) /\
          (!x z. (?y. R x y /\ RTC' y z) ==> RTC' x z) ==>
          !a0 a1. RTC R a0 a1 ==> RTC' a0 a1,

     |- !R a0 a1. RTC R a0 a1 =
                  (a1 = a0) \/ ?y. R a0 y /\ RTC R y a1)

  : thm * thm * thm
```

Comments

Being a definition principle, the `Hol_reln` function takes a quotation rather than a term. The structure `IndDefRules` provides functions for applying the results of an invocation of `Hol_reln`.

See also

`bossLib.Define`, `bossLib.Hol_datatype`, `IndDefRules`

hol_type (Type)

```
Type.eqtype hol_type
```

Type of HOL types.

The ML type `hol_type` represents the type of HOL types.

Comments

Since `hol_type` is an ML eqtype, any two `hol_types` `ty1` and `ty2` can be tested for equality by `ty1 = ty2`.

See also

`Term.term`

HOL_WARNING (Feedback)

```
Feedback.HOL_WARNING : string -> string -> string -> unit
```

Prints out a message in a special format.

There are three kinds of informative messages that the Feedback structure supports: errors, warnings, and messages. Errors are signalled by the raising of an exception built from HOL_ERR; warnings, which are printed by HOL_WARNING, are less severe than errors, and lead only to a formatted message being printed; finally, messages have no pejorative weight at all, and may be freely employed, via HOL_MESG, to keep users informed in the normal course of processing.

HOL_WARNING prints out its arguments after formatting them. The formatting is controlled by the function held in WARNING_to_string, which is format_WARNING by default. The output stream that the message is printed on is controlled by WARNING_outstream, and is TextIO.stdOut by default.

A call HOL_WARNING s1 s2 s3 is formatted with the assumption that s1 and s2 are the names of the module and function, respectively, from which the warning is emitted. The string s3 is the actual warning message.

Failure

The invocation fails if the formatting or output routines fail.

Example

```
> HOL_WARNING "Module" "function" "stern message.";
<<HOL warning: Module.function: stern message.>>
val it = (): unit
```

See also

Feedback, Feedback.HOL_ERR, Feedback.Raise, Feedback.HOL_MESG, Feedback.WARNING_-to_string, Feedback.format_WARNING, Feedback.WARNING_outstream

hyp (Thm)

```
Thm.hyp : thm -> term list
```

Returns the hypotheses of a theorem.

When applied to a theorem A |- t, the function hyp returns A, the list of hypotheses of the theorem.

Failure

Never fails.

Comments

The order in which hypotheses are returned can not be relied on.

See also

`Thm.dest_thm`, `Thm.concl`

HYP_CONV_RULE (Conv)

```
Conv.HYP_CONV_RULE : (term -> bool) -> (conv -> thm -> thm)
```

Makes an inference rule by applying a conversion to hypotheses of a theorem.

If `conv` is a conversion, then `HYP_CONV_RULE sel conv` is an inference rule that applies `conv` to those hypotheses of a theorem which are selected by `sel`. That is, if `conv` maps a term `"h"` to the theorem `|- h = h'`, then the rule `HYP_CONV_RULE sel conv` infers `A, h' |- c` from the theorem `A, h |- c`. More precisely, if `conv "h"` returns `A' |- h = h'`, then:

```
      A, h |- c
---------------  HYP_CONV_RULE sel conv
 A u A', h' |- c
```

Note that if the conversion `conv` returns a theorem with assumptions, then the resulting inference rule adds these to the assumptions of the theorem it returns.

Failure

`HYP_CONV_RULE sel conv th` fails if `sel` fails when applied to a hypothesis of `th`, or if `conv` fails when applied to a hypothesis selected by `sel`. The function returned by `HYP_CONV_RULE sel conv` will also fail if the ML function `conv:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `h` to a theorem `|- h = h'`).

See also

Conv.CONV_RULE, Tactic.CONV_TAC, Conv.RIGHT_CONV_RULE

I (Lib)

```
Lib.I : 'a -> 'a
```

Performs identity operation: `I x` = `x`.

Failure

Never fails.

See also

Lib, Lib.##, Lib.C, Lib.K, Lib.S, Lib.W

IF (Tactical)

```
Tactical.IF : tactic -> tactic -> tactic -> tactic
```

Implements an if-then-else for tactics, using exceptions as failure.

Applying the tactic `IF gt tt et` to a goal `g` first applies `gt` to `g`. If this tactic application succeeds (does not throw an exception), then the tactic `tt` is applied to all of the generated sub-goals. (If there are none because `gt` has completely proved the goal, this has no effect and the result is a proved goal.) If `gt g` raises an exception, then `et` is applied to `g`.

Failure

The application of `IF` to three tactic arguments never fails. The resulting tactic will fail on a goal `g` if `gt g` succeeds and `tt` fails on one of the resulting subgoals, or if `gt g` and `et g` both fail.

Example

```
> IF CONJ_TAC CONJ_TAC DISCH_TAC ([], “(p1 ∧ p2) ∧ (q1 ∧ q2)”);
val it = ([([], “p1”), ([], “p2”), ([], “q1”), ([], “q2”)], fn):
   goal list * validation

> IF CONJ_TAC CONJ_TAC DISCH_TAC ([], “p ⇒ q”);
val it = ([([“p”], “q”)], fn): goal list * validation

> IF CONJ_TAC CONJ_TAC DISCH_TAC ([], ``(p1 ∧ p2) ∧ q``);
Exception- HOL_ERR at Tactic.CONJ_TAC: raised

> IF CONJ_TAC CONJ_TAC DISCH_TAC ([], “p ∨ q”);
Exception- HOL_ERR at Tactic.DISCH_TAC: raised
```

Comments

A call to `IF gt tt et` does not behave the same as `(gt THEN tt) ORELSE et` because the latter catches possible errors in the application of `tt` to the goals generated by `gt`.

See also

`Tactical.ORELSE`, `Tactical.THEN`

IFC (Conv)

```
Conv.IFC : conv -> conv -> conv -> conv
```

Apply a conversion and branch to next conversion.

A call to `IFC c1 c2 c3 t` applies the conversion `c1` to `t`. If this application succeeds (or raises `UNCHANGED`) then `c2` is applied next. Otherwise, `c3` is applied to `t`.

Failure

Fails when `c1` succeeds and `c2` fails, or when `c1` fails and `c3` fails.

Example

```
> IFC (RATOR_CONV BETA_CONV) BETA_CONV NO_CONV ``(\x y. x ==> y) T F``;
val it = |- T ==> F : thm
```

The first RATOR_CONV BETA_CONV succeeds and chains successfully with the second BETA_CONV.

```
> IFC ALL_CONV BETA_CONV (RATOR_CONV BETA_CONV) ``(\x y. x ==> y) T F``;
Exception - BETA_CONV: not a beta redex
```

Although ALL_CONV succeeds, it does nothing, so BETA_CONV is not applicable.

```
> IFC NO_CONV BETA_CONV (RATOR_CONV BETA_CONV) ``(\x y. x ==> y) T F``;
val it = |- (\y. T ==> y) F : thm
```

The NO_CONV fails, so RATOR_CONV BETA_CONV applies.

See also

Conv.ORELSEC, Conv.REPEATC

iffLR (Drule)

```
Drule.iffLR : thm -> thm
```

Returns the left-to-right direction of a "guarded" iff theorem

A call to iffLR th, where th has the form

```
A |- !x1 .. xn. p1 /\ .. /\ pm ==> !y... q1 /\ .. ==> ... ==> (l <=> r)
```

returns the left-to-right implication

```
A |- !x1 .. xn. p1 /\ .. /\ pm ==> !y... q1 /\ .. ==> ... ==> l ==> r
```

The universal variables and various antecedents are said to "guard" the if-and-only-if conclusion l <=> r in this situation. They may be nested abitrarily deep, or not present at all. They are restored after a call to EQ_IMP_RULE is made.

Failure

Fails if the theorem is not of the form specified above.

See also

`Drule.EQ_IMP_RULE`, `Drule.iffRL`, `Drule.underAIs`

iffRL (Drule)

```
Drule.iffRL : thm -> thm
```

Returns the right-to-left direction of a "guarded" iff theorem

A call to `iffRL th`, where `th` has the form

```
A |- !x1 .. xn. p1 /\ .. /\ pm ==> !y... q1 /\ .. ==> ... ==> (l <=> r)
```

returns the right-to-left implication

```
A |- !x1 .. xn. p1 /\ .. /\ pm ==> !y... q1 /\ .. ==> ... ==> r ==> l
```

The universal variables and various antecedents are said to "guard" the if-and-only-if conclusion `l <=> r` in this situation. They may be nested abitrarily deep, or not present at all. They are restored after a call to `EQ_IMP_RULE` is made.

Failure

Fails if the theorem is not of the form specified above.

See also

`Drule.EQ_IMP_RULE`, `Drule.iffLR`, `Drule.underAIs`

IgnAsm (bossLib)

```
bossLib.IgnAsm : 'a quotation -> thm
```

Creates marker theorems causing matching assumptions to be ignored

A call to `IgnAsm q` creates a theorem that can be passed to various simplification tactics (those based on `simpLib.ASM_SIMP_TAC`) which will in turn those tactics to not use assumptions matching the provided pattern `q`. If the quotation includes the string '(* sa *)' as a suffix, the matching will be considered successful (leading to an assumption being ignored) if the pattern matches any sub-term of the assumption.

All assumptions matching the pattern will be ignored (see last example below). The matching process treats variables from the goal as constants.

Failure

Fails if the provided quotation includes any anti-quotations.

Example

In the first example below, the pattern mentions `x`, which occurs in the goal, so that this pattern does not match the assumption about variable `y`:

```
> simp[IgnAsm‘x = _’] ([“x = F”, “y = T”], “p ∧ x ∧ y”);
val it = ([([“x ⇔ F”, “y ⇔ T”], “p ∧ x”)], fn): goal list * validation

> simp[IgnAsm‘F’] ([“x = F”, “y = T”], “p ∧ x ∧ y”);
val it = ([([“x ⇔ F”, “y ⇔ T”], “F”)], fn): goal list * validation

> simp[IgnAsm‘F (* sa *)’] ([“x = F”, “y = T”], “p ∧ x ∧ y”);
val it = ([([“x ⇔ F”, “y ⇔ T”], “p ∧ x”)], fn): goal list * validation

> simp[IgnAsm‘_ = _’] ([“x = F”, “y = T”, “p:bool”], “p ∧ x ∧ y”);
val it = ([([“x ⇔ F”, “y ⇔ T”, “p”], “x ∧ y”)], fn): goal list * validation
```

See also

`bossLib.NoAsms`

IMAGE_CONV (pred_setLib)

```
pred_setLib.IMAGE_CONV : conv -> conv -> conv
```

Compute the image of a function on a finite set.

The function `IMAGE_CONV` is a parameterized conversion for computing the image of a function `f:ty1->ty2` on a finite set `{t1;...;tn}` of type `ty1->bool`. The first argument to `IMAGE_CONV` is expected to be a conversion that computes the result of applying the function `f` to each element of this set. When applied to a term `f ti`, this conversion should return a theorem of the form `|- (f ti) = ri`, where `ri` is the result of applying the function `f` to the element `ti`. This conversion is used by `IMAGE_CONV` to compute a theorem of the form

```
|- IMAGE f {t1;...;tn} = {r1;...;rn}
```

The second argument to `IMAGE_CONV` is used (optionally) to simplify the resulting image set `{r1;...;rn}` by removing redundant occurrences of values. This conversion expected to decide equality of values of the result type `ty2`; given an equation `e1 = e2`, where `e1` and `e2` are terms of type `ty2`, the conversion should return either `|- (e1 = e2) = T` or `|- (e1 = e2) = F`, as appropriate.

Given appropriate conversions `conv1` and `conv2`, the function `IMAGE_CONV` returns a conversion that maps a term of the form `IMAGE f {t1;...;tn}` to the theorem

```
|- IMAGE f {t1;...;tn} = {rj;...;rk}
```

where `conv1` proves a theorem of the form `|- (f ti) = ri` for each element `ti` of the set `{t1;...;tn}`, and where the set `{rj;...;rk}` is the smallest subset of `{r1;...;rn}` such no two elements are alpha-equivalent and `conv2` does not map `rl = rm` to the theorem `|- (rl = rm) = T` for any pair of values `rl` and `rm` in `{rj;...;rk}`. That is, `{rj;...;rk}` is the set obtained by removing multiple occurrences of values from the set `{r1;...;rn}`, where the equality conversion `conv2` (or alpha-equivalence) is used to determine which pairs of terms in `{r1;...;rn}` are equal.

Example

The following is a very simple example in which `REFL` is used to construct the result of applying the function `f` to each element of the set `{1; 2; 1; 4}`, and `NO_CONV` is the supplied 'equality conversion'.

```
- IMAGE_CONV REFL NO_CONV ``IMAGE (f:num->num) {1; 2; 1; 4}``;
> val it = |- IMAGE f {1; 2; 1; 4} = {f 2; f 1; f 4} : thm
```

The result contains only one occurrence of `f 1`, even though `NO_CONV` always fails, since `IMAGE_CONV` simplifies the resulting set by removing elements that are redundant up to alpha-equivalence.

For the next example, we construct a conversion that maps `SUC n` for any numeral `n` to the numeral standing for the successor of `n`.

```
- fun SUC_CONV tm =
   let open numLib Arbnum
       val n = dest_numeral (rand tm)
       val sucn = mk_numeral (n + one)
   in
     SYM (num_CONV sucn)
   end;

> val SUC_CONV = fn : term -> thm
```

The result is a conversion that inverts `num_CONV`:

```
- numLib.num_CONV ``4``;
> val it = |- 4 = SUC 3 : thm

- SUC_CONV ``SUC 3``;
> val it = |- SUC 3 = 4 : thm
```

The conversion `SUC_CONV` can then be used to compute the image of the successor function on a finite set:

```
- IMAGE_CONV SUC_CONV NO_CONV ``IMAGE SUC {1; 2; 1; 4}``;
> val it = |- IMAGE SUC {1; 2; 1; 4} = {3; 2; 5} : thm
```

Note that `2` (= `SUC 1`) appears only once in the resulting set.

Finally, here is an example of using `IMAGE_CONV` to compute the image of a paired addition function on a set of pairs of numbers:

```
- IMAGE_CONV (pairLib.PAIRED_BETA_CONV THENC reduceLib.ADD_CONV)
             numLib.REDUCE_CONV
         ``IMAGE (\(n,m).n+m) {{(1,2), (3,4), (0,3), (1,3)}}``;
> val it = |- IMAGE (\(n,m). n + m) {(1,2); (3,4); (0,3); (1,3)} = {7; 3; 4}
```

Failure

`IMAGE_CONV conv1 conv2` fails if applied to a term not of the form `IMAGE f {t1;...;tn}`. An application of `IMAGE_CONV conv1 conv2` to a term `IMAGE f {t1;...;tn}` fails unless for all `ti` in the set `{t1;...;tn}`, evaluating `conv1 ``f ti``` returns `|- (f ti) = ri` for some `ri`.

IMP_ANTISYM_RULE (Drule)

```
Drule.IMP_ANTISYM_RULE : thm -> thm -> thm
```

Deduces equality of boolean terms from forward and backward implications.

When applied to the theorems `A1 |- t1 ==> t2` and `A2 |- t2 ==> t1`, the inference rule `IMP_ANTISYM_RULE` returns the theorem `A1 u A2 |- t1 = t2`.

```
   A1 |- t1 ==> t2     A2 |- t2 ==> t1
  ------------------------------------  IMP_ANTISYM_RULE
           A1 u A2 |- t1 = t2
```

Failure

Fails unless the theorems supplied are a complementary implicative pair as indicated above.

See also

`Thm.EQ_IMP_RULE`, `Thm.EQ_MP`, `Tactic.EQ_TAC`

IMP_CANON (Drule)

```
Drule.IMP_CANON : (thm -> thm list)
```

Puts theorem into a 'canonical' form.

IMP_CANON puts a theorem in 'canonical' form by removing quantifiers and breaking apart conjunctions, as well as disjunctions which form the antecedent of implications. It applies the following transformation rules:

```
   A |- t1 /\ t2          A |- !x. t          A |- (t1 /\ t2) ==> t
------------------     ------------     ------------------------
 A |- t1   A |- t2         A |- t           A |- t1 ==> (t2 ==> t)

    A |- (t1 \/ t2) ==> t              A |- (?x. t1) ==> t2
------------------------------     ----------------------
 A |- t1 ==> t   A |- t2 ==> t          A |- t1[x'/x] ==> t2
```

Failure

Never fails, but if there is no scope for one of the above reductions, merely gives a list whose only member is the original theorem.

Comments

This is a rather ad-hoc inference rule, and its use is not recommended.

See also

Thm.CONJUNCT1, Thm.CONJUNCT2, Drule.CONJUNCTS, Thm.DISJ1, Thm.DISJ2, Thm.EXISTS, Thm.SPEC

IMP_CONJ (Drule)

```
Drule.IMP_CONJ : (thm -> thm -> thm)
```

Conjoins antecedents and consequents of two implications.

When applied to theorems `A1 |- p ==> r` and `A2 |- q ==> s`, the IMP_CONJ inference rule returns the theorem `A1 u A2 |- p /\ q ==> r /\ s`.

```
 A1 |- p ==> r    A2 |- q ==> s
-------------------------------  IMP_CONJ
  A1 u A2 |- p /\ q ==> r /\ s
```

Failure

Fails unless the conclusions of both theorems are implicative.

See also

`Thm.CONJ`

IMP_CONV (reduceLib)

```
reduceLib.IMP_CONV : conv
```

Simplifies certain implicational expressions.

If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `IMP_CONV tm` returns the corresponding theorem. Note that in the last case the antecedent and consequent need only be alpha-equivalent rather than strictly identical.

```
IMP_CONV “T ==> t” = |- T ==> t = t
IMP_CONV “t ==> T” = |- t ==> T = T
IMP_CONV “F ==> t” = |- F ==> t = T
IMP_CONV “t ==> F” = |- t ==> F = ~t
IMP_CONV “t ==> t” = |- t ==> t = T
```

Failure

`IMP_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
> IMP_CONV “T ==> F”;
```

```
val it =  ⊢ T ⇒ F ⇔ F : thm

> IMP_CONV “F ==> x”;
val it = ⊢ F ⇒ x ⇔ T : thm

> IMP_CONV “(!z:(num)list. z = z) ==> (!x:(num)list. x = x)”;
val it =  ⊢ (∀z. z = z) ⇒ (∀z. z = z) ⇔ T : thm
```

IMP_ELIM (Drule)

```
Drule.IMP_ELIM : (thm -> thm)
```

Transforms `|- s ==> t` into `|- ~s \/ t`.

When applied to a theorem `A |- s ==> t`, the inference rule `IMP_ELIM` returns the theorem `A |- ~s \/ t`.

```
    A |- s ==> t
   --------------  IMP_ELIM
    A |- ~s \/ t
```

Failure

Fails unless the theorem is implicative.

See also

`Thm.NOT_INTRO`, `Thm.NOT_ELIM`

IMP_RES_FORALL_CONV (res_quanLib)

```
res_quanLib.IMP_RES_FORALL_CONV : conv
```

Converts an implication to a restricted universal quantification.

When applied to a term of the form `!x. x IN P ==> Q`, the conversion `IMP_RES_FORALL_CONV` returns the theorem:

```
|- (!x. x IN P ==> Q) = !x::P. Q
```

Failure

Fails if applied to a term not of the form `!x. x IN P ==> Q`.

See also

`res_quanLib.RES_FORALL_CONV`

IMP_RES_TAC (Tactic)

```
Tactic.IMP_RES_TAC : thm_tactic
```

Enriches assumptions by repeatedly resolving an implication with them.

Given a theorem `th`, the theorem-tactic `IMP_RES_TAC` uses `RES_CANON` to derive a canonical list of implications, each of which has the form:

```
A |- u1 ==> u2 ==> ... ==> un ==> v
```

`IMP_RES_TAC` then tries to repeatedly 'resolve' these theorems against the assumptions of a goal by attempting to match the antecedents `u1`, `u2`, ..., `un` (in that order) to some assumption of the goal (i.e. to some candidate antecedents among the assumptions). If all the antecedents can be matched to assumptions of the goal, then an instance of the theorem

```
A u {a1,...,an} |- v
```

called a 'final resolvent' is obtained by repeated specialization of the variables in the implicative theorem, type instantiation, and applications of modus ponens. If only the first `i` antecedents `u1`, ..., `ui` can be matched to assumptions and then no further matching is possible, then the final resolvent is an instance of the theorem:

```
A u {a1,...,ai} |- u(i+1) ==> ... ==> v
```

All the final resolvents obtained in this way (there may be several, since an antecedent `ui` may match several assumptions) are added to the assumptions of the goal, in the stripped form produced by using `STRIP_ASSUME_TAC`. If the conclusion of any final resolvent is a contradiction '`F`' or is alpha-equivalent to the conclusion of the goal, then `IMP_RES_TAC` solves the goal.

Failure

Never fails.

See also

`Tactic.drule`, `Thm_cont.IMP_RES_THEN`, `Drule.RES_CANON`, `Tactic.RES_TAC`, `Thm_cont.RES_THEN`

IMP_RES_THEN (Thm_cont)

```
Thm_cont.IMP_RES_THEN : thm_tactical
```

Resolves an implication with the assumptions of a goal.

The function `IMP_RES_THEN` is the basic building block for resolution in HOL. This is not full higher-order, or even first-order, resolution with unification, but simply one way simultaneous pattern-matching (resulting in term and type instantiation) of the antecedent of an implicative theorem to the conclusion of another theorem (the candidate antecedent).

Given a theorem-tactic `ttac` and a theorem `th`, the theorem-tactical `IMP_RES_THEN` uses `RES_CANON` to derive a canonical list of implications from `th`, each of which has the form:

```
Ai |- !x1...xn. ui ==> vi
```

`IMP_RES_THEN` then produces a tactic that, when applied to a goal `A ?- g` attempts to match each antecedent `ui` to each assumption `aj |- aj` in the assumptions `A`. If the antecedent `ui` of any implication matches the conclusion `aj` of any assumption,

then an instance of the theorem `Ai u {aj} |- vi`, called a 'resolvent', is obtained by specialization of the variables `x1`, ..., `xn` and type instantiation, followed by an application of modus ponens. There may be more than one canonical implication and each implication is tried against every assumption of the goal, so there may be several resolvents (or, indeed, none).

Tactics are produced using the theorem-tactic `ttac` from all these resolvents (failures of `ttac` at this stage are filtered out) and these tactics are then applied in an unspecified sequence to the goal. That is,

```
IMP_RES_THEN ttac th  (A ?- g)
```

has the effect of:

```
MAP_EVERY (mapfilter ttac [... , (Ai u {aj} |- vi) , ...]) (A ?- g)
```

where the theorems `Ai u {aj} |- vi` are all the consequences that can be drawn by a (single) matching modus-ponens inference from the assumptions of the goal `A ?- g` and the implications derived from the supplied theorem `th`. The sequence in which the theorems `Ai u {aj} |- vi` are generated and the corresponding tactics applied is unspecified.

Failure

Evaluating `IMP_RES_THEN ttac th` fails if the supplied theorem `th` is not an implication, or if no implications can be derived from `th` by the transformation process described under the entry for `RES_CANON`. Evaluating `IMP_RES_THEN ttac th (A ?- g)` fails if no assumption of the goal `A ?- g` can be resolved with the implication or implications derived from `th`. Evaluation also fails if there are resolvents, but for every resolvent `Ai u {aj} |- vi` evaluating the application `ttac (Ai u {aj} |- vi)` fails—that is, if for every resolvent `ttac` fails to produce a tactic. Finally, failure is propagated if any of the tactics that are produced from the resolvents by `ttac` fails when applied in sequence to the goal.

Example

The following example shows a straightforward use of `IMP_RES_THEN` to infer an equational consequence of the assumptions of a goal, use it once as a substitution in the conclusion of goal, and then 'throw it away'. Suppose the goal is:

```
a + n = a  ?- !k. k - n = k
```

By the built-in theorem:

```
ADD_INV_0 = |- !m n. (m + n = m) ==> (n = 0)
```

the assumption of this goal implies that `n` equals `0`. A single-step resolution with this theorem followed by substitution:

```
IMP_RES_THEN SUBST1_TAC ADD_INV_0
```

can therefore be used to reduce the goal to:

```
a + n = a  ?- !k. k - 0 = m
```

Here, a single resolvent `a + n = a |- n = 0` is obtained by matching the antecedent of `ADD_INV_0` to the assumption of the goal. This is then used to substitute `0` for `n` in the conclusion of the goal.

See also

`Tactic.IMP_RES_TAC`, `Drule.MATCH_MP`, `Drule.RES_CANON`, `Tactic.RES_TAC`, `Thm_cont.RES_THEN`

IMP_TRANS (Drule)

```
Drule.IMP_TRANS : (thm -> thm -> thm)
```

Implements the transitivity of implication.

When applied to theorems `A1 |- t1 ==> t2` and `A2 |- t2 ==> t3`, the inference rule `IMP_TRANS` returns the theorem `A1 u A2 |- t1 ==> t3`.

```
 A1 |- t1 ==> t2   A2 |- t2 ==> t3
---------------------------------  IMP_TRANS
      A1 u A2 |- t1 ==> t3
```

Failure

Fails unless the theorems are both implicative, with the consequent of the first being the same as the antecedent of the second (up to alpha-conversion).

See also

`Drule.IMP_ANTISYM_RULE`, `Thm.SYM`, `Thm.TRANS`

impl_keep_tac (Tactic)

```
Tactic.impl_keep_tac : tactic
```

Implements a version of implication-left sequent calculus rule as tactic

Given a goal of the form `A ?- ((p ==> q) ==> r)`, an application of `impl_tac` will produce two sub-goals: `A ?- p` and `A, p ?- (q ==> r)`. This can be useful if `p` should be dealt with in isolation, when, say, the tactics that solve `p` can't safely be applied to `q` and/or `r`.

This tactic differs from `impl_tac` in that it keeps `p` as an assumption in the second sub-goal.

Failure

Fails if the goal is not an implication with another implication as its antecdent. Note that for the purpose of this tactic, a negation `~p` is viewed as the implication `p ==> F`. This means that `impl_tac` will succeed when applied to goals whose conclusions are `~p ==> q`, `~(p ==> q)` and `~~p`.

See also

`Tactic.impl_tac`, `Thm_cont.PROVEHYP_THEN`

impl_tac (Tactic)

```
Tactic.impl_tac : tactic
```

Implements analogue of implication-left sequent calculus rule as tactic

Given a goal of the form `A ?- ((p ==> q) ==> r)`, an application of `impl_tac` will produce two sub-goals: `A ?- p` and `A ?- (q ==> r)`. This can be useful if `p` should be dealt with in isolation, when, say, the tactics that solve `p` can't safely be applied to `q` and/or `r`.

Failure

Fails if the goal is not an implication with another implication as its antecdent. Note that for the purpose of this tactic, a negation `~p` is viewed as the implication `p ==> F`. This means that `impl_tac` will succeed when applied to goals whose conclusions are `~p ==> q`, `~(p ==> q)` and `~~p`.

See also

`Tactic.impl_keep_tac`, `Thm_cont.PROVEHYP_THEN`

implication (boolSyntax)

`boolSyntax.implication : term`

Constant denoting logical implication.

The ML variable `boolSyntax.implication` is bound to the term `min$==>`.

See also

`boolSyntax.equality`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`
`boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunc`
`boolSyntax.negation`, `boolSyntax.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_-`
`tm`, `boolSyntax.arb`

implicit_rewrites (Rewrite)

```
Rewrite.implicit_rewrites: unit -> rewrites
```

Contains a number of theorems used, by default, in rewriting.

The variable `implicit_rewrites` holds a collection of rewrite rules commonly used to simplify expressions. These rules include the clause for reflexivity:

```
|- !x. (x = x) = T
```

as well as rules to reason about equality:

```
|- !t.
   ((T = t) = t) /\ ((t = T) = t) /\ ((F = t) = ~t) /\ ((t = F) = ~t)
```

Negations are manipulated by the following clauses:

```
|- (!t. ~~t = t) /\ (~T = F) /\ (~F = T)
```

The set of tautologies includes truth tables for conjunctions, disjunctions, and implications:

```
|- !t.
    (T /\ t = t) /\
    (t /\ T = t) /\
    (F /\ t = F) /\
    (t /\ F = F) /\
    (t /\ t = t)
|- !t.
    (T \/ t = T) /\
    (t \/ T = T) /\
    (F \/ t = t) /\
    (t \/ F = t) /\
    (t \/ t = t)
|- !t.
    (T ==> t = t) /\
```

```
(t ==> T = T) /\
(F ==> t = T) /\
(t ==> t = T) /\
(t ==> F = ~t)
```

Simple rules for reasoning about conditionals are given by:

```
|- !t1 t2. ((T => t1 | t2) = t1) /\ ((F => t1 | t2) = t2)
```

Rewriting with the following tautologies allows simplification of universally and existentially quantified variables and abstractions:

```
|- !t. (!x. t) = t
|- !t. (?x. t) = t
|- !t1 t2. (\x. t1)t2 = t1
```

The value of `implicit_rewrites` can be augmented by `add_implicit_rewrites` and altered by `set_implicit_rewrites`.

The initial value of `implicit_rewrites` is `bool_rewrites`.

The rewrite rules held in `implicit_rewrites` are automatically included in the simplifications performed by some of the rewriting tools.

See also

`Rewrite.GEN_REWRITE_RULE`, `Rewrite.GEN_REWRITE_TAC`, `Rewrite.REWRITE_RULE`, `Rewrite.REWRITE_TAC`, `Rewrite.bool_rewrites`, `Rewrite.set_implicit_rewrites`, `Rewrite.add_implicit_rewrites`

IN_CONV (pred_setLib)

```
pred_setLib.IN_CONV : conv -> conv
```

Decision procedure for membership in finite sets.

The function `IN_CONV` is a parameterized conversion for proving or disproving membership assertions of the general form:

```
t IN {t1,...,tn}
```

where `{t1;...;tn}` is a set of type `ty->bool` and `t` is a value of the base type `ty`. The first argument to `IN_CONV` is expected to be a conversion that decides equality between values of the base type `ty`. Given an equation `e1 = e2`, where `e1` and `e2` are terms of type `ty`, this conversion should return the theorem `|- (e1 = e2) = T` or the theorem `|- (e1 = e2) = F`, as appropriate.

Given such a conversion, the function `IN_CONV` returns a conversion that maps a term of the form `t IN {t1;...;tn}` to the theorem

```
|- t IN {t1;...;tn} = T
```

if `t` is alpha-equivalent to any `ti`, or if the supplied conversion proves `|- (t = ti) = T` for any `ti`. If the supplied conversion proves `|- (t = ti) = F` for every `ti`, then the result is the theorem

```
|- t IN {t1;...;tn} = F
```

In all other cases, `IN_CONV` will fail.

Example

In the following example, the conversion `REDUCE_CONV` is supplied as a parameter and used to test equality of the candidate element `1` with the actual elements of the given set.

```
- IN_CONV REDUCE_CONV ``2 IN {0;SUC 1;3}``;
> val it = |- 2 IN {0; SUC 1; 3} = T : thm
```

The result is `T` because `REDUCE_CONV` is able to prove that `2` is equal to `SUC 1`. An example of a negative result is:

```
- IN_CONV REDUCE_CONV ``1 IN {0;2;3}``;
> val it = |- 1 IN {0; 2; 3} = F : thm
```

Finally the behaviour of the supplied conversion is irrelevant when the value to be tested for membership is alpha-equivalent to an actual element:

```
- IN_CONV NO_CONV ``1 IN {3;2;1}``;
> val it = |- 1 IN {3; 2; 1} = T : thm
```

The conversion `NO_CONV` always fails, but `IN_CONV` is nontheless able in this case to prove the required result.

Failure

`IN_CONV conv` fails if applied to a term that is not of the form `t IN {t1;...;tn}`. A call `IN_CONV conv t IN {t1;...;tn}` fails unless the term `t` is alpha-equivalent to some `ti`, or `conv ``t = ti``` returns `|- (t = ti) = T` for some `ti`, or `conv ``t = ti``` returns `|- (t = ti) = F` for every `ti`.

See also

`numLib.REDUCE_CONV`

ind (Type)

```
Type.ind : hol_type
```

Basic type constant.

The ML variable `Type.ind` is bound to the HOL type constant `ind`. The axiom `INFINITY_AX` in `boolTheory` states that `ind` represents an infinite set of individuals.

See also

`Type.bool`, `Type.-->`

IndDefRules (structure)

```
structure IndDefRules
```

Tom Melham's inference support for inductive definitions.

`IndDefRules` provides support for reasoning about inductively defined relations, including a general induction tactic, and an entrypoint for deriving so-called 'strong' rule induction.

index (Lib)

```
Lib.index : ('a -> bool) -> 'a list -> int
```

Finds index of first list element for which predicate holds.

An application `index P l` returns the index (0-based) to the first element (in a left-to-right scan) of `l` that `P` holds of.

Failure

If `P` doesn't hold of any element of `l`, then `index P l` fails. If `P x` fails for any `x` encountered in the scan, then `index P l` fails.

Example

```
> index (equal 3) [1,2,3];
val it = 2: int

> let fun even i = (i mod 2 = 0)
  in try (index even) [1,3,5,7,9]
  end;
Exception- Type error in function application.
   Function: try (index even) :
      (int -> int list option) list -> int -> int
   Argument: [1, 3, 5, 7, 9] : int list
   Reason:
      Can't unify int to int -> int list option (Incompatible types)
Fail "Static Errors" raised

> index (equal 3 o hd) [[1],[],[2,3]];
Exception- Empty raised
```

See also

`Lib.el`

Induct (BasicProvers)

```
BasicProvers.Induct : tactic
```

Re-exported from `bossLib.Induct`. See that entry for full documentation.

Induct (bossLib)

```
bossLib.Induct : tactic
```

Also exported as `BasicProvers.Induct`.

Performs structural induction over the type of the goal's outermost universally quantified variable.

Given a universally quantified goal, `Induct` attempts to perform an induction based on the type of the leading universally quantified variable. The induction theorem to be used is looked up in the `TypeBase` database, which holds useful facts about the system's defined types. `Induct` may also be used to reason about mutually recursive types.

Failure

`Induct` fails if the goal is not universally quantified, or if the type of the variable universally quantified does not have an induction theorem in the `TypeBase` database.

Example

If attempting to prove

```
!list. LENGTH (REVERSE list) = LENGTH list
```

one can apply `Induct` to begin a proof by induction on `list`.

```
- e Induct;
```

This results in the base and step cases of the induction as new goals.

```
?- LENGTH (REVERSE []) = LENGTH []

LENGTH (REVERSE list) = LENGTH list
?- !h. LENGTH (REVERSE (h::list)) = LENGTH (h::list)
```

The same tactic can be used for induction over numbers. For example expanding the goal

```
?- !n. n > 2 ==> !x y z. ~(x EXP n + y EXP n = z EXP n)
```

with `Induct` yields the two goals

```
?- 0 > 2 ==> !x y z. ~(x EXP 0 + y EXP 0 = z EXP 0)

n > 2 ==> !x y z. ~(x EXP n + y EXP n = z EXP n)
?- SUC n > 2 ==> !x y z. ~(x EXP SUC n + y EXP SUC n = z EXP SUC n)
```

`Induct` can also be used to perform induction on mutually recursive types. For example, given the datatype

```
Hol_datatype
    `exp = VAR of string                (* variables *)
         | IF  of bexp => exp => exp    (* conditional *)
         | APP of string => exp list    (* function application *)
      ;
    bexp = EQ  of exp => exp            (* boolean expressions *)
         | LEQ of exp => exp
         | AND of bexp => bexp
         | OR  of bexp => bexp
         | NOT of bexp`
```

one can use `Induct` to prove that all objects of type `exp` and `bexp` are of a non-zero size. (Recall that size definitions are automatically defined for datatypes.) Typically, mutually recursive types lead to mutually recursive induction schemes having multiple predicates. The scheme for the above definition has 3 predicates: `P0`, `P1`, and `P2`, which respectively range over expressions, boolean expressions, and lists of expressions.

```
|- !P0 P1 P2.
     (!a. P0 (VAR a)) /\
     (!b e e0. P1 b /\ P0 e /\ P0 e0 ==> P0 (IF b e e0)) /\
     (!l. P2 l ==> !b. P0 (APP b l)) /\
     (!e e0. P0 e /\ P0 e0 ==> P1 (EQ e e0)) /\
     (!e e0. P0 e /\ P0 e0 ==> P1 (LEQ e e0)) /\
     (!b b0. P1 b /\ P1 b0 ==> P1 (AND b b0)) /\
     (!b b0. P1 b /\ P1 b0 ==> P1 (OR b b0)) /\
     (!b. P1 b ==> P1 (NOT b)) /\
     P2 [] /\
     (!e l. P0 e /\ P2 l ==> P2 (e::l))
       ==>
     (!e. P0 e) /\ (!b. P1 b) /\ !l. P2 l
```

Invoking `Induct` on a goal such as

```
!e. 0 < exp_size e
```

yields the three subgoals

```
?- !s. 0 < exp_size (APP s l)


[ 0 < exp_size e, 0 < exp_size e' ] ?- 0 < exp_size (IF b e e')

?- !s. 0 < exp_size (VAR s)
```

In this case, `P1` and `P2` have been vacuously instantiated in the application of `Induct`, since it detects that only `P0` is needed. However, it is also possible to use `Induct` to start the proofs of

```
(!e. 0 < exp_size e) /\ (!b. 0 < bexp_size b)
```

and

```
(!e. 0 < exp_size e) /\
(!b. 0 < bexp_size b) /\
(!list. 0 < exp1_size list)
```

See also

`bossLib.Induct_on`, `bossLib.completeInduct_on`, `bossLib.measureInduct_on`, `Prim_rec.INDUCT_THEN`, `bossLib.Cases`, `bossLib.Hol_datatype`, `proofManagerLib.g`, `proofManagerLib.e`

Induct_on (BasicProvers)

```
BasicProvers.Induct_on : term quotation -> tactic
```

Re-exported from `bossLib.Induct_on`. See that entry for full documentation.

Induct_on (bossLib)

```
bossLib.Induct_on : term quotation -> tactic
```

Also exported as `BasicProvers.Induct_on`.

Performs structural induction, using the type of the given term.

Given a term `M`, `Induct_on` attempts to perform an induction based on the type of `M`. The induction theorem to be used is extracted from the `TypeBase` database, which holds useful facts about the system's defined types.

`Induct_on` can be used to specify variables that are buried in the quantifier prefix, i.e., not the leading quantified variable. `Induct_on` can also perform induction on non-variable terms. If `M` is a non-variable term that does not occur bound in the goal, then `Induct_on` equates `M` to a new variable `v` (one not occurring in the goal), moves all hypotheses in which free variables of `M` occur to the conclusion of the goal, adds the antecedent `v = M`, and quantifies all free variables of `M` before universally quantifying `v` and then finally inducting on `v`.

`Induct_on` may also be used to apply an induction theorem coming from declaration of a mutually recursive datatype.

Failure

`Induct_on` fails if an induction theorem corresponding to the type of `M` is not found in the `TypeBase` database.

Example

If attempting to prove

```
!x. LENGTH (REVERSE x) = LENGTH x
```

one can apply `Induct_on `x`` to begin a proof by induction on the list structure of `x`. In this case, `Induct_on` serves as an explicit version of `Induct`.

See also

`bossLib.Induct`, `bossLib.completeInduct_on`, `bossLib.measureInduct_on`, `Prim_rec.INDUCT_THEN`, `bossLib.Cases`, `bossLib.Hol_datatype`, `proofManagerLib.g`, `proofManagerLi`

INDUCT_TAC (numLib)

`numLib.INDUCT_TAC : tactic`

Performs tactical proof by mathematical induction on the natural numbers.

`INDUCT_TAC` reduces a goal `!n.P[n]`, where `n` has type `num`, to two subgoals corresponding to the base and step cases in a proof by mathematical induction on `n`. The induction hypothesis appears among the assumptions of the subgoal for the step case. The specification of `INDUCT_TAC` is:

```
                A ?- !n. P
=======================================  INDUCT_TAC
 A ?- P[0/n]     A u {P} ?- P[SUC n'/n]
```

where `n'` is a primed variant of `n` that does not appear free in the assumptions `A` (usually, `n'` just equals `n`). When `INDUCT_TAC` is applied to a goal of the form `!n.P`, where `n` does not appear free in `P`, the subgoals are just `A ?- P` and `A u {P} ?- P`.

Failure

`INDUCT_TAC g` fails unless the conclusion of the goal `g` has the form `!n.t`, where the variable `n` has type `num`.

INDUCT_THEN (Prim_rec)

```
Prim_rec.INDUCT_THEN : (thm -> thm_tactic -> tactic)
```

Structural induction tactic for automatically-defined concrete types.

The function `INDUCT_THEN` implements structural induction tactics for arbitrary concrete recursive types of the kind definable by `define_type`. The first argument to `INDUCT_THEN` is a structural induction theorem for the concrete type in question. This theorem must have the form of an induction theorem of the kind returned by `prove_induction_thm`. When applied to such a theorem, the function `INDUCT_THEN` constructs specialized tactic for doing structural induction on the concrete type in question.

The second argument to `INDUCT_THEN` is a function that determines what is be done with the induction hypotheses in the goal-directed proof by structural induction. Suppose that `th` is a structural induction theorem for a concrete data type `ty`, and that `A ?- !x.P` is a universally-quantified goal in which the variable `x` ranges over values of type `ty`. If the type `ty` has `n` constructors `C1`, ..., `Cn` and '`Ci(vs)`' represents a (curried) application of the `i`th constructor to a sequence of variables, then if `ttac` is a function that maps the induction hypotheses `hypi` of the `i`th subgoal to the tactic:

```
    A  ?- P[Ci(vs)/x]
=====================  MAP_EVERY ttac hypi
      A1 ?- Gi
```

then `INDUCT_THEN th ttac` is an induction tactic that decomposes the goal `A ?- !x.P` into a set of `n` subgoals, one for each constructor, as follows:

```
          A ?- !x.P
================================  INDUCT_THEN th ttac
  A1 ?- G1  ...   An ?- Gn
```

The resulting subgoals correspond to the cases in a structural induction on the variable `x` of type `ty`, with induction hypotheses treated as determined by `ttac`.

Failure

`INDUCT_THEN th ttac g` fails if `th` is not a structural induction theorem of the form returned by `prove_induction_thm`, or if the goal does not have the form `A ?- !x:ty.P` where `ty` is the type for which `th` is the induction theorem, or if `ttac` fails for any subgoal in the induction.

Example

The built-in structural induction theorem for lists is:

```
|- !P. P[] /\ (!t. P t ==> (!h. P(CONS h t))) ==> (!l. P l)
```

When `INDUCT_THEN` is applied to this theorem, it constructs and returns a specialized induction tactic (parameterized by a theorem-tactic) for doing induction on lists:

```
- val LIST_INDUCT_THEN = INDUCT_THEN listTheory.list_INDUCT;
```

The resulting function, when supplied with the `thm_tactic` `ASSUME_TAC`, returns a tactic that decomposes a goal `?- !l.P[l]` into the base case `?- P[NIL]` and a step case `P[l] ?- !h. P[CONS h l]`, where the induction hypothesis `P[l]` in the step case has been put on the assumption list. That is, the tactic:

```
LIST_INDUCT_THEN ASSUME_TAC
```

does structural induction on lists, putting any induction hypotheses that arise onto the assumption list:

```
                    A ?- !l. P
=======================================================
 A |- P[NIL/l]   A u {P[l'/l]} ?- !h. P[(CONS h l')/l]
```

Likewise `LIST_INDUCT_THEN STRIP_ASSUME_TAC` will also do induction on lists, but will strip induction hypotheses apart before adding them to the assumptions (this may be useful if P is a conjunction or a disjunction, or is existentially quantified). By contrast, the tactic:

```
LIST_INDUCT_THEN MP_TAC
```

will decompose the goal as follows:

```
                    A ?- !l. P
  ======================================================
   A |- P[NIL/l]   A ?- P[l'/l] ==> !h. P[CONS h l'/l]
```

That is, the induction hypothesis becomes the antecedent of an implication expressing the step case in the induction, rather than an assumption of the step-case subgoal.

See also

`Prim_rec.new_recursive_definition`, `Prim_rec.prove_cases_thm`, `Prim_rec.prove_constructors_distinct`, `Prim_rec.prove_constructors_one_one`, `Prim_rec.prove_induction_thm`, `Prim_rec.prove_rec_fn_exists`

Induct_word (wordsLib)

`wordsLib.Induct_word : tactic`

Initiate an induction on the value of a word.

The tactic `Induct_word` makes use of the tactic `bossLib.recInduct wordsTheory.WORD_INDUCT`.

Example

Given the goal

```
?- !w:word8. P w
```

one can apply `Induct_word` to begin a proof by induction.

```
> e wordsLib.Induct_word
Exception- OK..
HOL_ERR (at Induction.ndest_forall: too few quantified variables in goal) raised
```

This results in the base and step cases of the induction as new goals.

```
?- P 0w
```

```
[SUC n < 256, P (n2w n)] ?- P (n2w (SUC n))
```

See also

`bossLib.recInduct`

inferences (Count)

```
Count.inferences : ('a -> 'b) -> 'a -> 'b
```

Counts primitive inferences performed when a function is applied.

The `inferences` function provides a way of counting the primitive inferences that are performed when a function is applied to its argument. The reporting of the count is done when the function terminates (normally, or with an exception).

Example

```
> Count.apply (CONJUNCTS o SPEC_ALL) AND_CLAUSES;
runtime: 0.00001s,    gctime: 0.00000s,     systime: 0.00000s.
Axioms: 0, Defs: 0, Disk: 0, Orcl: 0, Prims: 9; Total: 9
val it = [⊢ T ∧ t ⇔ t, ⊢ t ∧ T ⇔ t, ⊢ F ∧ t ⇔ F, ⊢ t ∧ F ⇔ F, ⊢ t ∧ t ⇔ t]:
   thm list
```

Failure

The call to `inferences f x` will raise an exception if `f x` would. It will still report inference counts up to the point of the exception being raised.

See also

`Count.apply`

insert (Lib)

```
Lib.insert ''a -> ''a list -> ''a list
```

Add an element to a list if it is not already there.

If `x` is already in `list`, then `insert x list` equals `list`. Otherwise, `x` becomes an element of `list`.

Failure

Never fails.

Example

```
> insert 1 [3,2];
val it = [1, 3, 2]: int list

> insert 1 it;
val it = [1, 3, 2]: int list
```

Comments

In some programming situations, it is convenient to implement sets by lists, in which case `insert` may be helpful. However, such an implementation is only suitable for small sets.

A high-performance implementation of finite sets may be found in structure `HOLset`.

ML equality types are used in the implementation of `insert` and its kin. This limits its applicability to types that allow equality. For other types, typically abstract ones, use the 'op_' variants.

One should not write code that depends on where the 'list-as-set' algorithms place elements in the list which is being considered as a set.

See also

`Lib.op_insert`, `Lib.mem`, `Lib.mk_set`, `Lib.union`, `Lib.U`, `Lib.set_diff`, `Lib.subtract`, `Lib.intersect`, `Lib.null_intersection`, `Lib.set_eq`

INSERT_CONV (pred_setLib)

```
pred_setLib.INSERT_CONV : conv -> conv
```

Reduce `t INSERT {t1;...;t;...;tn}` to `{t1;...;t;...;tn}`.

The function `INSERT_CONV` is a parameterized conversion for reducing finite sets of the form `t INSERT {t1;...;tn}`, where `{t1;...;tn}` is a set of type `ty->bool` and `t` is equal to some element `ti` of this set. The first argument to `INSERT_CONV` is expected to be a conversion that decides equality between values of the base type `ty`. Given an equation `e1 = e2`, where `e1` and `e2` are terms of type `ty`, this conversion should return the theorem `|- (e1 = e2) = T` or the theorem `|- (e1 = e2) = F`, as appropriate.

Given such a conversion, the function `INSERT_CONV` returns a conversion that maps a term of the form `t INSERT {t1;...;tn}` to the theorem

```
|- t INSERT {t1;...;tn} = {t1;...;tn}
```

if `t` is alpha-equivalent to any `ti` in the set `{t1,...,tn}`, or if the supplied conversion proves `|- (t = ti) = T` for any `ti`.

Example

In the following example, the conversion `REDUCE_CONV` is supplied as a parameter and used to test equality of the inserted value `2` with the remaining elements of the set.

```
- INSERT_CONV REDUCE_CONV ``2 INSERT {1;SUC 1;3}``;
> val it = |- {2; 1; SUC 1; 3} = {1; SUC 1; 3} : thm
```

In this example, the supplied conversion `REDUCE_CONV` is able to prove that `2` is equal to `SUC 1` and the set is therefore reduced. Note that `2 INSERT {1; SUC 1; 3}` is just `{2; 1; SUC 1; 3}`.

A call to `INSERT_CONV` fails when the value being inserted is provably not equal to any of the remaining elements:

```
- INSERT_CONV REDUCE_CONV ``1 INSERT {2;3}``;
! Uncaught exception:
! HOL_ERR
```

But this failure can, if desired, be caught using `TRY_CONV`.

The behaviour of the supplied conversion is irrelevant when the inserted value is alpha-equivalent to one of the remaining elements:

```
- INSERT_CONV NO_CONV ``y INSERT {x;y;z}``;
> val it = |- {y; x; y; z} = {x; y; z} : thm
```

The conversion `NO_CONV` always fails, but `INSERT_CONV` is nontheless able in this case to prove the required result.

Note that `DEPTH_CONV(INSERT_CONV conv)` can be used to remove duplicate elements from a finite set, but the following conversion is faster:

```
- fun SETIFY_CONV conv tm =
   (SUB_CONV (SETIFY_CONV conv)
     THENC
    TRY_CONV (INSERT_CONV conv)) tm;
> val SETIFY_CONV = fn : conv -> conv

- SETIFY_CONV REDUCE_CONV ``{1;2;1;3;2;4;3;5;6}``;
> val it = |- {1; 2; 1; 3; 2; 4; 3; 5; 6} = {1; 2; 4; 3; 5; 6} : thm
```

Failure

`INSERT_CONV conv` fails if applied to a term not of the form `t INSERT {t1;...;tn}`. A call `INSERT_CONV conv ``t INSERT {t1;...;tn}` fails unless `t` is alpha-equivalent to some `ti`, or `conv ``t = ti``` returns `|- (t = ti) = T` for some `ti`.

See also

`pred_setLib.DELETE_CONV`, `numLib.REDUCE_CONV`

inst (Term)

```
Term.inst : (hol_type,hol_type)subst -> term -> term
```

Performs type instantiations in a term.

The function `inst` should be used as follows:

```
inst [{redex_1, residue_1},...,{redex_n, residue_n}] tm
```

where each 'redex' is a `hol_type` variable, and each 'residue' is a `hol_type` and `tm` a term to be type-instantiated. This call will replace each occurrence of a `redex` in `tm` by its associated `residue`. Replacement is done in parallel, i.e., once a `redex` has been replaced by its `residue`, at some place in the term, that `residue` at that place will not itself be replaced in the current call. Bound term variables may be renamed in order to preserve the term structure.

Failure

Never fails. A `redex` that is not a variable is simply ignored.

Example

```
> show_types := true;
val it = (): unit

> inst [alpha |-> Type`:num`] (Term`(x:'a) = (x:'a)`)
val it = “(x :num) = x”: term

> inst [bool |-> Type`:num`] (Term`x:bool`);
val it = “(x :bool)”: term

> inst [alpha |-> bool] (mk_abs(Term`x:bool`,Term`x:'a`))
val it = “λ(x' :bool). (x :bool)”: term
```

See also

`Type.type_subst`, `Lib.|->`

INST (Thm)

```
Thm.INST : (term,term) subst -> thm -> thm
```

Instantiates free variables in a theorem.

`INST` is a rule for substituting arbitrary terms for free variables in a theorem.

```
          A |- t                INST [x1 |-> t1,...,xn |-> tn]
   ----------------------------
    A[t1,...,tn/x1,...,xn]
     |-
    t[t1,...,tn/x1,...,xn]
```

Failure

Fails if, for `1 <= i <= n`, some `xi` is not a variable, or if some `xi` has a different type than its intended replacement `ti`.

Example

In the following example a theorem is instantiated for a specific term:

```
- load"arithmeticTheory";

- CONJUNCT1 arithmeticTheory.ADD_CLAUSES;
> val it = |- 0 + m = m : thm

- INST [``m:num`` |-> ``2*x``]
       (CONJUNCT1 arithmeticTheory.ADD_CLAUSES);

val it = |- 0 + (2 * x) = 2 * x : thm
```

See also

`Drule.INST_TY_TERM`, `Thm.INST_TYPE`, `Drule.ISPEC`, `Drule.ISPECL`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SUBS`, `Term.subst`, `Thm.SUBST`, `Lib.|->`

INST_TT_HYPS (Drule)

```
Drule.INST_TT_HYPS :
(term,term)subst * (hol_type,hol_type)subst -> thm -> thm * term list
```

Instantiates terms and types of a theorem.

INST_TT_HYPS instantiates types and terms in a theorem thm, in the same way INST_TY_TERM does. It also returns a list of the instantiated hypotheses, in the same order as the uninstantiated hypotheses appear in the list hyp thm.

Failure

INST_TT_HYPS fails under the same conditions as INST_TY_TERM.

See also

Drule.INST_TY_TERM, Thm.hyp

INST_TY_TERM (Drule)

```
Drule.INST_TY_TERM :
(term,term)subst * (hol_type,hol_type)subst -> thm -> thm
```

Instantiates terms and types of a theorem.

INST_TY_TERM instantiates types in a theorem, in the same way INST_TYPE does. Then it instantiates some or all of the free variables in the resulting theorem, in the same way as INST.

Comments

Because the types are instantiated first, the terms (redexes as well as residues) in the term substitution must contain the substituted types, not the original ones. Use norm_subst to achieve this.

Failure

INST_TY_TERM fails under the same conditions as either INST or INST_TYPE fail.

See also

`Thm.INST`, `Thm.INST_TYPE`, `Drule.ISPEC`, `Thm.SPEC`, `Drule.SUBS`, `Thm.SUBST`, `Term.norm_subst`, `Drule.INST_TT_HYPS`

INST_TYPE (Thm)

```
Thm.INST_TYPE : (hol_type,hol_type) subst -> thm -> thm
```

Instantiates types in a theorem.

`INST_TYPE` is a primitive rule in the HOL logic, which allows instantiation of type variables.

```
              A |- t
  ---------------------------------- INST_TYPE[vty1|->ty1,..., vtyn|->tyn]
   A[ty1,...,tyn/vty1,...,vtyn]
    |-
   t[ty1,...,tyn/vty1,...,vtyn]
```

Type substitution is performed throughout the hypotheses and the conclusion. Variables will be renamed if necessary to prevent distinct bound variables becoming identical after the instantiation.

Failure

Never fails.

`INST_TYPE` enables polymorphic theorems to be used at any type.

Example

Supposing one wanted to specialize the theorem `EQ_SYM_EQ` for particular values, the first attempt could be to use `SPECL` as follows:

```
- SPECL [``a:num``, ``b:num``] EQ_SYM_EQ;
uncaught exception HOL_ERR
```

The failure occurred because EQ_SYM_EQ contains polymorphic types. The desired specialization can be obtained by using INST_TYPE:

```
- load "numTheory";

- SPECL [Term `a:num`, Term`b:num`]
        (INST_TYPE [alpha |-> Type`:num`] EQ_SYM_EQ);

> val it = |- (a = b) = (b = a) : thm
```

See also

Term.inst, Thm.INST, Drule.INST_TY_TERM, Lib.|->

inst_word_lengths (wordsLib)

```
wordsLib.inst_word_lengths : term -> term
```

Guess and instantiate word index type variables in a term.

The function inst_word_lengths tries to instantiate type variables that correspond with the return type of word_concat and word_extract.

Example

```
> load "wordsLib";
val it = (): unit
> wordsLib.inst_word_lengths “(7 >< 5) a @@ (4 >< 0) a”;
val it = “(7 >< 5) a @@ (4 >< 0) a”: term
> type_of it;
val it = “:word8”: hol_type
```

Comments

The function guess_lengths adds inst_word_lengths as a post-processing stage to the term parser.

See also

`wordsLib.guess_lengths`, `wordsLib.notify_on_word_length_guess`

INSTANCE_T_CONV (Arith)

```
Arith.INSTANCE_T_CONV : ((term -> term list) -> conv -> conv)
```

Function which allows a proof procedure to work on substitution instances of terms that are in the domain of the procedure.

This function generalises a conversion that is used to prove formulae true. It does this by first replacing any syntactically unacceptable subterms with variables. It then attempts to prove the resulting generalised formula and if successful it re-instantiates the variables.

The first argument should be a function which computes a list of subterms of a term which are syntactically unacceptable to the proof procedure. This function should include in its result any variables that do not appear in other subterms returned. The second argument is the proof procedure to be generalised; this should be a conversion which when successful returns an equation between the argument formula and `T` (true).

Failure

Fails if either of the applications of the argument functions fail, or if the conversion does not return an equation of the correct form.

Example

```
#FORALL_ARITH_CONV "!f m (n:num). (m < (f n)) ==> (m <= (f n))";;
evaluation failed     FORALL_ARITH_CONV -- formula not in the allowed subset

#INSTANCE_T_CONV non_presburger_subterms FORALL_ARITH_CONV
# "!f m (n:num). (m < (f n)) ==> (m <= (f n))";;
|- (!f m n. m < (f n) ==> m <= (f n)) = T
```

int_sort (Lib)

```
Lib.int_sort : int list -> int list
```

Sorts a list of integers using the <= relation.

The call `int_sort list` is equal to `sort (curry (op <=))`. That is, it is the specialization of `sort` to the usual notion of less-than-or-equal on ML integers.

Failure

Never fails.

Example

A simple example is:

```
> int_sort [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9];
val it = [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9, 9, 9]: int list
```

Comments

The Standard ML Basis Library also provides implementations of sorting.

See also

`Lib.sort`

int_to_string (Lib)

```
Lib.int_to_string : int -> string
```

Translates an integer into a string.

An application `int_to_string i` returns the printable form of `i`.

Failure

Never fails.

Example

```
> int_to_string 12323;
val it = "12323": string

> int_to_string ~1;
val it = "~1": string
```

Comments

Equivalent functionality can be found in the Standard ML Basis Library function `Int.toString`.

See also

`Lib.string_to_int`

interactive (Globals)

```
Globals.interactive : bool ref
```

Variable specifying whether system is in interaction mode or batch mode.

The HOL system is either running via the "Read-Eval-Print" loop of the host SML system, or it is running in batch mode, for example via `Holmake`. Some system facilities, for example error handling, need to access this knowledge.

See also

`Feedback`, `Feedback.HOL_ERR`, `Feedback.render_exn`

intersect (Lib)

```
Lib.intersect : ''a list -> ''a list -> ''a list
```

Computes the intersection of two 'sets'.

`intersect l1 l2` returns a list consisting of those elements of `l1` that also appear in `l2`.

Failure

Never fails.

Example

```
> intersect [1,2,3] [3,5,4,1];
val it = [1, 3]: int list
```

Comments

Do not make the assumption that the order of items in the list returned by `intersect` is fixed. Later implementations may use different algorithms, and return a different concrete result while still meeting the specification.

A high-performance implementation of finite sets may be found in structure `HOLset`.

ML equality types are used in the implementation of `intersect` and its kin. This limits its applicability to types that allow equality. For other types, typically abstract ones, use the 'op_' variants.

See also

`Lib.op_intersect`, `Lib.union`, `Lib.U`, `Lib.mk_set`, `Lib.mem`, `Lib.insert`, `Lib.set_eq`, `Lib.set_diff`

IPSPEC (PairRules)

```
PairRules.IPSPEC : (term -> thm -> thm)
```

Specializes a theorem, with type instantiation if necessary.

This rule specializes a paired quantification as does PSPEC; it differs from it in also instantiating the type if needed:

```
    A |- !p:ty.tm
----------------------  IPSPEC "q:ty'"
    A |- tm[q/p]
```

(where q is free for p in tm, and ty' is an instance of ty).

Failure

IPSPEC fails if the input theorem is not universally quantified, if the type of the given term is not an instance of the type of the quantified variable, or if the type variable is free in the assumptions.

See also

Drule.ISPEC, Drule.INST_TY_TERM, Thm.INST_TYPE, PairRules.IPSPECL, PairRules.PSPEC, DB.match

IPSPECL (PairRules)

```
PairRules.IPSPECL : (term list -> thm -> thm)
```

Specializes a theorem zero or more times, with type instantiation if necessary.

IPSPECL is an iterative version of IPSPEC

```
       A |- !p1...pn.tm
 --------------------------  IPSPECL ["q1",...,"qn"]
  A |- t[q1,...qn/p1,...,pn]
```

(where qi is free for pi in tm).

Failure

IPSPECL fails if the list of terms is longer than the number of quantified variables in the term, if the type instantiation fails, or if the type variable being instantiated is free in the assumptions.

See also

Drule.ISPECL, Thm.INST_TYPE, Drule.INST_TY_TERM, PairRules.IPSPEC, Thm.SPEC, PairRules.PSPECL

irule (Tactic)

```
Tactic.irule : thm_tactic
```

Reduces the goal using a supplied implication, with matching.

When applied to a theorem of the form

```
A' |- !x1...xn. s ==> !y1...ym. t ==> !z1...zk. u
```

irule produces a tactic that reduces a goal whose conclusion u' is a substitution and/or type instance of u to the corresponding instances of s and of t. Any variables free in s or t but not in u will be existentially quantified in the resulting subgoal, and a variable free in both s and t will result in a subgoal which is s /\ t, existentially quantified

The following diagram is simplified: more implications and quantified variables than shown are allowed.

```
    A ?- u'
========================  irule (A' |- !x. s ==> !y. t ==> u)
 A ?- (?z. s') /\ ?w. t'
```

where z and w are (type instances of) variables among x, y that do not occur free in s, t. The assumptions A', instantiated, are added as further new subgoals.

Failure

Fails unless the theorem, when stripped of universal quantification and antecedents of implications, can be instantiated to match the goal.

Comments

The supplied theorem is pre-processed using `IRULE_CANON`, which removes the universal quantifiers and antecedents of implications, and existentially quantifies variables which were instantiated but appeared only in the antecedents of implications.

Then `MATCH_MP_TAC` or `MATCH_ACCEPT_TAC` is applied (depending on whether or not the result of the preprocessing is an implication). To avoid preprocessing entirely, one can use `prim_irule`.

Example

With goal `R a (f b)` and theorem `thm` being

```
|- !x u. P u x ==> !w y. Q w x y ==> R x (f y)
```

the proof step `e (irule thm)` gives new goal `(?u. P u a) /\ ?w. Q w a b`.

With goal `a = b` and theorem `trans`

```
|- !x y. (x = y) ==> !z. (y = z) ==> (x = z)
```

the proof step `e (irule trans)` gives new goal

```
?y. (a = y) /\ (y = b)
```

See also

`Tactic.irule_at`, `Tactic.MATCH_MP_TAC`, `Tactic.prim_irule`, `Drule.IRULE_CANON`

irule_at (Tactic)

```
Tactic.irule_at : match_position -> thm -> tactic
```

Applies an implicational theorem backwards in a particular position in the goal

A call to `irule_at pos th`, with `th` an "implicational" theorem of general form `∀xs... P ⇒ Q`, will attempt to find an instance of term `Q` at position `pos` within the current goal, and replace it with an appropriately instantiated version of `P`. (It is possible for `P` to be empty, in which case the term is effectively replaced by truth.) The possible positions encoded by `pos` are all "positive", meaning that this process is sound (it may nonetheless turn a provable goal into an unprovable one).

The possible positions encoded by parameter `pos` view the goal as if it is of form `?ys. c1 ∧ ... ∧ cn`, where the existential prefix `ys` may be empty, and where there may only be one conjunct. If `pos` encodes the choice of conjunct `cj`, then `irule_at pos` will instantiate type variables and term variables from `xs` in `th`, and variables from `ys` in the goal so as to make `cj` unify with `Q`. The conjunct `cj` will then be replaced with the correspondingly instantiated `P`, which may itself be multiple conjunctions. While the goal may lose variables from `ys` because they have been instantiated, it may also acquire new variables from `xs`; these will be added to the goal's existential prefix.

The new goal will be assembled to put new conjuncts first, followed by conjuncts from the original goal in their original order (these conjuncts may still be different if existential variables from `ys` have been instantiated). If two conjuncts become the same because of variable instantiation, only one will be present in the resulting goal. There is some effort made to keep variables from the existential prefix with the same names, but some renaming may occur, and the new goal's existential variables will be ordered arbitrarily. If the new goal has no conjuncts, then the tactic has proved the original.

There are four possible forms for the `pos` parameter. If it is of form `Pos f`, `f` will be a function of type `term list -> term`, and this function will be passed the list of conjuncts. The returned term should be one of those conjuncts. Typical values for this function are `hd`, `last` and `el i`, for positive integers `i`. If the `pos` parameter is of form `Pat q`, the quotation `q` will be parsed in the context of the goal (honouring the goal's free variables), generating a set of possible terms (multiple terms are possible because of ambiguities caused by overloading) that are then viewed as patterns against which the conjuncts of the goal are matched. The first conjunct that matches the earliest pattern in the sequence of possible parses, and which unifies with `th`'s conclusion, is used.

The pattern form `Concl` is used to indicate that the entire goal (including its existential prefix) should be viewed as the desired target for `th`. This results in a call to `irule th` being made.

Finally, the pattern form `Any` is used to have the tactic search for any conjunct that matches the conclusion (as with a pattern of `'_:bool'`), and if no conjunct unifies with

`th`'s conclusion, to then try to call `irule th`, as is done with the `Concl` pattern form.

Failure

Fails if the position parameter fails to specify a term, or if that term does not unify with the implicational theorem's conclusion. A position may fail to specify a term in mulitple ways depending on the form of the position. A position of form `Pos f` will fail if the function `f` fails when applied to the goal's conjuncts. (Note that there is no failure if `f` returns a term that is not actually a conjunct; if this term unifies, this will simply result in new conjuncts appearing in the goal without any old conjuncts being removed.)

A position of form `Pat q` can fail if no conjunct of the goal matches any of the terms parsed to by `q`. The other position forms always return at least one term to be considered. Failure after this point will only follow if none of these terms unifies with the implicational theorem's conclusion.

Example

Solving a goal outright:

```
   ?- ∃x. x ≤ 3
============== irule_at (Pos hd) (DECIDE "y ≤ y")
```

Refining a goal under an existential prefix (the theorem `RTC_SUBSET` states that `∀x y. R x y ⇒ RTC R x y`):

```
   ?- ∃x y z. P x ∧ RTC R x (f y) ∧ Q y z
======================================== irule_at Any RTC_SUBSET
   ?- ∃z y0 x. R x (f y0) ∧ P x ∧ Q y0 z
```

Instantiating existential variables (with `LESS_MONO` stating that `m < n ⇒ SUC m < SUC n`):

```
   ?- ∃x y z. P x ∧ SUC x < y ∧ Q y z
==================================== irule_at Any LESS_MONO
   ?- ∃z n m. m < n ∧ P m ∧ Q (SUC n) z
```

Comments

The underlying operation is resolution, where one resolvent is always the conclusion of `th`, and the other is the conjunct from the goal selected by the position parameter. The goal is viewed as a literal clause by negating it (via the action of `goal_assum`).

See also

Tactic.irule, Tactic.MATCH_MP_TAC

IRULE_CANON (Drule)

```
Drule.IRULE_CANON : thm -> thm
```

Canonicalises a theorem for use as an introduction rule.

A call to `IRULE_CANON th` returns a theorem `th'` that is equivalent to `th`, but syntactically rearranged to be in the form

```
!v1 .. vn. c1 /\ c2 ... /\ cm ==> c
```

(also allowing for no conjuncts at all). The variables `v1` to `vn` all occur in the conclusion `c`, which is not universally quantified, nor an implication.

Each of the conjuncts is of the form

```
?ev1 .. evi. ec1 /\ .. ecj
```

where it is possible that there are not existentially quantified variables. The existential quantification ensures that there are no free variables in the output theorem `th'`.

Failure

Never fails.

Comments

This function is used within the implementation of `irule`. The output theorem `th'` is appropriate for use as an argument to `MATCH_MP_TAC` (if the output is a quantified implication), or `MATCH_ACCEPT_TAC` if the output is not an implication.

See also

Tactic.irule, Tactic.MATCH_MP_TAC, Drule.RES_CANON

is_abs (Term)

```
Term.is_abs : (term -> bool)
```

Tests a term to see if it is an abstraction.

`is_abs "\var. t"` returns `true`. If the term is not an abstraction the result is `false`.

Failure

Never fails.

See also

`Term.mk_abs`, `Term.dest_abs`, `Term.is_var`, `Term.is_const`, `Term.is_comb`

is_arb (boolSyntax)

```
boolSyntax.is_arb : term -> bool
```

Tests a term to see if it's an instance of `ARB`.

Returns `true` if and only if `M` has the form `ARB`.

None known.

See also

`boolSyntax.mk_arb`, `boolSyntax.dest_arb`

is_bool_case (boolSyntax)

```
boolSyntax.is_bool_case : term -> bool
```

Tests a case expression over `bool`.

If `M` has the form `bool_case M1 M2 b`, then `is_bool_case M` returns `true`. Otherwise, it returns `false`.

Failure

Never fails.

See also

`boolSyntax.mk_bool_case`, `boolSyntax.dest_bool_case`

is_comb (Term)

```
Term.is_comb : term -> bool
```

Tests a term to see if it is a combination (function application).

If term `M` has the form `f x`, then `is_comb M` equals `true`. Otherwise, the result is `false`.

Failure

Never fails

See also

`Term.mk_comb`, `Term.dest_comb`, `Term.is_var`, `Term.is_const`, `Term.is_abs`

is_cond (boolSyntax)

```
boolSyntax.is_cond : term -> bool
```

Tests a term to see if it is a conditional.

If `M` has the form `if t then t1 else t2` then `is_cond M` returns `true` If the term is not a conditional the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_cond`, `boolSyntax.dest_cond`

is_conj (boolSyntax)

```
boolSyntax.is_conj : term -> bool
```

Tests a term to see if it is a conjunction.

If `M` has the form `t1 /\ t2`, then `is_conj M` returns `true`. If `M` is not a conjunction the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_conj`, `boolSyntax.dest_conj`

is_cons (listSyntax)

```
listSyntax.is_cons : (term -> bool)
```

Tests a term to see if it is an application of `CONS`.

`is_cons` returns `true` of a term representing a non-empty list. Otherwise it returns `false`.

Failure

Never fails.

See also

`listSyntax.mk_cons`, `listSyntax.dest_cons`, `listSyntax.mk_list`, `listSyntax.dest_list`, `listSyntax.is_list`

is_const (Term)

```
Term.is_const : term -> bool
```

Tests a term to see if it is a constant.

If `c` is an instance of a previously declared HOL constant, then `is_const c` returns `true`; otherwise the result is `false`.

Failure

Never fails.

See also

`Term.mk_const`, `Term.dest_const`, `Term.is_var`, `Term.is_comb`, `Term.is_abs`

is_disj (boolSyntax)

```
boolSyntax.is_disj : term -> bool
```

Tests a term to see if it is a disjunction.

If `M` has the form `t1 \/ t2`, then `is_disj M` returns `true`. If `M` is not a disjunction the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_disj`, `boolSyntax.dest_disj`

IS_EL_CONV (listLib)

```
listLib.IS_EL_CONV : conv -> conv
```

Computes by inference the result of testing whether a list contains a certain element.

`IS_EL_CONV` takes a conversion `conv` and a term `tm` in the following form:

```
IS_EL x [x0;...xn]
```

It returns the theorem

```
|- IS_EL x [x0;...xn] = F
```

if for every `xi` occurred in the list, `conv` "`x = xi`" returns a theorem `|- P xi = F`, otherwise, if for at least one `xi`, evaluating `conv` "`P xi`" returns the theorem `|- P xi = T`, then it returns the theorem

```
|- IS_EL P [x0;...xn] = T
```

Failure

`IS_EL_CONV conv tm` fails if `tm` is not of the form described above, or failure occurs when evaluating `conv` "`x = xi`" for some `xi`.

Example

Evaluating

```
IS_EL_CONV bool_EQ_CONV “IS_EL T [T;F;T]”;
```

returns the following theorem:

```
|- IS_EL($= T)[F;F] = F
```

See also

`listLib.SOME_EL_CONV`, `listLib.ALL_EL_CONV`, `listLib.FOLDL_CONV`, `listLib.FOLDR_CONV`, `listLib.list_FOLD_CONV`

is_eq (boolSyntax)

```
boolSyntax.is_eq : term -> bool
```

Tests a term to see if it is an equation.

If `M` has the form `t1 = t2` then `is_eq M` returns `true`. If `M` is not an equation the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_eq`, `boolSyntax.dest_eq`

is_exists (boolSyntax)

```
boolSyntax.is_exists : term -> bool
```

Tests a term to see if it is an existential quantification.

If `M` has the form `?v. t` then `is_exists M` returns `true`. If the term is not an existential quantification the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_exists`, `boolSyntax.dest_exists`

is_exists1 (boolSyntax)

```
boolSyntax.is_exists1 : term -> bool
```

Tests a term to see if it is a unique existence term.

If `M` has the form `?!v. t` then `is_exists1 M` returns `true`. If the term is not a unique existence quantification the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_exists1`, `boolSyntax.dest_exists`

is_forall (boolSyntax)

```
boolSyntax.is_forall : term -> bool
```

Tests a term to see if it is a universal quantification.

If `M` is a term with the form `!x. t`, then `is_forall M` returns `true`. If `M` is not a universal quantification the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_forall`, `boolSyntax.dest_forall`

is_gen_tyvar (Type)

```
Type.is_gen_tyvar : hol_type -> bool
```

Checks if a type variable has been created by `gen_tyvar`.

Failure

Never fails.

Example

```
> is_gen_tyvar (gen_tyvar());
val it = true: bool

> is_gen_tyvar bool;
val it = false: bool
```

See also

`Type.gen_tyvar`

is_genvar (Term)

```
Term.is_genvar : term -> bool
```

Tells if a variable has been built by invoking `genvar`.

`is_genvar v` attempts to tell if `v` has been created by a call to `genvar`.

Failure

Never fails.

Example

```
> is_genvar (genvar bool);
val it = true: bool

> is_genvar (mk_var ("%%genvar%%3",bool));
val it = true: bool
```

Comments

As the second example shows, it is possible to fool `is_genvar`. However, it is useful for derived proof tools which use it as part of their internal operations.

See also

`Term.is_var`, `Term.genvar`, `Type.is_gen_tyvar`, `Type.gen_tyvar`

is_imp (boolSyntax)

```
boolSyntax.is_imp : term -> bool
```

Tests a term to see if it is an implication or a negation.

If `M` has the form `t1 ==> t2`, or the form `~t`, then `is_imp M` returns `true`. If the term is neither an implication nor a negation the result is `false`.

Failure

Never fails.

Comments

Yields true of negations because `dest_imp` destructs negations (for backwards compatibility with PPLAMBDA). Use `is_imp_only` if you don't want this behaviour.

See also

`boolSyntax.mk_imp`, `boolSyntax.dest_imp`, `boolSyntax.is_imp_only`, `boolSyntax.dest_imp_only`

is_imp_only (boolSyntax)

```
boolSyntax.is_imp_only : term -> bool
```

Tests a term to see if it is an implication.

If `M` has the form `t1 ==> t2` then `is_imp_only M` returns `true`. If the term is not an implication, the result is `false`.

Failure

Never fails.

See also

`boolSyntax.is_imp`, `boolSyntax.mk_imp`, `boolSyntax.dest_imp`, `boolSyntax.dest_imp_only`, `boolSyntax.list_mk_imp`, `boolSyntax.strip_imp`

is_let (boolSyntax)

```
boolSyntax.is_let : term -> bool
```

Tests a term to see if it is a `let`-expression.

If `tm` is a term of the form `LET M N`, then `dest_let tm` returns `true`. Otherwise, it returns `false`.

Failure

Never fails.

Example

```
> Term `LET f x`;
val it = “LET f x”: term

> is_let it;
val it = true: bool

> is_let (Term `let x = P /\ Q in x \/ x`);
Exception- HOL_ERR
  (at boolpp.Term: on line 1, characters 18-27: let with non-equality) raised
```

See also

`boolSyntax.mk_let`, `boolSyntax.dest_let`

is_list (listSyntax)

```
listSyntax.is_list : (term -> bool)
```

Tests a term to see if it is a list.

`is_list` returns `true` of a term representing a list. Otherwise it returns `false`.

Failure

Never fails.

See also

`listSyntax.mk_list`, `listSyntax.dest_list`, `listSyntax.mk_cons`, `listSyntax.dest_cons`, `listSyntax.is_cons`

is_neg (boolSyntax)

```
boolSyntax.is_neg : term -> bool
```

Tests a term to see if it is a negation.

If `M` has the form `~t`, then `is_neg M` returns `true`. If the term is not a negation the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_neg`, `boolSyntax.dest_neg`

is_numeral (numSyntax)

```
numSyntax.is_numeral : term -> bool
```

Check if HOL term is a numeral.

An invocation `is_numeral tm`, where `tm` is a HOL term with the following form

```
<numeral> ::= 0 | NUMERAL <bits>
<bits>    ::= ZERO | BIT1 (<bits>) | BIT2 (<bits>)
```

returns `true`; otherwise, `false` is returned. The `NUMERAL` constant is used as a tag signalling that its argument is indeed a numeric literal. The `ZERO` constant is equal to `0`, and `BIT1(n) = 2*n + 1` while `BIT2(n) = 2*n + 2`. This representation allows asymptotically efficient operations on numeric values.

The system prettyprinter will print a numeral as a string of digits.

Example

```
> numSyntax.is_numeral ``1234``;
val it = true: bool
```

Failure

Fails if `tm` is not in the specified format.

See also

`numSyntax.dest_numeral`, `numSyntax.mk_numeral`

is_pabs (pairSyntax)

```
pairSyntax.is_pabs : term -> bool
```

Tests a term to see if it is a paired abstraction.

`is_pabs "\pair. t"` returns `true`. If the term is not a paired abstraction the result is `false`.

Failure

Never fails.

See also

`Term.is_abs`, `pairSyntax.mk_pabs`, `pairSyntax.dest_pabs`

is_pair (pairSyntax)

```
pairSyntax.is_pair : (term -> bool)
```

Tests a term to see if it is a pair.

`is_pair "(t1,t2)"` returns `true`. If the term is not a pair the result is `false`.

Failure

Never fails.

See also

`pairSyntax.mk_pair`, `pairSyntax.dest_pair`

is_pexists (pairSyntax)

```
pairSyntax.is_pexists : (term -> bool)
```

Tests a term to see if it as a paired existential quantification.

`is_pexists "?pair. t"` returns `true`. If the term is not a paired existential quantification the result is `false`.

Failure

Never fails.

See also

`boolSyntax.is_exists`, `pairSyntax.dest_pexists`

is_pforall (pairSyntax)

```
pairSyntax.is_pforall : (term -> bool)
```

Tests a term to see if it is a paired universal quantification.

`is_pforall "!pair. t"` returns `true`. If the term is not a paired universal quantification the result is `false`.

Failure

Never fails.

See also

`boolSyntax.is_forall`, `pairSyntax.dest_pforall`

is_prenex (Arith)

```
Arith.is_prenex : (term -> bool)
```

Determines whether a formula is in prenex normal form.

This function returns true if the term it is given as argument is in prenex normal form. If the term is not a formula the result will be true provided there are no nested Boolean expressions involving quantifiers.

Failure

Never fails.

Example

```
> Arith.is_prenex ``!x. ?y. x \/ y``;
val it = true: bool

> Arith.is_prenex ``!x. x ==> (?y. x /\ y)``;
val it = false: bool
```

Useful for determining whether it is necessary to apply a prenex normaliser to a formula before passing it to a function which requires the formula to be in prenex normal form.

See also

Arith.PRENEX_CONV

is_presburger (Arith)

```
Arith.is_presburger : (term -> bool)
```

Determines whether a formula is in the Presburger subset of arithmetic.

This function returns true if the argument term is a formula in the Presburger subset of natural number arithmetic. Presburger natural arithmetic is the subset of arithmetic formulae made up from natural number constants, numeric variables, addition, multiplication by a constant, the natural number relations <, <=, =, >=, > and the logical connectives ~, /\, \/, ==>, = (if-and-only-if), ! ('forall') and ? ('there exists').

Products of two expressions which both contain variables are not included in the subset, but the function SUC which is not normally included in a specification of Presburger arithmetic is allowed in this HOL implementation. This function also considers subtraction and the predecessor function, PRE, to be part of the subset.

Failure

Never fails.

Example

```
> Arith.is_presburger ``!m n p. m < (2 * n) /\ (n + n) <= p ==> m < SUC p``;
val it = true: bool

> Arith.is_presburger ``!m n p q. m < (n * p) /\ (n * p) < q ==> m < q``;
val it = false: bool

> Arith.is_presburger ``(m <= n) ==> !p. (m < SUC(n + p))``;
val it = true: bool

> Arith.is_presburger ``(m + n) - m = n``;
val it = true: bool
```

Useful for determining whether a decision procedure for Presburger arithmetic is applicable to a term.

See also

`Arith.non_presburger_subterms`, `Arith.FORALL_ARITH_CONV`, `Arith.EXISTS_ARITH_CONV`, `Arith.is_prenex`

is_prod (pairSyntax)

```
pairSyntax.is_prod : hol_type -> bool
```

Tests a type to see if it is a product type.

If `ty` is a type of the form `ty1 # ty2`, then `is_prod ty` returns `true`.

Failure

Never fails.

See also

`pairSyntax.dest_prod`, `pairSyntax.mk_prod`

is_pselect (pairSyntax)

```
pairSyntax.is_pselect : (term -> bool)
```

Tests a term to see if it is a paired choice-term.

`is_select "@pair. t"` returns `true`. If the term is not a paired choice-term the result is `false`.

Failure

Never fails.

See also

`boolSyntax.is_select`, `pairSyntax.dest_pselect`

is_ptree (patriciaLib)

```
patriciaLib.is_ptree : term -> bool
```

Term recogniser for Patricia trees.

The destructor `is_ptree` will return true if, and only if, the supplied term is a well-constructed, ground Patricia tree.

Example

```
> patriciaLib.is_ptree ``t:unit ptree``;
val it = false: bool

> patriciaLib.is_ptree ``Branch 1 2 (Leaf 2 2) (Leaf 3 3)``;
val it = false: bool
```

```
> patriciaLib.is_ptree ``Branch 0 0 (Leaf 1 1) (Leaf 2 2)``;
val it = true: bool
```

See also

`patriciaLib.mk_ptree`, `patriciaLib.dest_ptree`

is_res_abstract (res_quanLib)

```
res_quanLib.is_res_abstract : term -> bool
```

Tests a term to see if it is a restricted abstraction.

`is_res_abstract "\var::P. t"` returns `true`. If the term is not a restricted abstraction the result is `false`.

Failure

Never fails.

See also

`res_quanLib.mk_res_abstract`, `res_quanLib.dest_res_abstract`

is_res_exists (res_quanLib)

```
res_quanLib.is_res_exists : term -> bool
```

Tests a term to see if it is a restricted existential quantification.

`is_res_exists "?var::P. t"` returns `true`. If the term is not a restricted existential quantification the result is `false`.

Failure

Never fails.

See also

`res_quanLib.mk_res_exists`, `res_quanLib.dest_res_exists`

is_res_exists_unique (res_quanLib)

```
res_quanLib.is_res_exists_unique : term -> bool
```

Tests a term to see if it is a restricted unique existential quantification.

`is_res_exists_unique "?!var::P. t"` returns `true`. If the term is not a restricted unique existential quantification the result is `false`.

Failure

Never fails.

See also

`res_quanLib.mk_res_exists_unique`, `res_quanLib.dest_res_exists_unique`

is_res_forall (res_quanLib)

```
res_quanLib.is_res_forall : term -> bool
```

Tests a term to see if it is a restricted universal quantification.

`is_res_forall "!var::P. t"` returns `true`. If the term is not a restricted universal quantification the result is `false`.

Failure

Never fails.

See also

`res_quanLib.mk_res_forall`, `res_quanLib.dest_res_forall`

is_res_select (res_quanLib)

```
res_quanLib.is_res_select : term -> bool
```

Tests a term to see if it is a restricted choice quantification.

`is_res_select "@var::P. t"` returns `true`. If the term is not a restricted choice quantification the result is `false`.

Failure

Never fails.

See also

`res_quanLib.mk_res_select`, `res_quanLib.dest_res_select`

is_select (boolSyntax)

```
boolSyntax.is_select : (term -> bool)
```

Tests a term to see if it is a choice binding.

`is_select "@var. t"` returns `true`. If the term is not an epsilon-term the result is `false`.

Failure

Never fails.

See also

`boolSyntax.mk_select`, `boolSyntax.dest_select`

is_type (Type)

```
Type.is_type : hol_type -> bool
```

Tests whether a HOL type is not a type variable.

`is_type ty` returns `true` if `ty` is an application of a type operator and `false` otherwise.

Failure

Never fails.

See also

`Type.op_arity`, `Type.mk_type`, `Type.mk_thy_type`, `Type.dest_type`, `Type.dest_thy_type`

is_var (Term)

```
Term.is_var : term -> bool
```

Tests a term to see if it is a variable.

If `M` is a HOL variable, then `is_var M` returns `true`. If the term is not a variable the result is `false`.

Failure

Never fails.

See also

`Term.mk_var`, `Term.dest_var`, `Term.is_const`, `Term.is_comb`, `Term.is_abs`

is_vartype (Type)

```
Type.is_vartype : hol_type -> bool
```

Tests a type to see if it is a type variable.

Failure

Never fails.

Example

```
> is_vartype Type.alpha;
val it = true: bool

> is_vartype bool;
val it = false: bool

> is_vartype (Type `:'a  -> bool`);
val it = false: bool
```

See also

`Type.mk_vartype`, `Type.dest_vartype`

isEmpty (Tag)

```
Tag.isEmpty : tag -> bool
```

Tells if a tag is empty.

An invocation `isEmpty t` returns `true` just in case `t` is the empty tag. Only theorems built solely by HOL proof have an empty tag.

Failure

Never fails.

Example

```
> Tag.isEmpty (Thm.tag NOT_FORALL_THM);
val it = false: bool
```

See also

`Thm.tag`, `Thm.mk_oracle_thm`

ISPEC (Drule)

```
Drule.ISPEC : (term -> thm -> thm)
```

Specializes a theorem, with type instantiation if necessary.

This rule specializes a quantified variable as does `SPEC`; it differs from it in also instantiating the type if needed:

```
    A |- !x:ty.tm
----------------------  ISPEC "t:ty'"
    A |- tm[t/x]
```

(where `t` is free for `x` in `tm`, and `ty'` is an instance of `ty`).

Failure

ISPEC fails if the input theorem is not universally quantified, if the type of the given term is not an instance of the type of the quantified variable, or if the type variable is free in the assumptions.

See also

Drule.INST_TY_TERM, Thm.INST_TYPE, Drule.ISPECL, Thm.SPEC, Term.match_term

ISPECL (Drule)

```
Drule.ISPECL : term list -> thm -> thm
```

Specializes a theorem zero or more times, with type instantiation if necessary.

ISPECL is an iterative version of ISPEC

```
        A |- !x1...xn.t
----------------------------  ISPECL [t1,...,tn]
 A |- t[t1,...tn/x1,...,xn]
```

(where ti is free for xi in tm).

Failure

ISPECL fails if the list of terms is longer than the number of quantified variables in the term, if the type instantiation fails, or if the type variable being instantiated is free in the assumptions.

See also

Thm.INST_TYPE, Drule.INST_TY_TERM, Drule.ISPEC, Drule.PART_MATCH, Thm.SPEC, Drule.SPECL

istream (Lib)

```
Lib.type ('a,'b) istream
```

Type of imperative streams.

The type `('a,'b) istream` is an abstract type of imperative streams. These may be created with `mk_istream`, advanced by `next`, accessed by `state`, and reset with `reset`.

Comments

Purely functional streams are well-known in functional programming, and more elegant. However, this type proved useful in implementing some imperative 'gensym'-like algorithms used in HOL.

See also

`Lib.mk_istream`, `Lib.next`, `Lib.state`, `Lib.reset`

itlist (Lib)

```
Lib.itlist : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

List iteration function. Applies a binary function between adjacent elements of a list.

`itlist f [x1,...,xn] b` returns

```
f x1 (f x2 ... (f xn b)...)
```

An invocation `itlist f list b` returns `b` if `list` is empty.

Failure

Fails if some application of `f` fails.

Example

```
> itlist (curry op+) [1,2,3,4] 0;
val it = 10: int
```

See also

`Lib.itlist2`, `Lib.rev_itlist`, `Lib.rev_itlist2`, `Lib.end_itlist`

itlist2 (Lib)

```
Lib.itlist2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
```

Applies a function to corresponding elements of 2 lists.

`itlist2 f [x1,...,xn] [y1,...,yn] z` returns

```
   f x1 y1 (f x2 y2 ... (f xn yn z)...)
```

An invocation `itlist2 f list1 list2 b` returns `b` if `list1` and `list2` are empty.

Failure

Fails if the two lists are of different lengths, or if one of the applications of `f` fails.

Example

```
> itlist2 (fn x => fn y => fn z => (x,y)::z) [1,2] [3,4] [];
val it = [(1, 3), (2, 4)]: (int * int) list
```

See also

`Lib.itlist`, `Lib.rev_itlist`, `Lib.rev_itlist2`, `Lib.end_itlist`

K (Lib)

```
Lib.K : 'a -> 'b -> 'a
```

Forms a constant function: `K x y` = `x`.

Failure

Never fails.

See also

`Lib.##`, `Lib.C`, `Lib.I`, `Lib.S`, `Lib.W`

KNOW_TAC (Tactic)

```
Tactic.KNOW_TAC : term -> tactic
```

A shorthand form of `SUBGOAL_THEN`.

A call to `KNOW_TAC t` is equivalent to a call to `SUBGOAL_THEN t MP_TAC`.

Failure

A call to `KNOW_TAC t` will fail on being applied to a goal if the provided term `t` is not of boolean type.

Comments

If `t` is to be created through parsing a user-provided string, it may be helpful to do that parsing in the context of the current goal, for which `Q_TAC` may be helpful.

Equally, the `by` and `suffices_by` tactics have a similar effect: taking a quotation, and generating two subgoals to be proved. In all cases, the behaviour is to give the user an opportunity to be creative in the specification of the fresh sub-goal that arises from applying modus ponens backwards.

See also

`bossLib.by`, `Tactical.Q_TAC`, `Tactic.SUBGOAL_THEN`, `Tactic.SUFF_TAC`, `bossLib.suffices_by`

known_constants (Parse)

```
Parse.known_constants : unit -> string list
```

Returns the list of constants known to the parser.

A call to this functions returns the list of constants that will be treated as such by the parser. Those constants with names not on the list will be parsed as if they were variables.

Failure

Never fails.

See also

`Parse.hide`, `Parse.reveal`, `Parse.set_known_constants`

lambdify (bossLib)

```
bossLib.lambdify : thm -> thm
```

Convert a theorem representing a single-line definition into a fully lambda-abstracted version.

Given a theorem which describes an equation for a constant applied to a series of distinct variables, derive a reformulation which equates the constant with a lambda-abstraction over those variables.

To advance a proof by unfolding a partially-applied function. Most effectively used on theorems produced by `oneline`.

Example

Consider the result of applying `oneline` to `listTheory.MAP`:

```
> oneline listTheory.MAP;
val it = ⊢ MAP f v = case v of [] => [] | h::t => f h::MAP f t: thm

> lambdify it;
val it = ⊢ MAP = (λf v. case v of [] => [] | h::t => f h::MAP f t): thm
```

Failure

Fails on theorems of the wrong form, i.e. theorems which are not a single equation with a left-hand side consisting of an application to a series of distinct variables.

Comments

Shorthand for `DefnBase.LIST_HALF_MK_ABS`.

See also

`bossLib.oneline`, `jrhUtils.HALF_MK_ABS`

LAND_CONV (Conv)

```
Conv.LAND_CONV : conv -> conv
```

Applies a conversion to the left-hand argument of a binary operator.

If `c` is a conversion that maps a term `t1` to the theorem `|- t1 = t1'`, then the conversion `LAND_CONV c` maps applications of the form `f t1 t2` to theorems of the form:

```
|- f t1 t2 = f t1' t2
```

Failure

`LAND_CONV c tm` fails if `tm` is not an application where the rator of the application is in turn another application, as `f t1 t2`, or if `tm` has this form but the conversion `c` fails when applied to the term `t1`. The function returned by `LAND_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

Example

```
> LAND_CONV reduceLib.REDUCE_CONV (Term`(3 + 5) * 7`);
val it = ⊢ (3 + 5) * 7 = 8 * 7: thm
```

See also

`Conv.ABS_CONV`, `Conv.BINOP_CONV`, `Conv.RAND_CONV`, `Conv.RATOR_CONV`, `numLib.REDUCE_CONV`, `Conv.LHS_CONV`

last (Lib)

```
Lib.last : 'a list -> 'a
```

Computes the last element of a list.

`last [x1,...,xn]` returns `xn`.

Failure

Fails if the list is empty.

See also

`Lib.butlast`, `Lib.el`, `Lib.front_last`

LAST_ASSUM (Tactical)

```
Tactical.LAST_ASSUM : (thm_tactic -> tactic)
```

Maps a theorem-tactic over the assumptions, in reverse order, applying first successful tactic.

`LAST_ASSUM` behaves like `FIRST_ASSUM`, except that it goes through the list of assumptions in reverse order

See also

`Tactical.FIRST_ASSUM`, `Tactical.LAST_X_ASSUM`

LAST_ASSUME_TAC (Tactic)

```
Tactic.LAST_ASSUME_TAC : thm_tactic
```

Adds an assumption to the top of the assumptions.

Given a theorem `th` of the form `A' |- u`, and a goal, `LAST_ASSUME_TAC th` adds `u` to the assumptions of the goal.

```
      A ?- t
==============  LAST_ASSUME_TAC (A' |- u)
 {u} u A ?- t
```

Note that unless `A'` is a subset of `A`, this tactic is invalid.

Failure

Never fails.

`LAST_ASSUME_TAC` is the naive way of manipulating assumptions (i.e. without recourse to advanced tacticals); and it is useful for enriching the assumption list with lemmas as a prelude to resolution (`RES_TAC`, `IMP_RES_TAC`), rewriting with assumptions (`ASM_REWRITE_TAC` and so on), and other operations involving assumptions.

See also

`Tactic.ASSUME_TAC`

LAST_CONV (listLib)

```
listLib.LAST_CONV : conv
```

Computes by inference the result of taking the last element of a list.

For any object language list of the form "`[x0;...x(n-1)]`", the result of evaluating

```
LAST_CONV "LAST [x0;...;x(n-1)]"
```

is the theorem

```
|- LAST [x0;...;x(n-1)] = x(n-1)
```

Failure

`LAST_CONV tm` fails if `tm` is an empty list.

LAST_EXISTS_CONV (Conv)

```
Conv.LAST_EXISTS_CONV : conv -> conv
```

Applies a conversion to the last existential quantifier (and its body) in a chain.

Application of `LAST_EXISTS_CONV c` to the term ``` ``?x1 .. xn x. body`` ``` will apply `c` to the term ``` ``?x. body`` ```. If the result of this application is the theorem `|- (?x. body) = t`, then the result of the whole will be

```
|- (?x1 .. xn x. body) = (?x1 .. xn. t)
```

Failure

Fails if the term is not existentially quantified, or if the conversion `c` fails when it is applied.

See also

`Conv.BINDER_CONV`, `Conv.LAST_FORALL_CONV`, `Conv.STRIP_QUANT_CONV`

LAST_FORALL_CONV (Conv)

```
Conv.LAST_FORALL_CONV : conv -> conv
```

Applies a conversion to the last universal quantifier (and its body) in a chain.

Application of `LAST_FORALL_CONV v` to the term ``` ``!x1 .. xn x. body`` ``` will apply `c` to the term ``` ``!x. body`` ```. If the result of this application is the theorem `|- (!x. body) = t`, then the result of the whole will be

```
|- (?x1 .. xn x. body) = (?x1 .. xn. t)
```

Failure

Fails if the term is not universally quantified, or if the conversion `c` fails when it is applied.

See also

`Conv.BINDER_CONV`, `Conv.LAST_EXISTS_CONV`, `Conv.STRIP_QUANT_CONV`

LAST_X_ASSUM (Tactical)

```
Tactical.LAST_X_ASSUM : thm_tactic -> tactic
```

Maps a theorem-tactic over the assumptions, in reverse order, applying first successful tactic and removing the assumption that gave rise to the successful tactic.

`LAST_X_ASSUM` behaves like `FIRST_X_ASSUM`, except that it goes through the list of assumptions in reverse order

See also

`Tactical.FIRST_X_ASSUM`, `Tactical.LAST_ASSUM`

LASTGOAL (Tactical)

```
Tactical.LASTGOAL : tactic -> list_tactic
```

The list-tactic which applies a tactic to the last member of a list of goals.

If `tac` is a tactic, `LASTGOAL tac` is a list-tactic which applies the tactic `tac` to the last member of a list of goals.

Failure

The application of `LASTGOAL` to a tactic never fails. The resulting list-tactic fails the goal list is empty or or finally if `tac` fails when applied to the last member of the goal list.

Applying a tactic to the last subgoal.

Example

Where `tac1` and `tac2` are tactics, `tac1 THEN_LT LASTGOAL tac2` applies `tac1` to a goal, and then applies `tac2` to the last resulting subgoal.

See also

`Tactical.THEN_LT`, `Tactical.NTH_GOAL`, `Tactical.HEADGOAL`

LASTN_CONV (listLib)

```
listLib.LASTN_CONV : conv
```

Computes by inference the result of taking the last n elements of a list.

For any object language list of the form "`[x0;...x(n-k);...;x(n-1)]`" , the result of evaluating

```
LASTN_CONV "LASTN k [x0;...x(n-k);...;x(n-1)]"
```

is the theorem

```
|- LASTN k [x0;...;x(n-k);...;x(n-1)] = [x(n-k);...;x(n-1)]
```

Failure

`LASTN_CONV tm` fails if `tm` is not of the form described above, or `k` is greater than the length of the list.

LE_CONV (reduceLib)

```
reduceLib.LE_CONV : conv
```

Proves result of less-than-or-equal-to ordering on two numerals.

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...), then `LE_CONV "m <= n"` returns the theorem:

```
|- (m <= n) = T
```

if the natural number denoted by `m` is less than or equal to that denoted by `n`, or

```
|- (m <= n) = F
```

otherwise.

Failure

`LE_CONV tm` fails unless `tm` is of the form `"m <= n"`, where `m` and `n` are numerals.

Example

```
#LE_CONV "12 <= 198";;
|- 12 <= 198 = T

#LE_CONV "46 <= 46";;
|- 46 <= 46 = T

#LE_CONV "13 <= 12";;
|- 13 <= 12 = F
```

LEAST_ELIM_TAC (numLib)

```
numLib.LEAST_ELIM_TAC : tactic
```

Eliminates a `LEAST` term from the current goal.

`LEAST_ELIM_TAC` searches the goal it is applied to for free sub-terms involving the `LEAST` operator, of the form `$LEAST P` (`P` will usually be an abstraction). If such a term is found, the tactic produces a new goal where instances of the `LEAST`-term have disappeared. The resulting goal will require the proof that there exists a value satisfying `P`, and that a minimal value satisfies the original goal.

Thus, `LEAST_ELIM_TAC` can be seen as a higher-order match against the theorem

```
|- !P Q.
      (?n. P x) /\ (!n. (!m. m < n ==> ~P m) /\ P n ==> Q n) ==>
      Q ($LEAST P)
```

where the new goal is the antecdent of the implication. (This theorem is `LEAST_ELIM`, from theory `while`.)

Failure

The tactic fails if there is no free `LEAST`-term in the goal.

Example

When applied to the goal

```
?- (LEAST n. 4 < n) = 5
```

the tactic LEAST_ELIM_TAC produces

```
?- (?n. 4 < n) /\ !n. (!m. m < n ==> ~(4 < m)) /\ 4 < n ==> (n = 5)
```

Comments

This tactic assumes that there is indeed a least number satisfying the given predicate. If there is not, then the LEAST-term will have an arbitrary value, and the proof should proceed by showing that the enclosing predicate Q holds for all possible numbers.

If there are multiple different LEAST-terms in the goal, then LEAST_ELIM_TAC will pick the first free LEAST-term returned by the standard find_terms function.

See also

Tactic.DEEP_INTRO_TAC, Tactic.SELECT_ELIM_TAC

LEFT_AND_EXISTS_CONV (Conv)

```
Conv.LEFT_AND_EXISTS_CONV : conv
```

Moves an existential quantification of the left conjunct outwards through a conjunction.

When applied to a term of the form (?x.P) /\ Q, the conversion LEFT_AND_EXISTS_CONV returns the theorem:

```
|- (?x.P) /\ Q = (?x'. P[x'/x] /\ Q)
```

where x' is a primed variant of x that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `(?x.P) /\ Q`.

See also

`Conv.AND_EXISTS_CONV`, `Conv.EXISTS_AND_CONV`, `Conv.RIGHT_AND_EXISTS_CONV`

LEFT_AND_FORALL_CONV (Conv)

```
Conv.LEFT_AND_FORALL_CONV : conv
```

Moves a universal quantification of the left conjunct outwards through a conjunction.

When applied to a term of the form `(!x.P) /\ Q`, the conversion `LEFT_AND_FORALL_CONV` returns the theorem:

```
|- (!x.P) /\ Q = (!x'. P[x'/x] /\ Q)
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `(!x.P) /\ Q`.

See also

`Conv.AND_FORALL_CONV`, `Conv.FORALL_AND_CONV`, `Conv.RIGHT_AND_FORALL_CONV`

LEFT_AND_PEXISTS_CONV (PairRules)

```
PairRules.LEFT_AND_PEXISTS_CONV : conv
```

Moves a paired existential quantification of the left conjunct outwards through a conjunction.

When applied to a term of the form `(?p. t) /\ u`, the conversion `LEFT_AND_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) /\ u = (?p'. t[p'/p] /\ u)
```

where `p'` is a primed variant of the pair `p` that does not contains variables free in the input term.

Failure

Fails if applied to a term not of the form `(?p. t) /\ u`.

See also

`Conv.LEFT_AND_EXISTS_CONV`, `PairRules.AND_PEXISTS_CONV`, `PairRules.PEXISTS_AND_CONV`, `PairRules.RIGHT_AND_PEXISTS_CONV`

LEFT_AND_PFORALL_CONV (PairRules)

```
PairRules.LEFT_AND_PFORALL_CONV : conv
```

Moves a paired universal quantification of the left conjunct outwards through a conjunction.

When applied to a term of the form `(!p. t) /\ u`, the conversion `LEFT_AND_PFORALL_CONV` returns the theorem:

```
|- (!p. t) /\ u = (!p'. t[p'/p] /\ u)
```

where `p'` is a primed variant of `p` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `(!p. t) /\ u`.

See also

Conv.LEFT_AND_FORALL_CONV, PairRules.AND_PFORALL_CONV, PairRules.PFORALL_-AND_CONV, PairRules.RIGHT_AND_PFORALL_CONV

LEFT_IMP_EXISTS_CONV (Conv)

```
Conv.LEFT_IMP_EXISTS_CONV : conv
```

Moves an existential quantification of the antecedent outwards through an implication.

When applied to a term of the form `(?x.P) ==> Q`, the conversion LEFT_IMP_EXISTS_-CONV returns the theorem:

```
|- (?x.P) ==> Q = (!x'. P[x'/x] ==> Q)
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `(?x.P) ==> Q`.

See also

Conv.FORALL_IMP_CONV, Conv.RIGHT_IMP_FORALL_CONV

LEFT_IMP_FORALL_CONV (Conv)

```
Conv.LEFT_IMP_FORALL_CONV : conv
```

Moves a universal quantification of the antecedent outwards through an implication.

When applied to a term of the form `(!x.P) ==> Q`, the conversion LEFT_IMP_FORALL_-CONV returns the theorem:

```
|- (!x.P) ==> Q = (?x'. P[x'/x] ==> Q)
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `(!x.P) ==> Q`.

See also

`Conv.EXISTS_IMP_CONV`, `Conv.RIGHT_IMP_FORALL_CONV`

LEFT_IMP_PEXISTS_CONV (PairRules)

`PairRules.LEFT_IMP_PEXISTS_CONV : conv`

Moves a paired existential quantification of the antecedent outwards through an implication.

When applied to a term of the form `(?p. t) ==> u`, the conversion `LEFT_IMP_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) ==> u = (!p'. t[p'/p] ==> u)
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure

Fails if applied to a term not of the form `(?p. t) ==> u`.

See also

`Conv.LEFT_IMP_EXISTS_CONV`, `PairRules.PFORALL_IMP_CONV`, `PairRules.RIGHT_IMP_PFORALL_CONV`

LEFT_IMP_PFORALL_CONV (PairRules)

```
PairRules.LEFT_IMP_PFORALL_CONV : conv
```

Moves a paired universal quantification of the antecedent outwards through an implication.

When applied to a term of the form `(!p. t) ==> u`, the conversion `LEFT_IMP_PFORALL_CONV` returns the theorem:

```
|- (!p. t) ==> u = (?p'. t[p'/p] ==> u)
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure

Fails if applied to a term not of the form `(!p. t) ==> u`.

See also

`Conv.LEFT_IMP_FORALL_CONV`, `PairRules.PEXISTS_IMP_CONV`, `PairRules.RIGHT_IMP_PFORALL_CONV`

LEFT_LIST_PBETA (PairRules)

```
PairRules.LEFT_LIST_PBETA : (thm -> thm)
```

Iteratively beta-reduces a top-level paired beta-redex on the left-hand side of an equation.

When applied to an equational theorem, `LEFT_LIST_PBETA` applies paired beta-reduction over a top-level chain of beta-redexes to the left-hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
 A |- (\p1...pn. t) q1 ... qn = s
---------------------------------  LEFT_LIST_BETA
    A |- t[q1/p1]...[qn/pn] = s
```

Failure

Fails unless the theorem is equational, with its left-hand side being a top-level paired beta-redex.

See also

Drule.RIGHT_LIST_BETA, PairRules.PBETA_CONV, PairRules.PBETA_RULE, PairRules.PBETA_TAC, PairRules.LIST_PBETA_CONV, PairRules.LEFT_PBETA, PairRules.RIGHT_PBETA, PairRules.RIGHT_LIST_PBETA

LEFT_OR_EXISTS_CONV (Conv)

```
Conv.LEFT_OR_EXISTS_CONV : conv
```

Moves an existential quantification of the left disjunct outwards through a disjunction.

When applied to a term of the form `(?x.P) \/ Q`, the conversion `LEFT_OR_EXISTS_CONV` returns the theorem:

```
|- (?x.P) \/ Q = (?x'. P[x'/x] \/ Q)
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `(?x.P) \/ Q`.

See also

Conv.EXISTS_OR_CONV, Conv.OR_EXISTS_CONV, Conv.RIGHT_OR_EXISTS_CONV

LEFT_OR_FORALL_CONV (Conv)

```
Conv.LEFT_OR_FORALL_CONV : conv
```

Moves a universal quantification of the left disjunct outwards through a disjunction.

When applied to a term of the form `(!x.P) \/ Q`, the conversion `LEFT_OR_FORALL_CONV` returns the theorem:

```
|- (!x.P) \/ Q = (!x'. P[x'/x] \/ Q)
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `(!x.P) \/ Q`.

See also

`Conv.OR_FORALL_CONV`, `Conv.FORALL_OR_CONV`, `Conv.RIGHT_OR_FORALL_CONV`

LEFT_OR_PEXISTS_CONV (PairRules)

`PairRules.LEFT_OR_PEXISTS_CONV : conv`

Moves a paired existential quantification of the left disjunct outwards through a disjunction.

When applied to a term of the form `(?p. t) \/ u`, the conversion `LEFT_OR_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) \/ u = (?p'. t[p'/p] \/ u)
```

where `p'` is a primed variant of the pair `p` that does not contain any variables free in the input term.

Failure

Fails if applied to a term not of the form `(?p. t) \/ u`.

See also

Conv.LEFT_OR_EXISTS_CONV, PairRules.PEXISTS_OR_CONV, PairRules.OR_PEXISTS_-CONV, PairRules.RIGHT_OR_PEXISTS_CONV

LEFT_OR_PFORALL_CONV (PairRules)

PairRules.LEFT_OR_PFORALL_CONV : conv

Moves a paired universal quantification of the left disjunct outwards through a disjunction.

When applied to a term of the form `(!p. t) \/ u`, the conversion LEFT_OR_FORALL_CONV returns the theorem:

```
|- (!p. t) \/ u = (!p'. t[p'/p] \/ u)
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure

Fails if applied to a term not of the form `(!p. t) \/ u`.

See also

Conv.LEFT_OR_FORALL_CONV, PairRules.OR_PFORALL_CONV, PairRules.PFORALL_OR_-CONV, PairRules.RIGHT_OR_PFORALL_CONV

LEFT_PBETA (PairRules)

PairRules.LEFT_PBETA : (thm -> thm)

Beta-reduces a top-level paired beta-redex on the left-hand side of an equation.

When applied to an equational theorem, LEFT_PBETA applies paired beta-reduction at top level to the left-hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
 A |- (\x. t1) t2 = s
---------------------  LEFT_PBETA
  A |- t1[t2/x] = s
```

Failure

Fails unless the theorem is equational, with its left-hand side being a top-level paired beta-redex.

See also

Drule.RIGHT_BETA, PairRules.PBETA_CONV, PairRules.PBETA_RULE, PairRules.PBETA_TAC, PairRules.RIGHT_PBETA, PairRules.RIGHT_LIST_PBETA, PairRules.LEFT_LIST_PBETA

LENGTH_CONV (listLib)

```
listLib.LENGTH_CONV : conv
```

Computes by inference the length of an object-language list.

For any object language list of the form "[x1;x2;...;xn]", where x1, x2, ..., xn are arbitrary terms of the same type, the result of evaluating

```
LENGTH_CONV "LENGTH [x1;x2;...;xn]"
```

is the theorem

```
|- LENGTH [x1;x2;...;xn] = n
```

where n is the numeral constant that denotes the length of the list.

Failure

`LENGTH_CONV tm` fails if `tm` is not of the form "`LENGTH [x1;x2;...;xn]`" or "`LENGTH []`".

LESS_CONV (wordsLib)

```
wordsLib.LESS_CONV : conv
```

Converts terms of the form `n < m` into `(n = m - 1) \/ ... \/ (n = 1) \/ (n = 0)`, provided that `m` is a natural number literal.

Example

```
> wordsLib.LESS_CONV "n < 4n";
val it = ⊢ n < 4 ⇔ n = 3 ∨ n = 2 ∨ n = 1 ∨ n = 0: thm
```

let_tm (boolSyntax)

```
boolSyntax.let_tm : term
```

Constant denoting `let` expressions.

The ML variable `boolSyntax.let_tm` is bound to the term `bool$LET`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.bool_case`, `boolSyntax.arb`

lhand (boolSyntax)

```
boolSyntax.lhand : term -> term
```

Returns the left-hand argument of a binary application.

A call to `lhand t` returns `x` in those situations where `t` is of the form ``` ``f x y`` ```.

Failure

Fails if the argument is not of the required form.

Example

```
> lhand ``3 + 2``;
val it = “3”: term
```

Comments

The name `lhand` is an abbreviation of “left-hand”, but `rand` is so-named as an abbreviation of “operand”. Nonetheless, `rand` does return the right-hand argument of a binary application.

See also

`Term.rand`, `Term.rator`

lhs (boolSyntax)

```
boolSyntax.lhs : term -> term
```

Returns the left-hand side of an equation.

If `M` has the form `t1 = t2` then `lhs M` returns `t1`.

Failure

Fails if the term is not an equation.

See also

`boolSyntax.rhs`, `boolSyntax.dest_eq`, `boolSyntax.mk_eq`

LHS_CONV (Conv)

```
Conv.LHS_CONV : conv -> conv
```

Applies a conversion to the left-hand argument of an equality.

If `c` is a conversion that maps a term `t1` to the theorem `|- t1 = t1'`, then the conversion `LHS_CONV c` maps applications of the form `t1 = t2` to theorems of the form:

```
|- (t1 = t2) = (t1' = t2)
```

Failure

`LHS_CONV c tm` fails if `tm` is not an an equality `t1 = t2`, or if `tm` has this form but the conversion `c` fails when applied to the term `t1`. The function returned by `LHS_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

Example

```
> LHS_CONV reduceLib.REDUCE_CONV (Term`(3 + 5) = 7`);
val it = ⊢ 3 + 5 = 7 ⇔ 8 = 7: thm
```

Comments

`LAND_CONV` is similar, but works for any binary operator

See also

`Conv.BINOP_CONV`, `Conv.RHS_CONV`, `numLib.REDUCE_CONV`, `Conv.LAND_CONV`

Lib (structure)

```
structure Lib
```

Collection of commonly used functions.

`Lib` is a collection of functions that have been found useful in writing the HOL system.

Comments

The SML Basis Library offers alternatives to some of the functions found in `Lib`.

line_name (unwindLib)

```
unwindLib.line_name : (term -> string)
```

Computes the line name of an equation.

`line_name "!y1 ... ym. f x1 ... xn = t"` returns the string `` `f` ``.

Failure

Fails if the argument term is not of the specified form.

See also

`unwindLib.line_var`

line_var (unwindLib)

```
unwindLib.line_var : (term -> term)
```

Computes the line variable of an equation.

`line_var "!y1 ... ym. f x1 ... xn = t"` returns the variable `"f"`.

Failure

Fails if the argument term is not of the specified form.

See also

`unwindLib.line_name`

LIST_BETA_CONV (Drule)

```
Drule.LIST_BETA_CONV : conv
```

Performs an iterated beta conversion.

The conversion `LIST_BETA_CONV` maps terms of the form

```
"(\x1 x2 ... xn. u) v1 v2 ... vn"
```

to the theorems of the form

```
|- (\x1 x2 ... xn. u) v1 v2 ... vn = u[v1/x1][v2/x2] ... [vn/xn]
```

where `u[vi/xi]` denotes the result of substituting `vi` for all free occurrences of `xi` in `u`, after renaming sufficient bound variables to avoid variable capture.

Failure

`LIST_BETA_CONV tm` fails if `tm` does not have the form `"(\x1 ... xn. u) v1 ... vn"` for `n` greater than 0.

Example

```
> LIST_BETA_CONV (Term `(\x y. x+y) 1 2`);
val it = ⊢ (λx y. x + y) 1 2 = 1 + 2: thm
```

See also

`Thm.BETA_CONV`, `Conv.BETA_RULE`, `Tactic.BETA_TAC`, `Drule.RIGHT_BETA`, `Drule.RIGHT_LIST_BETA`

list_compare (Lib)

```
Lib.list_compare : 'a cmp -> 'a list cmp
```

Lifts a comparison function to a lexicographic ordering on lists.

An application `list_compare comp (L1,L2)` uses `comp` as a basis for comparing the lists `L1` and `L2` lexicographically, in left-to-right order. The returned value is one of `{LESS, EQUAL, GREATER}`.

Failure

If `comp` fails when applied to corresponding elements of `L1` and `L2`.

Example

```
> list_compare Int.compare ([1,2,3,4], [1,2,3,4]);
val it = EQUAL: order

> list_compare Int.compare ([1,2,3,4], [1,2,3,4,5]);
val it = LESS: order
```

```
> list_compare Int.compare ([1,2,3,4], [1,2,3,2]);
val it = GREATER: order
```

LIST_CONJ (Drule)

```
Drule.LIST_CONJ : thm list -> thm
```

Conjoins the conclusions of a list of theorems.

```
        A1 |- t1 ... An |- tn
  ---------------------------------  LIST_CONJ
   A1 u ... u An |- t1 /\ ... /\ tn
```

Failure

`LIST_CONJ` fails if applied to an empty list of theorems.

See also

`Drule.BODY_CONJUNCTS`, `Thm.CONJ`, `Thm.CONJUNCT1`, `Thm.CONJUNCT2`, `Drule.CONJUNCTS`, `Drule.CONJ_PAIR`, `Tactic.CONJ_TAC`

list_FOLD_CONV (listLib)

```
listLib.list_FOLD_CONV : thm -> conv -> conv
```

Computes by inference the result of applying a function to the elements of a list.

Evaluating `list_FOLD_CONV fthm conv tm` returns a theorem

```
   |- CONST x0' ... xi' ... xn' = tm'
```

The first argument `fthm` should be a theorem of the form

```
|- !x0 ... xi ... xn. CONST x0 ... xi ... xn = FOLD[LR] f e xi
```

where `FOLD[LR]` means either `FOLDL` or `FOLDR`. The last argument `tm` is a term of the following form:

```
CONST x0' ... xi' ... xn'
```

where `xi'` is a concrete list. `list_FOLD_CONV` first instantiates the input theorem using `tm`. It then calls either `FOLDL_CONV` or `FOLDR_CONV` with the user supplied conversion `conv` on the right-hand side.

Failure

`list_FOLD_CONV fthm conv tm` fails if `fthm` or `tm` is not of the form described above, or if they do not agree, or the call to `FOLDL_CONV` OR `FOLDR_CONV` fails.

This function is used to implement conversions for logical constants which can be expressed in terms of the fold operators. For example, the constant `SUM` can be expressed in terms of `FOLDR` as in the following theorem:

```
|- !l. SUM l = FOLDR $+ 0 l
```

The conversion for `SUM`, `SUM_CONV` can be implemented as

```
load_library_in_place num_lib;
val SUM_CONV =
   list_FOLD_CONV (theorem "list" "SUM_FOLDR") Num_lib.ADD_CONV;
```

Then, evaluating `SUM_CONV` “`SUM [0;1;2;3]`” will return the following theorem:

```
|- SUM [0;1;2;3] = 6
```

See also

`listLib.FOLDL_CONV`, `listLib.FOLDR_CONV`

LIST_INDUCT_TAC (listLib)

```
listLib.LIST_INDUCT_TAC : tactic
```

Performs tactical proof by structural induction on lists.

`LIST_INDUCT_TAC` reduces a goal `!l.P[l]`, where `l` ranges over lists, to two subgoals corresponding to the base and step cases in a proof by structural induction on `l`. The induction hypothesis appears among the assumptions of the subgoal for the step case. The specification of `LIST_INDUCT_TAC` is:

```
                    A ?- !l. P
   ================================================  LIST_INDUCT_TAC
    A |- P[NIL/l]   A u {{P[l'/l]}} ?- !h. P[CONS h l'/l]
```

where `l'` is a primed variant of `l` that does not appear free in the assumptions `A` (usually, `l'` is just `l`). When `LIST_INDUCT_TAC` is applied to a goal of the form `!l.P`, where `l` does not appear free in `P`, the subgoals are just `A ?- P` and `A u {{P}} ?- !h.P`.

Failure

`LIST_INDUCT_TAC g` fails unless the conclusion of the goal `g` has the form `!l.t`, where the variable `l` has type `(ty)list` for some type `ty`.

See also

`listLib.EQ_LENGTH_INDUCT_TAC`, `listLib.EQ_LENGTH_SNOC_INDUCT_TAC`, `listLib.SNOC_INDUCT_TAC`

list_mk_abs (Term)

```
Term.list_mk_abs : term list * term -> term
```

Also exported as `boolSyntax.list_mk_abs`.

Performs a sequence of lambda binding operations.

An application `list_mk_abs ([v1,...,vn], M)` yields the term `\v1 ... vn. M`. Free occurrences of `v1,...,vn` in `M` become bound in the result.

Failure

Fails if some `vi` (1 <= i <= n) is not a variable.

Example

```
> list_mk_abs ([mk_var("v1",bool),mk_var("v2",bool),mk_var("v3",bool)],
               Term `v1 /\ v2 /\ v3`);
val it = “λv1 v2 v3. v1 ∧ v2 ∧ v3”: term
```

Comments

In the current implementation, `list_mk_abs` is more efficient than iteration of `mk_abs` for larger tasks.

See also

`Term.mk_abs`, `boolSyntax.list_mk_forall`, `boolSyntax.list_mk_exists`

list_mk_abs (boolSyntax)

```
boolSyntax.list_mk_abs : term list * term -> term
```

Re-exported from `Term.list_mk_abs`. See that entry for full documentation.

list_mk_anylet (pairSyntax)

```
pairSyntax.list_mk_anylet : (term * term) list list * term -> term
```

Construct arbitrary `let` terms.

The invocation `list_mk_anylet ([[(a1,b1),...,(an,bn)], ... [(u1,v1),...,(uk,vk)]],body` returns a term with surface syntax

```
let a1 = b1 and ... an = bn in
   ...                       in
let u1 = v1 and ... and uk = vk
 in body
```

Failure

If any binding pair `(x,y)` is such that `x` and `y` have different types.

Example

```
list_mk_anylet
  ([[(``x:'a``, ``P:'a``)],
    [(``(y:'a, z:ind)``, ``M:'a#ind``)],
    [(``f (x:'a):bool``, ``N:bool``),
     (``g:bool->'a``,    ``K (v:'a):bool->'a``)]], ``g (f (x:'a):bool):'a``);
> val it = `let x = P in
            let (y,z) = M in
            let f x = N
            and g = K v
            in g (f x)`
val it =
   [QUOTE
      " (*#loc 1 11*)let x = P in\n         let (y,z) = M in\n         let f x = N\
   'a frag list
```

Programming that involves manipulation of term syntax.

See also

`boolSyntax.dest_let`, `pairSyntax.mk_anylet`, `pairSyntax.strip_anylet`, `pairSyntax.dest_-anylet`

list_mk_binder (Term)

```
Term.list_mk_binder : term option -> term list * term -> term
```

Performs a sequence of variable binding operations on a term.

An application `list_mk_binder (SOME c) ([v1,...,vn],M)` builds the term `c (\v1. ... (c (\vn. M) ...))`. The term `c` should be a binder, that is, a constant that takes a lambda abstraction and returns a bound term. Thus `list_mk_binder` implements Church's view that variable binding operations should be reduced to lambda-binding.

An application `list_mk_binder NONE ([v1,...,vn],M)` builds the term `\v1...vn. M`.

Failure

`list_mk_binder opt ([v1,...,vn],M)` fails if some `vi 1 <= i <= n` is not a variable. It also fails if the constructed term `c (\v1. ... (c (\vn. M) ...))` is not well typed.

Example

Repeated existential quantification is easy to code up using `list_mk_binder`. For testing, we make a list of boolean variables.

```
- fun upto b t acc = if b >= t then rev acc else upto (b+1) t (b::acc)

  fun vlist n = map (C (curry mk_var) bool o concat "v" o int_to_string)
                    (upto 0 n []);
  val vars = vlist 100;

> val vars =
 [`v0`, `v1`, `v2`, `v3`, `v4`, `v5`, `v6`, `v7`, `v8`, `v9`, `v10`, `v11`,
  `v12`, `v13`, `v14`, `v15`, `v16`, `v17`, `v18`, `v19`, `v20`, `v21`,
  `v22`, `v23`, `v24`, `v25`, `v26`, `v27`, `v28`, `v29`, `v30`, `v31`,
  `v32`, `v33`, `v34`, `v35`, `v36`, `v37`, `v38`, `v39`, `v40`, `v41`,
  `v42`, `v43`, `v44`, `v45`, `v46`, `v47`, `v48`, `v49`, `v50`, `v51`,
  `v52`, `v53`, `v54`, `v55`, `v56`, `v57`, `v58`, `v59`, `v60`, `v61`,
  `v62`, `v63`, `v64`, `v65`, `v66`, `v67`, `v68`, `v69`, `v70`, `v71`,
  `v72`, `v73`, `v74`, `v75`, `v76`, `v77`, `v78`, `v79`, `v80`, `v81`,
```

```
  `v82`, `v83`, `v84`, `v85`, `v86`, `v87`, `v88`, `v89`, `v90`, `v91`,
  `v92`, `v93`, `v94`, `v95`, `v96`, `v97`, `v98`, `v99`] : term list
```

Now we exercise `list_mk_binder`.

```
- val exl_tm = list_mk_binder (SOME boolSyntax.existential)
                              (vars, list_mk_conj vars);
> val exl_tm =
 `?v0 v1 v2 v3 v4 v5 v6 v7 v8 v9 v10 v11 v12 v13 v14 v15 v16 v17 v18 v19 v20
   v21 v22 v23 v24 v25 v26 v27 v28 v29 v30 v31 v32 v33 v34 v35 v36 v37 v38
   v39 v40 v41 v42 v43 v44 v45 v46 v47 v48 v49 v50 v51 v52 v53 v54 v55 v56
   v57 v58 v59 v60 v61 v62 v63 v64 v65 v66 v67 v68 v69 v70 v71 v72 v73 v74
   v75 v76 v77 v78 v79 v80 v81 v82 v83 v84 v85 v86 v87 v88 v89 v90 v91 v92
   v93 v94 v95 v96 v97 v98 v99.
    v0 /\ v1 /\ v2 /\ v3 /\ v4 /\ v5 /\ v6 /\ v7 /\ v8 /\ v9 /\ v10 /\
    v11 /\ v12 /\ v13 /\ v14 /\ v15 /\ v16 /\ v17 /\ v18 /\ v19 /\ v20 /\
    v21 /\ v22 /\ v23 /\ v24 /\ v25 /\ v26 /\ v27 /\ v28 /\ v29 /\ v30 /\
    v31 /\ v32 /\ v33 /\ v34 /\ v35 /\ v36 /\ v37 /\ v38 /\ v39 /\ v40 /\
    v41 /\ v42 /\ v43 /\ v44 /\ v45 /\ v46 /\ v47 /\ v48 /\ v49 /\ v50 /\
    v51 /\ v52 /\ v53 /\ v54 /\ v55 /\ v56 /\ v57 /\ v58 /\ v59 /\ v60 /\
    v61 /\ v62 /\ v63 /\ v64 /\ v65 /\ v66 /\ v67 /\ v68 /\ v69 /\ v70 /\
    v71 /\ v72 /\ v73 /\ v74 /\ v75 /\ v76 /\ v77 /\ v78 /\ v79 /\ v80 /\
    v81 /\ v82 /\ v83 /\ v84 /\ v85 /\ v86 /\ v87 /\ v88 /\ v89 /\ v90 /\
    v91 /\ v92 /\ v93 /\ v94 /\ v95 /\ v96 /\ v97 /\ v98 /\ v99` : term
```

Comments

Terms with many consecutive binders should be constructed using `list_mk_binder` and its instantiations `list_mk_abs`, `list_mk_forall`, and `list_mk_exists`. In the current implementation of HOL, iterating `mk_abs`, `mk_forall`, or `mk_exists` is far slower for terms with many consecutive binders.

See also

`Term.list_mk_abs`, `boolSyntax.list_mk_forall`, `boolSyntax.list_mk_exists`, `Term.strip_binder`

list_mk_comb (Term)

```
Term.list_mk_comb : term * term list -> term
```

Iteratively constructs combinations (function applications).

`list_mk_comb(t,[t1,...,tn])` returns `t t1 ... tn`.

Failure

Fails if the types of `t1,...,tn` are not equal to the argument types of `t`. It is not necessary for all the arguments of `t` to be given. In particular the list of terms `t1,...,tn` may be empty.

Example

```
> list_mk_comb(conditional,[T, mk_var("one",alpha), mk_var("two",alpha)]);
val it = “if T then one else two”: term

> list_mk_comb(universal,[]);
val it = “$!”: term

> try list_mk_comb(universal,[F]);
Exception- Type error in function application.
   Function: try list_mk_comb :
      (term -> (term * term list) option) list -> term -> term
   Argument: (universal, [F]) : term * term list
   Reason:
      Can't unify (term -> (term * term list) option) list to
         term * term list (Incompatible types)
Fail "Static Errors" raised
```

See also

`boolSyntax.strip_comb`, `Term.mk_comb`

list_mk_conj (boolSyntax)

```
boolSyntax.list_mk_conj : term list -> term
```

Constructs the conjunction of a list of terms.

`list_mk_conj([t1,...,tn])` returns `t1 /\ ... /\ tn`.

Failure

Fails if the list is empty or if the list has more than one element, one or more of which are not of type `bool`.

Example

```
> list_mk_conj [T,F,T];
val it = “T ∧ F ∧ T”: term

> try list_mk_conj [T,mk_var("x",alpha),F];
Exception- HOL_ERR (at boolSyntax.mk_conj: Non-boolean argument) raised

> list_mk_conj [mk_var("x",alpha)];
val it = “x”: term
```

See also

`boolSyntax.strip_conj`, `boolSyntax.mk_conj`

list_mk_disj (boolSyntax)

```
boolSyntax.list_mk_disj : term list -> term
```

Constructs the disjunction of a list of terms.

`list_mk_disj([t1,...,tn])` returns `t1 \/ ... \/ tn`.

Failure

Fails if the list is empty or if the list has more than one element, one or more of which are not of type `bool`.

Example

```
> list_mk_disj [T,F,T];
val it = “T ∨ F ∨ T”: term

> try list_mk_disj [T,mk_var("x",alpha),F];
Exception- HOL_ERR (at boolSyntax.mk_disj: Non-boolean argument) raised

> list_mk_disj [mk_var("x",alpha)];
val it = “x”: term
```

See also

`boolSyntax.strip_disj`, `boolSyntax.mk_disj`

LIST_MK_EXISTS (Drule)

```
Drule.LIST_MK_EXISTS : (term list -> thm -> thm)
```

Multiply existentially quantifies both sides of an equation using the given variables.

When applied to a list of terms `[x1;...;xn]`, where the `xi` are all variables, and a theorem `A |- t1 = t2`, the inference rule `LIST_MK_EXISTS` existentially quantifies both sides of the equation using the variables given, none of which should be free in the assumption list.

```
              A |- t1 = t2
------------------------------------  LIST_MK_EXISTS ["x1";...;"xn"]
 A |- (?x1...xn. t1) = (?x1...xn. t2)
```

Failure

Fails if any term in the list is not a variable or is free in the assumption list, or if the theorem is not equational.

See also

`Drule.EXISTS_EQ`, `Drule.MK_EXISTS`

list_mk_exists (boolSyntax)

```
boolSyntax.list_mk_exists : term list * term -> term
```

Iteratively constructs an existential quantification.

`list_mk_exists([x1,...,xn],t)` returns `?x1 ... xn. t`.

Failure

Fails if the terms in the list are not variables or if `t` is not of type `bool` and the list of terms is non-empty. If the list of terms is empty the type of `t` can be anything.

See also

`boolSyntax.strip_exists`, `boolSyntax.mk_exists`

list_mk_forall (boolSyntax)

```
boolSyntax.list_mk_forall : term list * term -> term
```

Iteratively constructs a universal quantification.

`list_mk_forall([x1,...,xn],t)` returns `!x1 ... xn. t`.

Failure

Fails if the terms in the list are not variables or if `t` is not of type `bool` and the list of terms is non-empty. If the list of terms is empty the type of `t` can be anything.

See also

`boolSyntax.strip_forall`, `boolSyntax.mk_forall`

list_mk_fun (boolSyntax)

```
boolSyntax.list_mk_fun : hol_type list * hol_type -> hol_type
```

Iteratively constructs function types.

`list_mk_fun([ty1,...,tyn],ty)` returns `ty1 -> ( ... (tyn -> t)...)`.

Failure

Never fails.

Example

```
> list_mk_fun ([alpha,bool],beta);
val it = ":α -> bool -> β": hol_type
```

See also

`boolSyntax.strip_fun`, `Type.mk_type`, `Type.-->`

list_mk_icomb (boolSyntax)

```
boolSyntax.list_mk_icomb : term * term list -> term
```

Folds mk_icomb over a series of arguments.

A call to list_mk_icomb(f,args) combines f with each of the elements of the list args in turn, moving from left to right. If args is empty, then the result is simply f. When args is non-empty, the growing application-term is created with successive calls to mk_icomb, possibly causing type variables in any of the terms to become instantiated.

Failure

Fails if any of the underlying calls to mk_icomb fails, which will occur if the type of the accumulating term (starting with f) is not of a function type, or if it has a domain type that can not be instantiated to equal the type of the next argument term.

Comments

list_mk_icomb is to mk_icomb what list_mk_comb is to mk_comb. However, it is important to be aware that list_mk_icomb instantiates types sequentially, and not, as one might expect, in parallel. For example the pairing function (,) is usually written infix and has type :'a -> 'b -> 'a # 'b. A pair where the first component has type :'b and the second has type :num can be built with

```
> pairSyntax.mk_pair(``x:'b``, ``y:num``);
val it = “(x,y)”: term
> type_of it;
val it = “:β # num”: hol_type
```

but an attempt to do the same via list_mk_icomb

```
> list_mk_icomb
    (``(,):'a -> 'b -> 'a # 'b``, [``x:'b``, ``y:num``]);
val it = “(x,y)”: term
> type_of it;
val it = “:num # num”: hol_type
```

confounds expectations since the call first instantiates :'a in the type of (,) by :'b, then instantiates :'b in the result by :num.

See also

Term.list_mk_comb, Term.mk_comb, boolSyntax.mk_icomb

list_mk_imp (boolSyntax)

```
boolSyntax.list_mk_imp : term list * term -> term
```

Iteratively constructs implications.

`list_mk_imp([t1,...,tn],t)` returns `t1 ==> ( ... (tn ==> t)...)`.

Failure

Fails if any of `t1,...,tn` are not of type `bool`. Also fails if the list of terms is non-empty and `t` is not of type `bool`. If the list of terms is empty the type of `t` can be anything.

Example

```
> list_mk_imp ([T,F],T);
val it = “T ⇒ F ⇒ T”: term

> try list_mk_imp ([T,F],mk_var("x",alpha));
Exception- HOL_ERR (at boolSyntax.mk_imp: Non-boolean argument) raised

> list_mk_imp ([],mk_var("x",alpha));
val it = “x”: term
```

See also

`boolSyntax.strip_imp`, `boolSyntax.mk_imp`

list_mk_pabs (pairSyntax)

```
pairSyntax.list_mk_pabs : term list * term -> term
```

Iteratively constructs paired abstractions.

`list_mk_pabs([p1,...,pn], t)` returns `\p1 ... pn. t`.

Failure

Fails with `list_mk_pabs` if the terms in the list are not paired structures of variables.

See also

`boolSyntax.list_mk_abs`, `pairSyntax.strip_pabs`, `pairSyntax.mk_pabs`

list_mk_pair (pairSyntax)

```
pairSyntax.list_mk_pair : term list -> term
```

Constructs a tuple from a list of terms.

`list_mk_pair([t1,...,tn])` returns the term `(t1,...,tn)`.

Failure

Fails if the list is empty.

Example

```
> pairSyntax.list_mk_pair [Term `1`, T, Term `2`];
val it = “(1,T,2)”: term

> pairSyntax.list_mk_pair [Term `1`];
val it = “1”: term
```

See also

`pairSyntax.strip_pair`, `pairSyntax.mk_pair`

LIST_MK_PEXISTS (PairRules)

```
PairRules.LIST_MK_PEXISTS : (term list -> thm -> thm)
```

Multiply existentially quantifies both sides of an equation using the given pairs.

When applied to a list of terms `[p1;...;pn]`, where the `pi` are all paired structures of variables, and a theorem `A |- t1 = t2`, the inference rule `LIST_MK_PEXISTS` existentially quantifies both sides of the equation using the pairs given, none of the variables in the pairs should be free in the assumption list.

```
              A |- t1 = t2
------------------------------------  LIST_MK_PEXISTS ["x1";...;"xn"]
 A |- (?x1...xn. t1) = (?x1...xn. t2)
```

Failure

Fails if any term in the list is not a paired structure of variables, or if any variable is free in the assumption list, or if the theorem is not equational.

See also

`Drule.LIST_MK_EXISTS`, `PairRules.PEXISTS_EQ`, `PairRules.MK_PEXISTS`

LIST_MK_PFORALL (PairRules)

```
PairRules.LIST_MK_PFORALL : (term list -> thm -> thm)
```

Multiply universally quantifies both sides of an equation using the given pairs.

When applied to a list of terms `[p1;...;pn]`, where the `pi` are all paired structures of variables, and a theorem `A |- t1 = t2`, the inference rule `LIST_MK_PFORALL` universally quantifies both sides of the equation using the pairs given, none of the variables in the pairs should be free in the assumption list.

```
              A |- t1 = t2
------------------------------------  LIST_MK_PFORALL ["x1";...;"xn"]
 A |- (!x1...xn. t1) = (!x1...xn. t2)
```

Failure

Fails if any term in the list is not a paired structure of variables, or if any variable is free in the assumption list, or if the theorem is not equational.

See also

`Drule.LIST_MK_EXISTS`, `PairRules.PFORALL_EQ`, `PairRules.MK_PFORALL`

list_mk_res_exists (res_quanLib)

```
res_quanLib.list_mk_res_exists : ((term # term) list # term) -> term)
```

Iteratively constructs a restricted existential quantification.

```
   list_mk_res_exists([("x1","P1");...;("xn","Pn")],"t")
```

returns `"?x1::P1. ... ?xn::Pn. t"`.

Failure

Fails with `list_mk_res_exists` if the first terms `xi` in the pairs are not a variable or if the second terms `Pi` in the pairs and `t` are not of type `":bool"` if the list is non-empty. If the list is empty the type of `t` can be anything.

See also

`res_quanLib.strip_res_exists`, `res_quanLib.mk_res_exists`

list_mk_res_forall (res_quanLib)

```
res_quanLib.list_mk_res_forall : (term # term) list # term) -> term
```

Iteratively constructs a restricted universal quantification.

```
list_mk_res_forall([("x1","P1");...;("xn","Pn")],"t")
```

returns `"!x1::P1. ... !xn::Pn. t"`.

Failure

Fails with `list_mk_res_forall` if the first terms `xi` in the pairs are not a variable or if the second terms `Pi` in the pairs and `t` are not of type `":bool"` if the list is non-empty. If the list is empty the type of `t` can be anything.

See also

`res_quanLib.strip_res_forall`, `res_quanLib.mk_res_forall`

list_mk_ucomb (boolSyntax)

```
boolSyntax.list_mk_ucomb : term * term list -> term
```

Folds `mk_ucomb` over a series of arguments.

A call to `list_mk_ucomb(f, args)` combines `f` with each of the elements of the list `args` in turn, moving from left to right. If `args` is empty, then the result is simply `f`. When `args` is non-empty, the growing application-term is created with successive calls to `mk_ucomb`, possibly causing type variables in any of the terms to become instantiated.

Failure

Fails if any of the underlying calls to `mk_ucomb` fails, which will occur if the type of the accumulating term (starting with `f`) is not of a function type, or if it has a domain type that can not be instantiated to equal the type of some instantiation of the next argument term.

Comments

`list_mk_ucomb` is to `mk_ucomb` what `list_mk_comb` is to `mk_comb`.

See also

`Term.list_mk_comb`, `Term.mk_comb`, `boolSyntax.mk_ucomb`

LIST_MP (Drule)

```
Drule.LIST_MP : thm list -> thm -> thm
```

Performs a chain of Modus Ponens inferences.

When applied to theorems `A1 |- t1, ..., An |- tn` and a theorem which is a chain of implications with the successive antecedents the same as the conclusions of the theorems in the list (up to alpha-conversion), `A |- t1 ==> ... ==> tn ==> t`, the `LIST_MP` inference rule performs a chain of `MP` inferences to deduce `A u A1 u ... u An |- t`.

```
 A1 |- t1 ... An |- tn      A |- t1 ==> ... ==> tn ==> t
-------------------------------------------------------  LIST_MP
                A u A1 u ... u An |- t
```

Failure

Fails unless the theorem is a chain of implications whose consequents are the same as the conclusions of the list of theorems (up to alpha-conversion), in sequence.

See also

`Thm.EQ_MP`, `Drule.MATCH_MP`, `Tactic.MATCH_MP_TAC`, `Thm.MP`, `Tactic.MP_TAC`

LIST_PBETA_CONV (PairRules)

```
PairRules.LIST_PBETA_CONV : conv
```

Performs an iterated paired beta-conversion.

The conversion `LIST_PBETA_CONV` maps terms of the form

```
(\p1 p2 ... pn. t) q1 q2 ... qn
```

to the theorems of the form

```
|- (\p1 p2 ... pn. t) q1 q2 ... qn = t[q1/p1][q2/p2] ... [qn/pn]
```

where `t[qi/pi]` denotes the result of substituting `qi` for all free occurrences of `pi` in `t`, after renaming sufficient bound variables to avoid variable capture.

Failure

`LIST_PBETA_CONV tm` fails if `tm` does not have the form `(\p1 ... pn. t) q1 ... qn` for `n` greater than 0.

Example

```
> PairRules.LIST_PBETA_CONV (Term `(\(a,b) (c,d) . a + b + c + d) (1,2) (3,4)`);
val it = ⊢ (λ(a,b) (c,d). a + b + c + d) (1,2) (3,4) = 1 + 2 + 3 + 4: thm
```

See also

`Drule.LIST_BETA_CONV`, `PairRules.PBETA_CONV`, `Conv.BETA_RULE`, `Tactic.BETA_TAC`, `PairRules.RIGHT_PBETA`, `PairRules.RIGHT_LIST_PBETA`, `PairRules.LEFT_PBETA`, `PairRules.LEFT_LIST_PBETA`

LIST_REFINE_EXISTS_TAC (Q)

```
Q.LIST_REFINE_EXISTS_TAC : term quotation list -> tactic
```

Attacks existential goals, making existential variables more concrete.

Q.LIST_REFINE_EXISTS_TAC is similar to Q.REFINE_EXISTS_TAC, except it accepts a list of quotations rather than a single one. It further skips quotations of a single variable beginning with an underscore, permitting straightforward refinement of nested existentials.

Note that quotations are parsed right-to-left, so earlier quotations are parsed in the context of later ones.

Failure

Fails if passed too many quotations for the current goal. Otherwise fails similarly to Q.REFINE_EXISTS_TAC.

Example

```
- Q.LIST_REFINE_EXISTS_TAC [`_`,`SUC c`,`_`,`n + m`]
    ([``n = 2``,``c = 5``], ``∃a b c d. a + b = c + d``);
> val it =
     ([([``n = 2``, ``c = 5``], ``∃a c' m. a + SUC c = c' + (n + m)``)], fn)
     : goal list * validation
```

Like Q.REFINE_EXISTS_TAC, Q.LIST_REFINE_EXISTS_TAC is useful if it is clear that an existential goal can be solved by a term of particular form, but it is not yet clear exactly what this term will be.

It is particularly useful when refining deeply-nested existentials, or many existentials at once.

Comments

This tactic is also available under the alias bossLib.qrefinel.

See also

Q.REFINE_EXISTS_TAC, Tactic.EXISTS_TAC

list_ss (bossLib)

bossLib.list_ss : simpset

Simplification set for lists.

The simplification set `list_ss` is a version of `arith_ss` enhanced for the theory of lists. The following rewrites are currently used to augment those already present from `arith_ss`:

```
|- (!l. APPEND [] l = l) /\
    !l1 l2 h. APPEND (h::l1) l2 = h::APPEND l1 l2
|- (!l1 l2 l3. (APPEND l1 l2 = APPEND l1 l3) = (l2 = l3)) /\
    !l1 l2 l3. (APPEND l2 l1 = APPEND l3 l1) = (l2 = l3)
|- (!l. EL 0 l = HD l) /\ !l n. EL (SUC n) l = EL n (TL l)
|- (!P. EVERY P [] = T) /\ !P h t. EVERY P (h::t) = P h /\ EVERY P t
|- (FLAT [] = []) /\ !h t. FLAT (h::t) = APPEND h (FLAT t)
|- (LENGTH [] = 0) /\ !h t. LENGTH (h::t) = SUC (LENGTH t)
|- (!f. MAP f [] = []) /\ !f h t. MAP f (h::t) = f h::MAP f t
|- (!f. MAP2 f [] [] = []) /\
    !f h1 t1 h2 t2.
       MAP2 f (h1::t1) (h2::t2) = f h1 h2::MAP2 f t1 t2
|- (!x. MEM x [] = F) /\ !x h t. MEM x (h::t) = (x = h) \/ MEM x t
|- (NULL [] = T) /\ !h t. NULL (h::t) = F
|- (REVERSE [] = []) /\ !h t. REVERSE (h::t) = APPEND (REVERSE t) [h]
|- (SUM [] = 0) /\ !h t. SUM (h::t) = h + SUM t
|- !h t. HD (h::t) = h
|- !h t. TL (h::t) = t
|- !l1 l2 l3. APPEND l1 (APPEND l2 l3) = APPEND (APPEND l1 l2) l3
|- !l. ~NULL l ==> (HD l::TL l = l)
|- !a0 a1 a0' a1'. (a0::a1 = a0'::a1') = (a0 = a0') /\ (a1 = a1')
|- !l1 l2. LENGTH (APPEND l1 l2) = LENGTH l1 + LENGTH l2
|- !l f. LENGTH (MAP f l) = LENGTH l
|- !f l1 l2. MAP f (APPEND l1 l2) = APPEND (MAP f l1) (MAP f l2)
|- !a1 a0. ~(a0::a1 = [])
|- !a1 a0. ~([] = a0::a1)
|- !l f. ((MAP f l = []) = (l = [])) /\
         (([] = MAP f l) = (l = []))
|- !l. APPEND l [] = l
|- !l x. ~(l = x::l) /\ ~(x::l = l)
|- (!v f. case v f [] = v) /\
    !v f a0 a1. case v f (a0::a1) = f a0 a1
|- (!l1 l2. ([] = APPEND l1 l2) = (l1 = []) /\ (l2 = [])) /\
```

```
    !l1 l2. (APPEND l1 l2 = []) = (l1 = []) /\ (l2 = [])
|- (ZIP ([][]) = []) /\
    !x1 l1 x2 l2. ZIP (x1::l1,x2::l2) = (x1,x2)::ZIP (l1,l2)
|- (UNZIP [] = ([],[])) /\
    !x l. UNZIP (x::l) = (FST x::FST (UNZIP l),SND x::SND (UNZIP l))
|- !P l1 l2. EVERY P (APPEND l1 l2) = EVERY P l1 /\ EVERY P l2
|- !P l1 l2. EXISTS P (APPEND l1 l2) = EXISTS P l1 \/ EXISTS P l2
|- !e l1 l2. MEM e (APPEND l1 l2) = MEM e l1 \/ MEM e l2
|- (!x. LAST [x] = x) /\ !x y z. LAST (x::y::z) = LAST (y::z)
|- (!x. FRONT [x] = []) /\ !x y z. FRONT (x::y::z) = x::FRONT (y::z)
|- (!f e. FOLDL f e [] = e) /\
    !f e x l. FOLDL f e (x::l) = FOLDL f (f e x) l
|- (!f e. FOLDR f e [] = e) /\
    !f e x l. FOLDR f e (x::l) = f x (FOLDR f e l)
```

See also

BasicProvers.RW_TAC, BasicProvers.SRW_TAC, simpLib.SIMP_TAC, simpLib.SIMP_CONV, simpLib.SIMP_RULE, BasicProvers.bool_ss, bossLib.std_ss, bossLib.arith_ss

listDB (DB)

```
DB.listDB : unit -> data list
```

All theorems, axioms, and definitions in the currently loaded theory segments.

An invocation `listDB()` returns everything that has been stored in all theory segments currently loaded.

Example

```
> length (listDB());
val it = 5489: int
```

See also

`DB.thy`, `DB.theorems`, `DB.definitions`, `DB.axioms`, `DB.find`, `DB.match`

listItems (computeLib)

```
computeLib.listItems : compset -> ((string * string) * transform list) list
```

List elements in compset

The function `listItems` expects a simplification set and returns a listing of its elements, in the form of an association list mapping constant names to the transformations that can be performed on applications of that constant. For a given constant, more than one transformation can be attached.

Example

```
> val compset = computeLib.bool_compset
val compset = <compset> : computeLib.compset

> computeLib.listItems compset;
val it =
   [(("/\\", "bool"),
     [RRules
       [|- $/\ F = (\t. F),
        |- $/\ T = (\t. t)],
      RRules
       [|- !t. t /\ t <=> t,
        |- !t. t /\ F <=> F,
        |- !t. t /\ T <=> t]]),
    (("=", "min"),
     [RRules
       [|- $<=> F = (\t. ~t),
        |- $<=> T = (\t. t)],
      RRules
       [|- !x. (x = x) <=> T],
      RRules
```

```
      [|- !t. (t <=> F) <=> ~t,
       |- !t. (t <=> T) <=> t]]),
   (("==>", "min"),
    [RRules
      [|- $==> F = (\t. T),
       |- $==> T = (\t. t)],
     RRules
      [|- !t. t ==> F <=> ~t,
       |- !t. t ==> t <=> T,
       |- !t. t ==> T <=> T]]),
   (("COND", "bool"),
    [RRules
      [|- COND F = (\t1 t2. t2),
       |- COND T = (\t1 t2. t1)],
     RRules
      [|- !t b. (if b then t else t) = t]]),
   (("F", "bool"), []),
   (("LET", "bool"),
    [RRules
      [|- !x f. LET f x = f x]]),
   (("T", "bool"), []),
   (("\\/", "bool"),
    [RRules
      [|- $\/ F = (\t. t),
       |- $\/ T = (\t. T)],
     RRules
      [|- !t. t \/ t <=> t,
       |- !t. t \/ F <=> t,
       |- !t. t \/ T <=> T]]),
   (("literal_case", "bool"),
    [RRules
      [|- !x f. literal_case f x = f x]]),
   (("~", "bool"),
    [RRules
      [|- ~F <=> T,
       |- ~T <=> F,
       |- !t. ~~t <=> t]])]
: ((string * string) * transform list) list
```

Failure

Should never fail.

See also

`computeLib.bool_compset`, `bossLib.EVAL`, `computeLib.transform`

LT_CONV (reduceLib)

`reduceLib.LT_CONV : conv`

Proves result of less-than ordering on two numerals.

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...) of type `:num`, then `LT_CONV ``m < n``` returns the theorem:

```
|- (m < n) = T
```

if the natural number denoted by `m` is less than that denoted by `n`, or

```
|- (m < n) = F
```

otherwise.

Failure

`LT_CONV tm` fails unless `tm` is of the form ``m < n``, where `m` and `n` are numerals of natural number type (`:num`).

Example

```
> LT_CONV ``0 < 12``;
val it = |- 0 < 12 <=> T : thm

> LT_CONV ``13 < 13``;
```

```
val it = |- 13 < 13 <=> F : thm

> LT_CONV ``25 < 12``;
val it = |- 25 < 12 <=> F : thm
```

See also

reduceLib.GT_CONV

map2 (Lib)

```
Lib.map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

Maps a function over two lists to create one new list.

`map2 f [x1,...,xn] [y1,...,yn]` returns `[f x1 y1,...,f xn yn]`.

Failure

Fails if the two lists are of different lengths. Also fails if any `f xi yi` fails.

Example

```
> map2 (curry op+) [1,2,3] [3,2,1];
val it = [4, 4, 4]: int list
```

See also

Lib.itlist, Lib.rev_itlist, Lib.itlist2, Lib.rev_itlist2

MAP2_CONV (listLib)

```
listLib.MAP2_CONV : conv -> conv
```

Compute the result of mapping a binary function down two lists.

The function MAP2_CONV is a conversion for computing the result of mapping a binary function f:ty1->ty2->ty3 down two lists "[l11;...;l1n]" whose elements are of type ty1 and "[l21;...;l2n]" whose elements are of type ty2. The lengths of the two lists must be identical. The first argument to MAP2_CONV is expected to be a conversion that computes the result of applying the function f to a pair of corresponding elements of these lists. When applied to a term "f l1i l2i", this conversion should return a theorem of the form |- (f l1i l2i) = ri, where ri is the result of applying the function f to the elements l1i and l2i.

Given an appropriate conv, the conversion MAP2_CONV conv takes a term of the form "MAP2 f [l11;...;dl2tn] [l21;...;l2n]" and returns the theorem

```
|- MAP2 f [l11;...;l1n] [l21;...;l2n] = [r1;...;rn]
```

where conv "f l1i l2i" returns |- (f l1i l2i) = ri for i from 1 to n.

Example

The following is a very simple example in which the corresponding elements from the two lists are summed to form the resulting list:

```
- load_library_in_place num_lib;
- MAP2_CONV Num_lib.ADD_CONV "MAP2 $+ [1;2;3] [1;2;3]";
|- MAP2 $+ [1;2;3] [1;2;3] = [2;4;6]
```

Failure

MAP2_CONV conv fails if applied to a term not of the form described above. An application of MAP2_CONV conv to a term "MAP2 f [l11;...;l1n] [l21;...;l2n]" fails unless for all i where 1<=i<=n evaluating conv "f l1i l2i" returns |- (f l1i l2i) = ri for some ri.

See also

listLib.MAP_CONV

MAP_CONV (listLib)

```
listLib.MAP_CONV : conv -> conv
```

Compute the result of mapping a function down a list.

The function MAP_CONV is a parameterized conversion for computing the result of mapping a function f:ty1->ty2 down a list "[t1;...;tn]" of elements of type ty1. The first argument to MAP_CONV is expected to be a conversion that computes the result of applying the function f to an element of this list. When applied to a term "f ti", this conversion should return a theorem of the form |- (f ti) = ri, where ri is the result of applying the function f to the element ti.

Given an appropriate conv, the conversion MAP_CONV conv takes a term of the form "MAP f [t1;...;tn]" to the theorem

```
|- MAP f [t1;...;tn] = [r1;...;rn]
```

where conv "f ti" returns |- (f ti) = ri for i from 1 to n.

Example

The following is a very simple example in which no computation is done for applications of the function being mapped down a list:

```
- MAP_CONV ALL_CONV "MAP SUC [1;2;1;4]";
|- MAP SUC[1;2;1;4] = [SUC 1;SUC 2;SUC 1;SUC 4]
```

The result just contains applications of SUC, since the supplied conversion ALL_CONV does no evaulation.

We now construct a conversion that maps SUC n for any numeral n to the numeral standing for the successor of n:

```
- fun SUC_CONV tm =
     let val n = string_to_int(#Name(dest_const(rand tm)))
         val sucn = mk_const{{Name =int_to_string(n+1), Ty=(==`:num`==)}}
      in
         SYM (num_CONV sucn)
      end;
SUC_CONV = - : conv
```

The result is a conversion that inverts num_CONV:

```
- num_CONV “4”;
|- 4 = SUC 3

- SUC_CONV “SUC 3”;
|- SUC 3 = 4
```

The conversion SUC_CONV can then be used to compute the result of mapping the successor function down a list of numerals:

```
- MAP_CONV SUC_CONV “MAP SUC [1;2;1;4]”;
|- MAP SUC[1;2;1;4] = [2;3;2;5]
```

Failure

MAP_CONV conv fails if applied to a term not of the form “MAP f [t1;...;tn]”. An application of MAP_CONV conv to a term “MAP f [t1;...;tn]” fails unless for all ti in the list [t1;...;tn], evaluating conv “f ti” returns |- (f ti) = ri for some ri.

MAP_EVERY (Tactical)

```
Tactical.MAP_EVERY : (('a -> tactic) -> 'a list -> tactic)
```

Sequentially applies all tactics given by mapping a function over a list.

When applied to a tactic-producing function f and an operand list [x1;...;xn], the elements of which have the same type as f’s domain type, MAP_EVERY maps the function f over the list, producing a list of tactics, then applies these tactics in sequence as in the case of EVERY. The effect is:

```
MAP_EVERY f [x1;...;xn] = (f x1) THEN ... THEN (f xn)
```

If the operand list is empty, then MAP_EVERY has no effect.

Failure

The application of MAP_EVERY to a function and operand list fails iff the function fails when applied to any element in the list. The resulting tactic fails iff any of the resulting tactics fails.

Example

A convenient way of doing case analysis over several boolean variables is:

```
MAP_EVERY BOOL_CASES_TAC ["var1:bool";...;"varn:bool"]
```

See also

Tactical.EVERY, Tactical.FIRST, Tactical.MAP_FIRST, Tactical.THEN

MAP_FIRST (Tactical)

```
Tactical.MAP_FIRST : (('a -> tactic) -> 'a list -> tactic)
```

Applies first tactic that succeeds in a list given by mapping a function over a list.

When applied to a tactic-producing function f and an operand list [x1,...,xn], the elements of which have the same type as f's domain type, MAP_FIRST maps the function f over the list, producing a list of tactics, then tries applying these tactics to the goal till one succeeds. If f(xm) is the first to succeed, then the overall effect is the same as applying f(xm). Thus:

```
MAP_FIRST f [x1,...,xn] = (f x1) ORELSE ... ORELSE (f xn)
```

Failure

The application of MAP_FIRST to a function and tactic list fails iff the function does when applied to any of the elements of the list. The resulting tactic fails iff all the resulting tactics fail when applied to the goal.

See also

`Tactical.EVERY`, `Tactical.FIRST`, `Tactical.MAP_EVERY`, `Tactical.ORELSE`

mapfilter (Lib)

```
Lib.mapfilter : ('a -> 'b) -> 'a list -> 'b list
```

Like map, but drops elements that raise exceptions.

Applies a function to every element of a list, returning a list of results for those elements for which application succeeds. The function is applied to the elements of the list from left to right (which is significant if its action includes side effects).

Failure

If `f x` raises `Interrupt` for some element `x` of `l`, then `mapfilter f l` fails (with an `Interrupt` exception).

Example

```
> mapfilter hd [[1,2,3],[4,5],[],[6,7,8],[]];
val it = [1, 4, 6]: int list
```

See also

`Lib.filter`

|-> (Lib)

```
op Lib.|-> : 'a * 'b -> {redex : 'a, residue : 'b}
```

Infix operator for building a component of a substitution.

An application `x |-> y` is equal to `{redex = x, residue = y}`. Since HOL substitutions are lists of `{redex,residue}` records, the `|->` operator is merely sugar used to create substitutions.

Failure

Never fails.

Example

```
> type_subst [alpha |-> beta, beta |-> gamma]
             (alpha --> beta);
val it = ":β -> γ": hol_type
```

See also

`Lib.subst`, `Type.type_subst`, `Term.subst`, `Term.inst`, `Thm.SUBST`

match (DB)

```
DB.match : string list -> term -> data list
```

Attempt to find matching theorems in the specified theories.

An invocation `DB.match [s1,...,sn] M` collects all theorems, definitions, and axioms of the theories designated by `s1,...,sn` that have a subterm that matches `M`. If there are no matches, the empty list is returned.

The strings `s1,...,sn` should be a subset of the currently loaded theory segments. The string `"-"` may be used to designate the current theory segment. If the list of theories is empty, then all currently loaded theories are searched.

Failure

Never fails.

Example

```
> DB.match ["bool","pair"] (Term `(a = b) = c`);
<<HOL message: inventing new type variable names: 'a>>
val it =
   [(("bool", "bool_case_eq"),
     (⊢ (if x then t1 else t2) = v ⇔ (x ⇔ T) ∧ t1 = v ∨ (x ⇔ F) ∧ t2 = v,
      Thm,
      Located
       {exact = true, linenum = 3552, scriptpath =
        "$(HOLDIR)/src/bool/boolScript.sml"})),
    (("bool", "EQ_CLAUSES"),
     (⊢ ∀t. ((T ⇔ t) ⇔ t) ∧ ((t ⇔ T) ⇔ t) ∧ ((F ⇔ t) ⇔ ¬t) ∧ ((t ⇔ F) ⇔ ¬t),
      Thm,
      Located
       {exact = true, linenum = 1310, scriptpath =
        "$(HOLDIR)/src/bool/boolScript.sml"})),
    (("bool", "EQ_EXPAND"),
[...Output elided...]
```

Comments

The notion of matching is a restricted version of higher-order matching.

For locating theorems when doing interactive proof.

See also

`DB.matcher`, `DB.matchp`, `DB.find`, `DB.theorems`, `DB.thy`, `DB.listDB`

match (hol88Lib)

```
hol88Lib.match : term -> term -> (term * term) list * (hol_type * hol_type) list
```

Finds instantiations to match one term to another.

When applied to two terms, `match_term` attempts to find a set of type and term instantiations for the first term (only) to make it equal the second. If it succeeds, it returns

the instantiations in the form of a pair containing a hol88 term substitution and a hol88 type substitution. If the first term represents the conclusion of a theorem, the returned instantiations are of the appropriate form to be passed to `INST_TY_TERM`.

Failure

Fails if the term cannot be matched by one-way instantiation.

Comments

Note that `INST_TY_TERM` may still fail (when a variable that is instantiated occurs free in the theorem's assumptions).

Superseded by `Term.match_term`.

See also

`Term.match_term`

MATCH_ABBREV_TAC (Q)

```
Q.MATCH_ABBREV_TAC : term quotation -> tactic
```

Introduces abbreviations by matching a pattern against the goal statement.

When applied to the goal `(asl, w)`, the tactic `Q.MATCH_ABBREV_TAC q` parses the quotation `q` in the context of the goal, producing a term to use as a pattern. The tactic then attempts a (first order) match of the pattern against the term `w`. Variables that occur in both the pattern and the goal are treated as "local constants", and will not acquire instantiations.

For each variable `v` in the pattern that has not been treated as a local constant, there will be an instantiation term `t`, such that the substitution pattern `[v1 |-> t1, v2 |-> t2, ...]` produces `w`. The effect of the tactic is to then perform abbreviations in the goal, replacing each `t` with the corresponding `v` (as long as `v` does not have a name beginning with an underscore character), and adding assumptions of the form `Abbrev(v = t)` to the goal.

Because the tactic ignores underscore variables, the user can abbreviate just those parts of the goal that are particularly relevant. Note also that the standard parser treats variables consisting of entirely underscores specially: each is expanded to a fresh name. This means that a pattern can use _ repeatedly, and it will not cause the match to look for the same instantiation for each occurrence. Nor it will require corresponding sub-terms to have the same type.

Failure

`MATCH_ABBREV_TAC` fails if the pattern provided does not match the goal, or if variables from the goal are used in the pattern in ways that make the pattern fail to type-check.

Example

If the current goal is

```
?- (n + 10) * y <= 42315 /\ (!x y. x < y ==> f x < f y)
```

then applying the tactic `Q.MATCH_ABBREV_TAC` `` `X <= Y /\ P` `` results in the goal

```
Abbrev(X = (n + 10) * y),
Abbrev(Y = 42315),
Abbrev(P = !x y. x < y ==> f x < f y)
   ?-
X <= Y /\ P
```

If the current goal is

```
?- (n + 10) * y <= 42315 /\ (!x y. x < y ==> f x < f y)
```

then applying the tactic `Q.MATCH_ABBREV_TAC` `` `a * _ <= b /\ _` `` results in the goal

```
Abbrev (a = n + 10)
Abbrev (b = 42315)
  ?-
a * y <= b /\ !x y. x < y ==> f x < f y
```

See also

`Q.ABBREV_TAC`, `Q.HO_MATCH_ABBREV_TAC`

MATCH_ACCEPT_TAC (Tactic)

```
Tactic.MATCH_ACCEPT_TAC : thm_tactic
```

Solves a goal which is an instance of the supplied theorem.

When given a theorem `A' |- t` and a goal `A ?- t'` where `t` can be matched to `t'` by instantiating variables which are either free or universally quantified at the outer level, including appropriate type instantiation, `MATCH_ACCEPT_TAC` completely solves the goal.

```
 A ?- t'
=========  MATCH_ACCEPT_TAC (A' |- t)
```

Unless `A'` is a subset of `A`, this is an invalid tactic.

Failure

Fails unless the theorem has a conclusion which is instantiable to match that of the goal.

Example

The following example shows variable and type instantiation at work. We can use the polymorphic list theorem `HD`:

```
HD = |- !h t. HD(CONS h t) = h
```

to solve the goal:

```
?- HD [1;2] = 1
```

simply by:

```
MATCH_ACCEPT_TAC HD
```

Comments

`prim_irule` is similar, with differences in the details

See also

`Tactic.ACCEPT_TAC`, `Tactic.prim_irule`

MATCH_ASMSUB_RENAME_TAC (Q)

```
Q.MATCH_ASMSUB_RENAME_TAC : term quotation -> string list -> tactic
```

Finds a match for a pattern in assumptions; instantiates goal to rename or abbreviate.

When applied to the goal `(asl,w)`, the tactic `Q.MATCH_ASMSUB_RENAME_TAC q` parses the quotation `q` in the context of the goal, producing a term `pat` to use as a pattern. The tactic then attempts a (first order) match of `pat` against each term in `asl`, stopping when it finds an assumption that either matches `pat` in its entirety, or has a sub-term that matches `pat` (holding existing free variables from the goal constant).

In either case, the match will return an instantiation mapping the fresh free variables of `pat` to terms occurring in the goal. This instantiation drops its bindings for variables whose names begin with an underscore, is next reversed, and is finally applied to the goal. This will both cause free variables in the goal to be renamed, and for larger terms to be replaced by variables (similar to what happens with the use of `SPEC_TAC`).

Failure

Fails if there is no valid match for the pattern among the assumptions and their sub-terms. A valid match will not bind variables that are bound in the assumption being searched.

Example

In the following example, the variable `x` is treated as if a constant, so the search for a match with the pattern does not succeed on the first assumption (featuring `P`). Instead the second assumption provides the instantiation, and so the variable `z` in the original is renamed to `n`.

```
> Q.MATCH_ASMSUB_RENAME_TAC `x < n`
    ([``P(y < m):bool``, ``Q(x < z) : bool``], ``x + z < 10``);
val it = ([([``P (y < m)``, ``Q (x < n)``], ``x + n < 10``)],
          fn): goal list * validation
```

See also

Q.MATCH_ASSUM_RENAME_TAC, Q.MATCH_GOALSUB_RENAME_TAC

MATCH_ASSUM_ABBREV_TAC (Q)

```
Q.MATCH_ASSUM_ABBREV_TAC : term quotation -> tactic
```

Introduces abbreviations by matching a pattern against an assumption.

When applied to the goal `(asl, w)`, the tactic `Q.MATCH_ASSUM_ABBREV_TAC q` parses the quotation `q` in the context of the goal, producing a term to use as a pattern. The tactic then attempts a (first order) match of the pattern against each term in `asl`, stopping on the first matching assumption `a`. Variables that occur in both the pattern and the goal are treated as "local constants", and will not acquire instantiations.

For each variable `v` in the pattern that has not been treated as a local constant, there will be an instantiation term `t`, such that the substitution `pattern[v1 |-> t1, v2 |-> t2, ...]` produces `a`. The effect of the tactic is to then perform abbreviations in the goal, replacing each `t` with the corresponding `v`, and adding assumptions of the form `Abbrev(v = t)` to the goal.

Failure

`MATCH_ABBREV_TAC` fails if the pattern provided does not match any assumption, or if variables from the goal are used in the pattern in ways that make the pattern fail to type-check.

Comments

This tactic improves on the following tedious workflow: `Q.PAT_ASSUM pat MP_TAC`, `Q.MATCH_ABBREV_TAC `pat ==> X``, `Q.UNABBREV_TAC `X``, `STRIP_TAC`.

See also

Q.MATCH_ABBREV_TAC, Q.MATCH_ASSUM_RENAME_TAC

MATCH_ASSUM_RENAME_TAC (Q)

```
Q.MATCH_ASSUM_RENAME_TAC : term quotation -> tactic
```

Replaces selected terms with new variables by matching a pattern against an assumption.

When applied to the goal `(asl, w)`, the tactic `Q.MATCH_ASSUM_RENAME_TAC q` parses the quotation `q` in the context of the goal, producing a term to use as a pattern. The tactic then attempts a (first order) match of the pattern against each term in `asl`, stopping on the first matching assumption `a`.

For each variable `v` in the pattern, there will be an instantiation term `t`, such that the substitution

```
pattern[v1 |-> t1, v2 |-> t2, ...]
```

produces `a`. The effect of the tactic is to then replace each `t` with the corresponding `v`, yielding a new goal. Note that underscores within a pattern, though strictly speaking variables, are not included in this reverse instantiation.

Failure

`MATCH_ASSUM_RENAME_TAC` fails if the pattern provided does not match any assumption, or if variables from the goal are used in the pattern in ways that make the pattern fail to type-check.

Example

If the current goal is

```
(f x = Pair C'' C0') ?- (f C'' = f C0')
```

then applying the tactic `Q.MATCH_ASSUM_RENAME_TAC` `` `_ = Pair c1 c2` `` results in the goal

```
(f x = Pair c1 c2) ?- (f c1 = f c2)
```

Comments

This tactic improves on the following tedious workflow: `Q.PAT_ASSUM pat MP_TAC`, `Q.MATCH_ABBREV_TAC `pat ==> X``, `Q.UNABBREV_TAC `X``, `markerLib.RM_ALL_ABBREVS_TAC`, `STRIP_TAC`.

See also

`Q.MATCH_RENAME_TAC`

MATCH_GOALSUB_RENAME_TAC (Q)

```
Q.MATCH_GOALSUB_RENAME_TAC : term quotation -> tactic
```

Renames a goal in accordance with a pattern matched against a subterm of the goal.

A call to `MATCH_GOALSUB_RENAME_TAC pat` attempts to find a match for the pattern `pat` in the current goal (using `gen_find_term` to find a sub-term of the goal that matches). If a match is found, the goal is adjusted so that the variables occurring in the pattern now also appear in the goal. This may rename variables in the goal, or even cause larger sub-terms to be replaced by variables (as with `SPEC_TAC`). Underscores may be used in `pat` to indicate "don't care" bindings, where no renaming or instantiation will take place.

Failure

Fails if there is no sub-term of the goal that matches the pattern. Fails if the instantiation changes a pattern variable that already exists in the goal.

Example

If the goal is

```
?- !x. x * 2 < y * (z + 1) * (y + a)
```

then applying `Q.MATCH_GOALSUB_RENAME_TAC `y + c`` will match the pattern `y + c` against the various subterms within the goal. The first obvious match, with `z + 1` will

be rejected because the variable `y` is free in the goal, and is treated as if it were a local constant. Because of this, `y + a` is the matching sub-term, and after renaming the goal becomes

```
?- !x. x * 2 < y * (z + 1) * (y + c)
```

See also

`Q.MATCH_RENAME_TAC`

MATCH_MP (Drule)

```
Drule.MATCH_MP : thm -> thm -> thm
```

Modus Ponens inference rule with automatic matching.

When applied to theorems `A1 |- !x1...xn. t1 ==> t2` and `A2 |- t1'`, the inference rule `MATCH_MP` matches `t1` to `t1'` by instantiating free or universally quantified variables in the first theorem (only), and returns a theorem `A1 u A2 |- !xa..xk. t2'`, where `t2'` is a correspondingly instantiated version of `t2`. Polymorphic types are also instantiated if necessary.

Variables free in the consequent but not the antecedent of the first argument theorem will be replaced by variants if this is necessary to maintain the full generality of the theorem, and any which were universally quantified over in the first argument theorem will be universally quantified over in the result, and in the same order.

```
 A1 |- !x1..xn. t1 ==> t2   A2 |- t1'
-------------------------------------  MATCH_MP
       A1 u A2 |- !xa..xk. t2'
```

As with `MP` and the underlying syntactic function `dest_imp`, negated terms (of the form `~p`) are treated as if they were implications from the argument of the negation to falsity.

Failure

Fails unless the first theorem is a (possibly repeatedly universally quantified) implication (in the sense of `dest_imp`) whose antecedent can be instantiated to match the conclusion

of the second theorem, without instantiating any variables which are free in A1, the first theorem's assumption list.

Example

In this example, automatic renaming occurs to maintain the most general form of the theorem, and the variant corresponding to z is universally quantified over, since it was universally quantified over in the first argument theorem.

```
- val ith = (GENL [Term `x:num`, Term `z:num`]
               o DISCH_ALL
               o AP_TERM (Term `$+ (w + z)`))
            (ASSUME (Term `x:num = y`));
> val ith = |- !x z. (x = y) ==> (w + z + x = w + z + y) : thm

- val th = ASSUME (Term `w:num = z`);
> val th = [w = z] |- w = z : thm

- MATCH_MP ith th;
> val it =  [w = z] |- !z'. w' + z' + w = w' + z' + z : thm
```

See also

boolSyntax.dest_imp, Thm.EQ_MP, Tactic.MATCH_MP_TAC, Thm.MP, Tactic.MP_TAC, ConseqConv.CONSEQ_REWRITE_CONV

MATCH_MP_TAC (Tactic)

```
Tactic.MATCH_MP_TAC : thm_tactic
```

Reduces the goal using a supplied implication, with matching.

When applied to a theorem of the form

```
A' |- !x1...xn. s ==> !y1...ym. t
```

MATCH_MP_TAC produces a tactic that reduces a goal whose conclusion t' is a substitution and/or type instance of t to the corresponding instance of s. Any variables free in s but not in t will be existentially quantified in the resulting subgoal:

```
    A ?- !v1...vi. t'
 ======================  MATCH_MP_TAC (A' |- !x1...xn. s ==> !y1...ym. t)
    A ?- ?z1...zp. s'
```

where z1, ..., zp are (type instances of) those variables among x1, ..., xn that do not occur free in t. Note that this is not a valid tactic unless A' is a subset of A.

Failure

Fails unless the theorem is an (optionally universally quantified) implication whose consequent can be instantiated to match the goal. The generalized variables v1, ..., vi must occur in s' in order for the conclusion t of the supplied theorem to match t'.

Comments

The tactic irule builds on MATCH_MP_TAC, normalising the input theorem more aggressively so that it succeeds more often.

See also

ConseqConv.CONSEQ_CONV_TAC, Thm.EQ_MP, Tactic.irule, Drule.MATCH_MP, Thm.MP, Tactic.MP_TAC

MATCH_RENAME_TAC (Q)

```
Q.MATCH_RENAME_TAC : term quotation -> tactic
```

Replaces selected terms with new variables by matching a pattern against the goal statement.

When applied to the goal (asl, w), the tactic Q.MATCH_RENAME_TAC q ls parses the quotation q in the context of the goal, producing a term to use as a pattern. The tactic then attempts a (first order) match of the pattern against the term w.

For each variable `v` in the pattern, there will be an instantiation term `t`, such that the substitution

```
pattern[v1 |-> t1, v2 |-> t2, ...]
```

produces `w`. The effect of the tactic is to then replace each `t` with the corresponding `v`, yielding a new goal. Note that underscores within a pattern, though strictly speaking variables, are not included in this reverse instantiation.

Failure

`MATCH_RENAME_TAC` fails if the pattern provided does not match the goal, or if variables from the goal are used in the pattern in ways that make the pattern fail to type-check.

Example

If the current goal is

```
?- (f x = Pair C'' C0') ==> (f C'' = f C0')
```

then applying the tactic `Q.MATCH_RENAME_TAC `(f x = Pair c1 c2) ==> _`` results in the goal

```
?- (f x = Pair c1 c2) ==> (f c1 = f c2)
```

Comments

This tactic is equivalent to first applying `Q.MATCH_ABBREV_TAC q`, then applying `Q.RM_ABBREV_TAC `v`` for each underscore in `q`.

See also

`Q.MATCH_ABBREV_TAC`, `Q.MATCH_ASSUM_RENAME_TAC`

match_term (Term)

```
Term.match_term : term -> term -> (term,term) subst * (hol_type,hol_type) subst
```

Finds instantiations to match one term to another.

An application `match_term M N` attempts to find a set of type and term instantiations for `M` to make it alpha-convertible to `N`. If `match_term` succeeds, it returns the instantiations in the form of a pair containing a term substitution and a type substitution. In particular, if `match_term pat ob` succeeds in returning a value `(S,T)`, then

```
aconv (subst S (inst T pat)) ob.
```

Failure

Fails if the term cannot be matched by one-way instantiation. If the pattern includes variables of the same name but different types, the resulting type instantiation may cause those variables to be identified and the term instantiation to be useless.

Example

The following shows how `match_term` could be used to match the conclusion of a theorem to a term.

```
> val th = REFL “x:'a”;
val th = ⊢ x = x: thm

> match_term (concl th) “1 = 1”;
val it =
   ([{redex = “x”, residue = “1”}], [{redex = “:α”, residue = “:num”}]):
   (term, term) Term.subst * (hol_type, hol_type) Term.subst

> INST_TY_TERM it th;
val it = ⊢ 1 = 1: thm
```

The following shows an attempt to use a bad pattern (the pattern term `t` has two variables called `x` at different types):

```
> val _ = show_types := true;

> val t = list_mk_comb(``f:'a -> 'b -> 'c``, [``x:'a``, ``x:'b``]);
val t = “(f :α -> β -> γ) (x :α) (x :β)”: term
```

```
> val (tminst, tyinst) = match_term t ``(g: 'a -> 'a -> 'b) a b``;
val tminst =
   [{redex = “(f :α -> α -> β)”, residue = “(g :α -> α -> β)”},
    {redex = “(x :α)”, residue = “(a :α)”},
    {redex = “(x :α)”, residue = “(b :α)”}]: (term, term) Term.subst
val tyinst =
   [{redex = “:γ”, residue = “:β”}, {redex = “:β”, residue = “:α”}]:
   (hol_type, hol_type) Term.subst
```

The `tminst` value is unusable as it seeks to instantiate two different `x` variables (one with `a`, one with `b`) that are now actually the same variable.

Comments

For instantiating theorems `PART_MATCH` is usually easier to use.

See also

`Term.match_terml`, `Type.match_type`, `Drule.INST_TY_TERM`, `Drule.PART_MATCH`

match_terml (Term)

```
Term.match_terml
     : hol_type list -> term set -> term -> term
       -> (term,term) subst * (hol_type,hol_type) subst
```

Match two terms while restricting some instantiations.

An invocation `match_terml avoid_tys avoid_tms pat ob (tmS,tyS)`, if it does not raise an exception, returns a pair of substitutions `(S,T)` such that

```
aconv (subst S (inst T pat)) ob.
```

The arguments `avoid_tys` and `avoid_tms` specify type and term variables in `pat` that are not allowed to become `redexes` in `S` and `T`.

Failure

`match_terml` will fail if no `S` and `T` meeting the above requirements can be found. If a match `(S,T)` between `pat` and `ob` can be found, but elements of `avoid_tys` would appear as redexes in `T` or elements of `avoid_tms` would appear as redexes in `S`, then `match_terml` will also fail.

Example

```
> val (tmS,tyS) = match_terml [] empty_varset
                  (Term `\x:'a. x = f (y:'b)`)
                  (Term `\a.    a = ~p`);
val tmS = [{redex = “f”, residue = “$¬”}, {redex = “y”, residue = “p”}]:
   (term, term) Term.subst
val tyS =
   [{redex = “:β”, residue = “:bool”}, {redex = “:α”, residue = “:bool”}]:
   (hol_type, hol_type) Term.subst

> match_terml [alpha] empty_varset  (* forbid instantiation of 'a *)
        (Term `\x:'a. x = f (y:'b)`)
        (Term `\a.    a = ~p`);
Exception- HOL_ERR (at Type.raw_match_type: double bind on type variable 'a) raised

> match_terml [] (HOLset.add(empty_varset,mk_var("y",beta)))
        (Term `\x:'a. x = f (y:'b)`)
        (Term `\a.    a = ~p`);
Exception- HOL_ERR (at Term.raw_match_term: double bind on variable "y") raised
```

See also

`Term.match_term`, `Term.raw_match`, `Term.subst`, `Term.inst`, `Type.match_typel`, `Type.type_subst`

match_type (Type)

```
Type.match_type : hol_type -> hol_type -> (hol_type,hol_type) subst
```

Calculates a substitution `theta` such that instantiating the first argument with `theta` equals the second argument.

If `match_type ty1 ty2` succeeds, then

```
type_subst (match_type ty1 ty2) ty1 = ty2
```

Failure

If no such substitution can be found.

Example

```
> match_type alpha (Type`:num`);
val it = [{redex = ":α", residue = ":num"}]: (hol_type, hol_type) Lib.subst

> let val patt = Type`:('a -> bool) -> 'b`
     val ty =   Type`:(num -> bool) -> bool`
  in
    type_subst (match_type patt ty) patt = ty
  end;
val it = true: bool

> match_type (alpha --> alpha)
           (ind   --> bool);
Exception- HOL_ERR (at Type.raw_match_type: double bind on type variable 'a) raised
```

See also

`Term.match_term`, `Type.type_subst`

matcher (DB)

```
DB.matcher : (term -> term -> 'a) -> string list -> term -> data list
```

All theory elements matching a given term.

An invocation `matcher pm [thy1,...,thyn] M` collects all elements of the theory segments `thy1,...,thyn` that have a subterm `N` such that `pm M` does not fail (raise an exception) when applied to `N`. Thus `matcher` potentially traverses all subterms of all theorems in all the listed theories in its search for 'matches'.

If the list of theory segments is empty, then all currently loaded segments are examined. The string `"-"` may be used to designate the current theory segment.

Failure

Never fails, but may return an empty list.

Example

```
> DB.matcher match_term ["relation"] (Term `P \/ Q`);
val it =
   [(("relation", "IN_RDOM_RUNION"),
     (⊢ x ∈ RDOM (R1 ∪ᵣ R2) ⇔ x ∈ RDOM R1 ∨ x ∈ RDOM R2, Thm,
      Located
       {exact = true, linenum = 2392, scriptpath =
        "$(HOLDIR)/src/relation/relationScript.sml"})),
    (("relation", "RC_DEF"),
     (⊢ ∀R x y. RC R x y ⇔ x = y ∨ R x y, Def, Unknown)),
    (("relation", "RINSERT"),
     (⊢ ∀R a b. RINSERT R a b = (λx y. R x y ∨ x = a ∧ y = b), Def, Unknown)),
    (("relation", "RTC_cases"),
     (⊢ ∀R a0 a1. R* a0 a1 ⇔ a1 = a0 ∨ ∃y. R a0 y ∧ R* y a1, Thm, Unknown)),
    (("relation", "RTC_CASES1"),
     (⊢ ∀R x y. R* x y ⇔ x = y ∨ ∃u. R x u ∧ R* u y, Thm,
      Located
[...Output elided...]

> DB.matcher (ho_match_term [] empty_varset) [] (Term `?x. P x \/ Q x`);
<<HOL message: inventing new type variable names: 'a>>
val it =
   [(("arithmetic", "ODD_OR_EVEN"),
     (⊢ ∀n. ∃m. n = SUC (SUC 0) * m ∨ n = SUC (SUC 0) * m + 1, Thm,
      Located
       {exact = true, linenum = 1571, scriptpath =
        "$(HOLDIR)/src/num/theories/arithmeticScript.sml"})),
```

```
    (("bool", "EXISTS_OR_THM"),
     (⊢ ∀P Q. (∃x. P x ∨ Q x) ⇔ (∃x. P x) ∨ ∃x. Q x, Thm,
      Located
       {exact = true, linenum = 1604, scriptpath =
        "$(HOLDIR)/src/bool/boolScript.sml"})),
    (("bool", "LEFT_OR_EXISTS_THM"),
     (⊢ ∀P Q. (∃x. P x) ∨ Q ⇔ ∃x. P x ∨ Q, Thm,
      Located
[...Output elided...]
```

Comments

Usually, `pm` will be a pattern-matcher, but it need not be.

See also

`DB.match`, `DB.apropos`, `DB.matchp`, `DB.find`

matches (DB)

```
DB.matches : term -> thm -> bool
```

Tells whether part of a theorem matches a pattern.

An invocation `DB.match pat th` tells whether the conclusion of `th` has a subterm matching `pat`.

Failure

Never fails.

Example

```
> DB.matches (Term `(a = b) = c`) EQ_CLAUSES ;
<<HOL message: inventing new type variable names: 'a>>
val it = true: bool
```

```
> DB.matches (Term `(a = b) = c`)  EQ_TRANS ;
<<HOL message: inventing new type variable names: 'a>>
val it = false: bool
```

Comments

The notion of matching is a restricted version of higher-order matching, as used by `DB.apropos`, `DB.apropos_in`, `DB.match`, etc.

For locating theorems relevant to a given pattern.

See also

`DB.matcher`, `DB.matchp`, `DB.apropos`, `DB.apropos_in`

matchp (DB)

```
DB.matchp : (thm -> bool) -> string list -> data list
```

All theory elements satisfying a predicate.

An invocation `matchp P [thy1,...,thyn]` collects all elements of the theory segments `thy1`,...,`thyn` that `P` holds of. If the list of theory segments is empty, then all currently loaded segments are examined. The string `"-"` may be used to designate the current theory segment.

Failure

Fails if `P` fails when applied to a theorem in one of the theories being searched.

Example

The following query returns all unconditional rewrite rules in the theory `pair`.

```
> matchp (is_eq o snd o strip_forall o concl) ["pair"];
val it =
```

```
   [(("pair", "C_UNCURRY_L"),
     (⊢ flip (UNCURRY f) x = UNCURRY (flip (flip ∘ f) x), Thm,
      Located
       {exact = true, linenum = 602, scriptpath =
        "$(HOLDIR)/src/coretypes/pairScript.sml"})),
    (("pair", "CLOSED_PAIR_EQ"),
     (⊢ ∀x y a b. (x,y) = (a,b) ⇔ x = a ∧ y = b, Thm,
      Located
       {exact = true, linenum = 106, scriptpath =
        "$(HOLDIR)/src/coretypes/pairScript.sml"})),
    (("pair", "COMMA_DEF"),
     (⊢ ∀x y. (x,y) = ABS_prod (λa b. a = x ∧ b = y), Def, Unknown)),
    (("pair", "CURRY_DEF"), (⊢ ∀f x y. CURRY f x y = f (x,y), Def, Unknown)),
    (("pair", "CURRY_DEF_lazyfied"),
[...Output elided...]
```

See also

DB.match, DB.matcher, DB.apropos, DB.find

max_print_depth (Globals)

```
Globals.max_print_depth : int ref
```

Sets depth bound on prettyprinting.

The reference variable max_print_depth is used to define the maximum depth of printing for the pretty printer. If the number of blocks (an internal notion used by the prettyprinter) becomes greater than the value set by max_print_depth then the blocks are abbreviated by the holophrast . . . . By default, the value of max_print_depth is ~1. This is interpreted to mean 'print everything'.

Failure

Never fails.

Example

To change the maximum depth setting to 10, the command will be:

```
- max_print_depth := 10;
> val it = () : unit
```

The theorem numeralTheory.numeral_distrib then prints as follows:

```
> numeralTheory.numeral_distrib;
val it =
   ⊢ (∀n. 0 + n = n) ∧ (∀n. n + 0 = n) ∧
     (∀n m. NUMERAL n + NUMERAL m = NUMERAL (numeral$iZ (n + m))) ∧
     (∀n. 0 * n = 0) ∧ (∀n. n * 0 = 0) ∧
     (∀n m. NUMERAL n * NUMERAL m = NUMERAL (n * m)) ∧ (∀n. 0 − n = 0) ∧
     (∀n. n − 0 = n) ∧ (∀n m. NUMERAL n − NUMERAL m = NUMERAL (n − m)) ∧
     (∀n. 0 ** NUMERAL (BIT1 n) = 0) ∧ (∀n. 0 ** NUMERAL (BIT2 n) = 0) ∧
     (∀n. n ** 0 = 1) ∧ (∀n m. NUMERAL n ** NUMERAL m = NUMERAL (n ** m)) ∧
     SUC 0 = 1 ∧ (∀n. SUC (NUMERAL n) = NUMERAL (SUC n)) ∧ PRE 0 = 0 ∧
     (∀n. PRE (NUMERAL n) = NUMERAL (PRE n)) ∧
     (∀n. NUMERAL n = 0 ⇔ n = ZERO) ∧ (∀n. 0 = NUMERAL n ⇔ n = ZERO) ∧
     (∀n m. NUMERAL n = NUMERAL m ⇔ n = m) ∧ (∀n. n < 0 ⇔ F) ∧
     (∀n. 0 < NUMERAL n ⇔ ZERO < n) ∧ (∀n m. NUMERAL n < NUMERAL m ⇔ n < m) ∧
     (∀n. 0 > n ⇔ F) ∧ (∀n. NUMERAL n > 0 ⇔ ZERO < n) ∧
     (∀n m. NUMERAL n > NUMERAL m ⇔ m < n) ∧ (∀n. 0 ≤ n ⇔ T) ∧
     (∀n. NUMERAL n ≤ 0 ⇔ n ≤ ZERO) ∧ (∀n m. NUMERAL n ≤ NUMERAL m ⇔ n ≤ m) ∧
     (∀n. n ≥ 0 ⇔ T) ∧ (∀n. 0 ≥ n ⇔ n = 0) ∧
     (∀n m. NUMERAL n ≥ NUMERAL m ⇔ m ≤ n) ∧ (∀n. ODD (NUMERAL n) ⇔ ODD n) ∧
     (∀n. EVEN (NUMERAL n) ⇔ EVEN n) ∧ ¬ODD 0 ∧ EVEN 0: thm
```

max_print_length (Globals)

```
Globals.max_print_length : int ref
```

Sets a length bound on pretty printing for list forms.

Defines the maximum number of elements that the pretty printer will print for list forms. This closely mirrors Globals.max_print_depth: remaining elements are abbreviated by . . ., and the default value of ~1 means ‘print all elements’.

Failure

Never fails.

See also

`Globals.max_print_depth`

measureInduct_on (bossLib)

```
bossLib.measureInduct_on : term quotation -> tactic
```

Perform complete induction with a supplied measure function.

If `q` parses into a well-typed term `M N`, an invocation `measureInduct_on q` begins a proof by induction, using `M` to map `N` into a number. The term `N` should occur free in the current goal.

Failure

If `M N` does not parse into a term or if `N` does not occur free in the current goal.

Example

Suppose we wish to prove `P (APPEND l1 l2)` by induction on the length of `l1`. Then `measureInduct_on` `` `LENGTH l1` `` yields the goal

```
{ !y. LENGTH y < LENGTH l1 ==> P (APPEND y l2) } ?- P (APPEND l1 l2)
```

See also

`bossLib.completeInduct_on`, `bossLib.Induct`, `bossLib.Induct_on`

mem (Lib)

```
Lib.mem : ''a -> ''a list -> bool
```

Tests whether a list contains a certain member.

An invocation `mem x [x1,...,xn]` returns `true` if some `xi` in the list is equal to `x`. Otherwise it returns `false`.

Failure

Never fails.

Comments

Note that the type of the members of the list must be an SML equality type. If set operations on a non-equality type are desired, use the 'op_' variants, which take an equality predicate as an extra argument.

A high-performance implementation of finite sets may be found in structure `HOLset`.

See also

`Lib.op_mem`, `Lib.insert`, `Lib.tryfind`, `Lib.exists`, `Lib.all`, `Lib.assoc`, `Lib.rev_assoc`

merge (Tag)

```
Tag.merge : tag -> tag -> tag
```

Combine two tags into one.

When two theorems interact via inference, their tags are merged. This propagates to the new theorem the fact that either or both were constructed via shortcut.

Failure

Never fails.

Example

```
> Tag.merge (Tag.read "foo") (Tag.read "bar");
val it = [oracles: ##] [axioms: ]: tag

> Tag.merge it (Tag.read "foo");
val it = [oracles: ##] [axioms: ]: tag
```

Comments

Although it is not harmful to use this entrypoint, there is little reason to, since the merge operation is only used inside the HOL kernel.

See also

`Tag.read`, `Thm.mk_oracle_thm`, `Thm.tag`

MESG_outstream (Feedback)

```
Feedback.MESG_outstream : TextIO.outstream ref
```

Reference to output stream used when printing `HOL_MESG`.

The value of reference cell `MESG_outstream` controls where `HOL_MESG` prints its argument.

The default value of `MESG_outstream` is a function that prints its argument to `TextIO.stdOut`.

Example

```
> val outputs = ref ([] : string list);
val outputs = ref []: string list ref
```

```
> MESG_outstream := (fn s => outputs := !outputs @ [s]);
val it = (): unit

> HOL_MESG "Nattering nabobs of negativity.";
val it = (): unit

> outputs;
val it = ref ["<<HOL message: Nattering nabobs of negativity.>>\n"]:
   string list ref
```

See also

Feedback, Feedback.HOL_MESG, Feedback.ERR_outstream, Feedback.WARNING_outstream, Feedback.emit_MESG

MESG_to_string (Feedback)

```
Feedback.MESG_to_string : (string -> string) ref
```

Alterable function for formatting HOL_MESG.

MESG_to_string is a reference to a function for formatting the argument to an application of HOL_MESG.

The default value of MESG_to_string is format_MESG.

Example

```
- fun alt_MESG_report s = String.concat["Dear HOL user: ", s, "\n"];

- MESG_to_string := alt_MESG_report;

- HOL_MESG "Hi there."

Dear HOL user: Hi there.
> val it = () : unit
```

See also

Feedback, Feedback.HOL_MESG, Feedback.format_MESG, Feedback.ERR_to_string, Feedback.WARNING_to_string

MESON_TAC (mesonLib)

mesonLib.MESON_TAC : thm list -> tactic

Performs first order proof search to prove the goal, using the given theorems as additional assumptions in the search.

MESON_TAC performs first order proof using the model elimination algorithm. This algorithm is semi-complete for pure first order logic. It makes special provision for handling polymorphic and higher-order values, and often this is sufficient. It does not handle conditional expressions at all, and these should be eliminated before MESON_TAC is applied.

MESON_TAC works by first converting the problem instance it is given into an internal format where it can do proof search efficiently, without having to do proof search at the level of HOL inference. If a proof is found, this is translated back into applications of HOL inference rules, proving the goal.

The feedback given by MESON_TAC is controlled by the level of the integer reference variable mesonLib.chatting. At level zero, nothing is printed. At the default level of one, a line of dots is printed out as the proof progresses. At all other values for this variable, MESON_TAC is most verbose. If the proof is progressing quickly then it is often worth waiting for it to go quite deep into its search. Once a proof slows down, it is not usually worth waiting for it after it has gone through a few (no more than five or six) levels. (At level one, a "level" is represented by the printing of a single dot.)

Failure

MESON_TAC fails if it searches to a depth equal to the contents of the reference variable mesonLib.max_depth (set to 30 by default, but changeable by the user) without finding a proof. Shouldn't fail otherwise.

MESON_TAC can only progress the goal to a successful proof of the (whole) goal or not at all. In this respect it differs from tactics such as simplification and rewriting. Its ability

to solve existential goals and to make effective use of transitivity theorems make it a particularly powerful tactic.

Comments

The assumptions of a goal are ignored when `MESON_TAC` is applied. To include assumptions use `ASM_MESON_TAC`.

See also

`mesonLib.ASM_MESON_TAC`, `mesonLib.GEN_MESON_TAC`

message_of (Feedback)

```
message_of : hol_error -> string
```

Extract message from a `hol_error`.

An invocation `message_of holerr` returns the message component of `holerr`.

Example

```
> message_of (mk_hol_error "Foo" "bar" locn.Loc_Unknown "bad input");
val it = "bad input": string
```

Failure

Never fails.

See also

`Feedback`, `Feedback.mk_hol_error`, `Feedback.origins_of`, `Feedback.top_structure_of`, `Feedback.top_function_of`

METIS_TAC (bossLib)

```
bossLib.METIS_TAC : thm list -> tactic
```

Performs first-order resolution to try to prove goal

When `METIS_TAC ths` is applied to a goal `(asl,w)`, it attempts to find a resolution proof that the provided theorems in `ths` and the assumptions in `asl` together imply the goal in `w`. `METIS_TAC` implements ordered resolution and as such its ability to reason about equality is generally better than `MESON_TAC`'s.

Failure

Fails if the underlying resolution machinery cannot prove the goal. `METIS_TAC` may also consume more and more time, and more and more memory as a search for a proof proceeds without ever explicitly failing.

Comments

The alternative lower-case spelling `metis_tac` is also available for this tactic from the `bossLib` structure. There is no "metis" entrypoint that allows one to ignore the assumptions (with "meson", there is both `MESON_TAC` and `ASM_MESON_TAC`).

See also

`mesonLib.MESON_TAC`

MK_ABS (Drule)

```
Drule.MK_ABS : (thm -> thm)
```

Abstracts both sides of an equation.

When applied to a theorem `A |- !x. t1 = t2`, whose conclusion is a universally quantified equation, `MK_ABS` returns the theorem `A |- \x. t1 = \x. t2`.

```
      A |- !x. t1 = t2
-------------------------  MK_ABS
 A |- (\x. t1) = (\x. t2)
```

Failure

Fails unless the theorem is a (singly) universally quantified equation.

See also

`Thm.ABS`, `jrhUtils.HALF_MK_ABS`, `Thm.MK_COMB`, `Drule.MK_EXISTS`

mk_abs (Term)

```
Term.mk_abs : term * term -> term
```

Constructs an abstraction.

`mk_abs (v, t)` returns the lambda abstraction `\v. t`. All free occurrences of `v` in `t` thereby become bound.

Failure

Fails if `v` is not a variable.

See also

`Term.dest_abs`, `Term.is_abs`, `boolSyntax.list_mk_abs`, `Term.mk_var`, `Term.mk_const`, `Term.mk_comb`

mk_anylet (pairSyntax)

```
pairSyntax.mk_anylet : (term * term) list * term -> term
```

Constructs arbitrary `let` terms.

The invocation `mk_anylet ([(a1,b1),...,(an,bn)],N)` returns a term of the form `` `LET P Q` ``, which will prettyprint as `let a1 = b1 and ... and an = bn in N`. The internal representation is equal to

```
LET (...(LET (\an ...\a1. N) bn) ...) b1
```

Each `ai` can be a varstruct (a single variable or a tuple of variables), or a function variable applied to a sequence of varstructs. In the usual case, only a single binding is made, i.e., `mk_anylet ([(a,b)],N)`, and the result is equal to `LET (\a. N) b`.

Failure

Fails if the type of any `ai` is not equal to the type of the corresponding `bi`.

Example

```
> strip_comb (pairSyntax.mk_anylet ([(Term`x`, Term`M`)], Term`N x`));
val it = (“LET”, [“λx. N x”, “M”]): term * term list

> pairSyntax.mk_anylet ([(``f (x:'a,y:'b):'c``, ``M:'c``), (``g (z:'c) :'d``, ``N:'
            ``g (f (a:'a,b:'b):'c):'d``);
val it = “let f (x,y) = M and g z = N in g (f (a,b))”: term
```

Programming that involves manipulation of term syntax.

See also

`boolSyntax.mk_let`, `boolSyntax.dest_let`, `boolSyntax.is_let`, `pairSyntax.list_mk_anylet`, `pairSyntax.dest_anylet`

mk_arb (boolSyntax)

```
boolSyntax.mk_arb : hol_type -> term
```

Creates a type instance of the `ARB` constant.

For any HOL type `ty`, `mk_arb ty` creates a type instance of the `ARB` constant.

Failure

Never fails.

Comments

`ARB` is a constant of type `'a`. It is sometimes used for creating pseudo-partial functions.

See also

`boolSyntax.dest_arb`, `boolSyntax.is_arb`, `boolSyntax.arb`

mk_asm (bossLib)

```
bossLib.mk_asm : string -> thm -> tactic
```

Creates a new named assumption with the given name and the given fact.

Failure

Never fails.

See also

`bossLib.asm`, `bossLib.asm_x`

mk_bool_case (boolSyntax)

```
boolSyntax.mk_bool_case : term * term * term -> term
```

Constructs a case expression over `bool`.

`mk_bool_case M1 M2 b` returns `bool_case M1 M2 b`. The prettyprinter displays this as `case b of T -> M1 || F -> M2`. The `bool_case` constant may be thought of as a pattern-matching version of the conditional.

Failure

Fails if `b` is not of type `bool`. Also fails if `M1` and `M2` do not have the same type.

Example

```
> mk_bool_case (Term`f x`,Term`b:'b`,Term`x:bool`);
val it = “if x then f x else b”: term
```

See also

`boolSyntax.dest_bool_case`, `boolSyntax.is_bool_case`

mk_comb (Term)

```
Term.mk_comb : term * term -> term
```

Constructs a combination (function application).

`mk_comb (t1,t2)` returns the combination `t1 t2`.

Failure

Fails if `t1` does not have a function type, or if `t1` has a function type, but its domain does not equal the type of `t2`.

Example

```
- mk_comb (neg_tm,T);
```

```
> val it = `~T` : term

- mk_comb(T, T) handle e => Raise e;

Exception raised at Term.mk_comb:
incompatible types
```

See also

`Term.dest_comb`, `Term.is_comb`, `Term.list_mk_comb`, `Term.mk_var`, `Term.mk_const`, `Term.mk_abs`

MK_COMB (Thm)

```
Thm.MK_COMB : thm * thm -> thm
```

Proves equality of combinations constructed from equal functions and operands.

When applied to theorems `A1 |- f = g` and `A2 |- x = y`, the inference rule `MK_COMB` returns the theorem `A1 u A2 |- f x = g y`.

```
 A1 |- f = g   A2 |- x = y
---------------------------  MK_COMB
    A1 u A2 |- f x = g y
```

Failure

Fails unless both theorems are equational and `f` and `g` are functions whose domain types are the same as the types of `x` and `y` respectively.

See also

`Thm.AP_TERM`, `Thm.AP_THM`, `Tactic.MK_COMB_TAC`

MK_COMB_TAC (Tactic)

```
Tactic.MK_COMB_TAC : tactic
```

Breaks an equality between applications into two equality goals: one for the functions, and other for the arguments.

`MK_COMB_TAC` reduces a goal of the form `A ?- f x = g y` to the goals `A ?- f = g` and `A ?- x = y`.

```
    A ?- f x = g y
   ============================  MK_COMB_TAC
     A ?- f = g,    A ?- x = y
```

Failure

Fails unless the goal is equational, with both sides being applications.

See also

`Thm.MK_COMB`, `Thm.AP_TERM`, `Thm.AP_THM`, `Tactic.AP_TERM_TAC`, `Tactic.AP_THM_TAC`

mk_cond (boolSyntax)

```
boolSyntax.mk_cond : term * term * term -> term
```

Constructs a conditional term.

`mk_cond(t,t1,t2)` constructs an application `COND t t1 t2`. This is rendered by the prettyprinter as `if t then t1 else t2`.

Failure

Fails if `t` is not of type `bool` or if `t2` and `t2` are of different types.

Comments

The prettyprinter can be trained to print `if t then t1 else t2` as `t => t1 | t2`.

See also

`boolSyntax.dest_cond`, `boolSyntax.is_cond`

mk_conj (boolSyntax)

```
boolSyntax.mk_conj : term * term -> term
```

Constructs a conjunction.

`mk_conj(t1, t2)` returns the term `t1 /\ t2`.

Failure

Fails if `t1` and `t2` do not both have type `bool`.

See also

`boolSyntax.dest_conj`, `boolSyntax.is_conj`, `boolSyntax.list_mk_conj`, `boolSyntax.strip_conj`

mk_cons (listSyntax)

```
listSyntax.mk_cons : term * term -> term
```

Constructs a `CONS` pair.

`mk_cons (``t``, ``[t1;...;tn]``)` returns ``[t;t1;...;tn]``.

Failure

Fails if the second element is not a list or if the first element is not of the same type as the elements of the list.

See also

`listSyntax.dest_cons`, `listSyntax.is_cons`, `listSyntax.mk_list`, `listSyntax.dest_list`, `listSyntax.is_list`

mk_const (Term)

```
Term.mk_const : string * hol_type -> term
```

Constructs a constant.

If `n` is a string that has been previously declared to be a constant with type `ty` and `ty1` is an instance of `ty`, then `mk_const(n,ty1)` returns the specified instance of the constant.

(A type `ty1` is an 'instance' of a type `ty2` when `match_type ty2 ty1` does not fail.)

Note, however, that constants with the same name (and type) may be declared in different theories. If two theories having constants with the same name `n` are in the ancestry of the current theory, then `mk_const(n,ty)` will issue a warning before arbitrarily selecting which constant to construct. In such situations, `mk_thy_const` allows one to specify exactly which constant to use.

Failure

Fails if `n` is not the name of a known constant, or if `ty` is not an instance of the type that the constant has in the signature.

Example

```
- mk_const ("T", ":bool");
> val it = `T` : term

- mk_const ("=", ":bool -> bool -> bool");
```

```
> val it = `$=` : term

- try mk_const ("test", ":bool");
Exception raised at Term.mk_const:
test not found
```

The following example shows a new constant being introduced that has the same name as the standard equality of HOL. Then we attempt to make an instance of that constant.

```
- new_constant ("=", ":bool -> bool -> bool");
> val it = () : unit

- mk_const("=", ":bool -> bool -> bool");
<<HOL warning: Term.mk_const: "=": more than one possibility>>

> val it = `$=` : term
```

See also

`Term.mk_thy_const`, `Term.dest_const`, `Term.is_const`, `Term.mk_var`, `Term.mk_comb`, `Term.mk_abs`, `Type.match_type`

mk_disj (boolSyntax)

```
boolSyntax.mk_disj : term * term -> term
```

Constructs a disjunction.

If `t1` and `t2` are terms, both of type `bool`, then `mk_disj(t1,t2)` returns the term `t1 \/ t2`.

Failure

Fails if `t1` or `t2` does not have type `bool`.

See also

boolSyntax.dest_disj, boolSyntax.is_disj, boolSyntax.list_mk_disj, boolSyntax.strip_disj

mk_eq (boolSyntax)

```
boolSyntax.mk_eq : term * term -> term
```

Constructs an equation.

`mk_eq(t1, t2)` returns the term `t1 = t2`.

Failure

Fails if the type of `t1` is not equal to that of `t2`.

See also

boolSyntax.dest_eq, boolSyntax.is_eq

MK_EXISTS (Drule)

```
Drule.MK_EXISTS : (thm -> thm)
```

Existentially quantifies both sides of a universally quantified equational theorem.

When applied to a theorem `A |- !x. t1 = t2`, the inference rule `MK_EXISTS` returns the theorem `A |- (?x. t1) = (?x. t2)`.

```
     A |- !x. t1 = t2
--------------------------  MK_EXISTS
 A |- (?x. t1) = (?x. t2)
```

Failure

Fails unless the theorem is a singly universally quantified equation.

See also

`Thm.AP_TERM`, `Drule.EXISTS_EQ`, `Thm.GEN`, `Drule.LIST_MK_EXISTS`, `Drule.MK_ABS`

mk_exists (boolSyntax)

```
boolSyntax.mk_exists : term * term -> term
```

Term constructor for existential quantification.

If `v` is a variable and `t` is a term of type `bool`, then `mk_exists (v,t)` returns the term `?v. t`.

Failure

Fails if `v` is not a variable or if `t` is not of type `bool`.

See also

`boolSyntax.dest_exists`, `boolSyntax.is_exists`, `boolSyntax.list_mk_exists`, `boolSyntax.strip_exists`

mk_exists1 (boolSyntax)

```
boolSyntax.mk_exists1 : term * term -> term
```

Term constructor for unique existence.

If `v` is a variable and `t` is a term of type `bool`, then `mk_exists1 (v,t)` returns the term `?!v. t`.

Failure

Fails if `v` is not a variable or if `t` is not of type `bool`.

See also

`boolSyntax.dest_exists1`, `boolSyntax.is_exists1`

mk_forall (boolSyntax)

```
boolSyntax.mk_forall : term * term -> term
```

Term constructor for universal quantification.

If `v` is a variable and `t` is a term of type `bool`, then `mk_forall (v,t)` returns the term `!v. t`.

Failure

Fails if `v` is not a variable or if `t` is not of type `bool`.

See also

`boolSyntax.dest_forall`, `boolSyntax.is_forall`, `boolSyntax.list_mk_forall`, `boolSyntax.st`
`forall`

mk_HOL_ERR (Feedback)

```
Feedback.mk_HOL_ERR : string -> string -> string -> exn
```

Creates an application of `HOL_ERR`.

`mk_HOL_ERR` provides a curried version of the `HOL_ERR` exception.

Failure

Never fails.

Example

```
> mk_HOL_ERR "<Module>" "<function>" "<message>";
val it = HOL_ERR (at <Module>.<function>: <message>): exn
```

See also

`Feedback`, `Feedback.HOL_ERR`, `Feedback.mk_HOL_ERRloc`

mk_HOL_ERRloc (Feedback)

```
Feedback.mk_HOL_ERRloc : string -> string -> locn.locn -> string -> exn
```

Creates an application of `HOL_ERR` with location information.

`mk_HOL_ERRloc` provides a curried version of the `HOL_ERR` exception including spcified location information.

Failure

Never fails.

Example

```
> val exloc = let open locn in Loc (LocA(0,0), LocA(3,42)) end;
val exloc = 1:0-4:42: locn.locn

> mk_HOL_ERRloc "<Module>" "<function>" exloc "<message>";
val it =
   HOL_ERR
     (at <Module>.<function>:
          between line 1, character 0 and line 4, character 42: <message>):
   exn
```

See also

Feedback, Feedback.HOL_ERR, Feedback.mk_HOL_ERR

mk_hol_error (Feedback)

```
mk_hol_error : string -> string -> locn.locn -> string -> hol_error
```

Create a hol_error value.

An invocation mk_hol_error sname fname loc m builds an initial hol_error value where sname is the enclosing module name, fname is the enclosing function name, loc is the location in the file where the error is detected, and m is the error message.

hol_error values are used to build HOL_ERR exceptions.

Example

```
> val holerr =
    mk_hol_error "Foo" "bar" locn.Loc_Unknown "unexpected input"
val holerr = at Foo.bar: unexpected input: hol_error

> val holexn = HOL_ERR holerr;
val holexn = HOL_ERR (at Foo.bar: unexpected input): exn
```

Representation

The representation of the hol_error type includes

- a message
- a list of origin elements.

Each origin value includes source information such as the enclosing SML module and function, plus the source location where the error originated. The list of origins is typically used as a stack, and can be used to provide a backtrace facility. The function wrap_hol_error is used to push an origin onto the existing origins of a hol_error, however it is more common to use wrap_exn on HOL_ERR values.

See also

Feedback, Feedback.HOL_ERR, Feedback.origins_of, Feedback.message_of, Feedback.top_structure_of, Feedback.top_function_of, Feedback.top_location_of, Feedback.set_top_function, Feedback.set_message, Feedback.wrap_hol_error

mk_icomb (boolSyntax)

```
boolSyntax.mk_icomb : term * term -> term
```

Forms an application term, possibly instantiating the function.

A call to `mk_icomb(f,x)` checks to see if the term `f`, which must have function type, can have any of its type variables instantiated so as to make the domain of the function match the type of `x`. If so, then the call returns the application of the instantiated `f` to `x`.

Failure

Fails if there is no way to instantiate the function term to make its domain match the argument's type.

Example

Note how both the S combinator and the argument have type variables invented for them when the two quotations are parsed.

```
- val t = mk_icomb(``S``, ``\n:num b. (n,b)``);
<<HOL message: inventing new type variable names: 'a, 'b, 'c>>
<<HOL message: inventing new type variable names: 'a>>
> val t = ``S (\n b. (n,b))`` : term
```

The resulting term `t` has only the type variable `:'a` left after instantiation.

```
- type_of t;
> val it = ``:(num -> 'a) -> num -> num # 'a`` : hol_type
```

This term can now be combined with an argument and the final type variable instantiated:

```
- mk_icomb(t, ``ODD``);
> val it = ``S (\n b. (n,b)) ODD`` : term

- type_of it;
> val it = ``:num -> num # bool``;
```

Attempting to use mk_comb above results in immediate error because it requires domain and arguments types to be identical:

```
- mk_comb(``S``, ``\n:num b. (n,b)``) handle e => Raise e;
<<HOL message: inventing new type variable names: 'a, 'b, 'c>>
<<HOL message: inventing new type variable names: 'a>>

Exception raised at Term.mk_comb:
incompatible types
! Uncaught exception:
! HOL_ERR
```

See also

boolSyntax.list_mk_icomb, Term.mk_comb

mk_imp (boolSyntax)

```
boolSyntax.mk_imp : term * term -> term
```

Constructs an implication.

If t1 and t2 are terms of type bool, then mk_imp(t1,t2) constructs the term t1 ==> t2.

Failure

Fails if t1 and t2 are not both of type bool.

See also

boolSyntax.dest_imp, boolSyntax.dest_imp_only, boolSyntax.is_imp, boolSyntax.is_imp_only, boolSyntax.list_mk_imp

mk_istream (Lib)

```
Lib.mk_istream : ('a -> 'a) -> 'a -> ('a -> 'b) -> ('a,'b) istream
```

Create a stream.

An application `mk_istream trans init proj` creates an imperative stream of elements. The stream is generated by applying `trans` to the state. The first element in the stream state is `init`. The value of the state is obtained by applying `proj`.

Failure

If an application of `trans` or `proj` fails when applied to the state.

Example

The following creates a stream of distinct strings.

```
- mk_istream (fn x => x+1) 0 (concat "gsym" o int_to_string);
> val it = <istream> : (int, string) istream
```

Comments

It is aesthetically unpleasant that the underlying implementation type is visible.

See any book on ML programming to see how functional streams are built.

See also

`Lib.next`, `Lib.state`, `Lib.reset`

mk_let (boolSyntax)

```
boolSyntax.mk_let : term * term -> term
```

Constructs a `let` term.

The invocation `mk_let (M,N)` returns the term `` `LET M N` ``. If `M` is of the form `\x.t` then the result will be pretty-printed as `let x = N in t`. Since `LET M N` is defined to be `M N`, one can think of a `let`-expression as a suspended beta-redex (if that helps).

Failure

Fails if the types of `M` and `N` are such that `LET M N` is not well-typed, i.e., the type of `M` must be a function type, and the type of `N` must equal the domain of the type of `M`.

Example

```
> mk_let(Term`\x. x \/ x`, Term`Q /\ R`);
val it = “let x = (Q ∧ R) in x ∨ x”: term
```

Comments

`let` expressions may be nested.

Pairing can also be used in the `let` syntax, provided `pairTheory` has been loaded. The library `pairLib` provides support for manipulating ‘paired’ lets.

See also

`boolSyntax.dest_let`, `boolSyntax.is_let`, `pairSyntax.mk_anylet`

mk_list (listSyntax)

```
listSyntax.mk_list : term list * hol_type -> term
```

Constructs an object-level (HOL) list from an ML list of terms.

`mk_list ([t1, ..., tn], ty)` returns `[t1;...;tn]:ty list`. The type argument is required so that empty lists can be constructed.

Failure

Fails if any term in the list is not of the type specified as the second argument.

See also

`listSyntax.dest_list`, `listSyntax.is_list`, `listSyntax.mk_cons`, `listSyntax.dest_cons`, `listSyntax.is_cons`

mk_neg (boolSyntax)

```
boolSyntax.mk_neg : (term -> term)
```

Constructs a negation.

`mk_neg "t"` returns `"~t"`.

Failure

Fails with `mk_neg` unless `t` is of type `bool`.

See also

`boolSyntax.dest_neg`, `boolSyntax.is_neg`

mk_numeral (numSyntax)

```
numSyntax.mk_numeral : Arbnum.num -> term
```

Convert ML bignum value to HOL numeral.

An invocation `mk_numeral n`, where `n` is an ML value of type `Arbnum.num` returns the corrresponding HOL term.

Example

```
> Arbnum.fromString "1234";
val it = 1234: num

> numSyntax.mk_numeral it;
val it = “1234”: term
```

Failure

Never fails.

See also

`numSyntax.dest_numeral`, `numSyntax.is_numeral`

mk_oracle_thm (Thm)

```
Thm.mk_oracle_thm : string -> term list * term -> thm
```

Construct a theorem without proof, and tag it.

In principle, nearly every theorem of interest can be proved in HOL by using only the axioms and primitive rules of inference. The use of ML to orchestrate larger inference steps from the primitives, along with support in HOL for goal-directed proof, considerably eases the task of formal proof. Nearly every theorem of interest can therefore be produced as the end product of a chain of primitive inference steps, and HOL implementations strive to keep this purity.

However, it is occasionally useful to interface HOL with trusted external tools that also produce, in some sense, theorems that would be derivable in HOL. It is clearly a burden to require that HOL proofs accompany such theorems so that they can be (re-)derived in HOL. In order to support greater interoperation of proof tools, therefore, HOL provides the notion of a ‘tagged’ theorem.

A tagged theorem is manufactured by invoking `mk_oracle_thm tag (A,w)`, where `A` is a list of HOL terms of type `bool`, and `w` is also a HOL term of boolean type. No proof

is done; the sequent is merely injected into the type of theorems, and the `tag` value is attached to it. The result is the theorem `A |- w`.

The `tag` value stays with the theorem, and it propagates in a hereditary fashion to any theorem derived from the tagged theorem. Thus, if one examines a theorem with `Thm.tag` and finds that it has no tag, then the theorem has been derived purely by proof steps in the HOL logic. Otherwise, shortcuts have been taken, and the external tools, also known as 'oracles', used to make the shortcuts are signified by the tags.

Failure

If some element of `A` does not have type `bool`, or `w` does not have type `bool`, or the tag string doesn't represent a valid tag (which occurs if it is the string `"DISK_THM"`, or if it is a string containing unprintable characters).

Example

In the following, we construct a tag and then make a rogue rule of inference.

```
> val tag = "SimonSays";
val tag = "SimonSays": string

> val SimonThm = mk_oracle_thm tag;
val SimonThm = fn: term list * term -> thm

> val th = SimonThm ([], Term `!x. x`);
val th = ⊢ ∀x. x: thm

> val th1 = SPEC F th;
val th1 = ⊢ F: thm

> (show_tags := true; th1);
val it = [oracles: SimonSays] [axioms: ] [] ⊢ F: thm
```

Tags accumulate in a manner similar to logical hypotheses.

```
> CONJ th1 th1;
val it = [oracles: SimonSays] [axioms: ] [] ⊢ F ∧ F: thm

> val SerenaThm = mk_oracle_thm "Serena";
```

```
val SerenaThm = fn: term list * term -> thm

> CONJ th1 (SerenaThm ([],T));
val it = [oracles: Serena, SimonSays] [axioms: ] [] ⊢ F ∧ T: thm
```

Comments

It is impossible to detach a tag from a theorem.

See also

`Thm.add_tag`, `Thm.mk_thm`, `Tag.read`, `Thm.tag`

MK_PABS (PairRules)

```
PairRules.MK_PABS : (thm -> thm)
```

Abstracts both sides of an equation.

When applied to a theorem `A |- !p. t1 = t2`, whose conclusion is a paired universally quantified equation, `MK_PABS` returns the theorem `A |- (\p. t1) = (\p. t2)`.

```
     A |- !p. t1 = t2
-------------------------  MK_PABS
 A |- (\p. t1) = (\p. t2)
```

Failure

Fails unless the theorem is a (singly) paired universally quantified equation.

See also

`Drule.MK_ABS`, `PairRules.PABS`, `PairRules.HALF_MK_PABS`, `PairRules.MK_PEXISTS`

mk_pabs (pairSyntax)

```
pairSyntax.mk_pabs : term * term -> term
```

Constructs a paired abstraction.

If `M` is the tuple `(v1,..(..)..,vn)`, and `N` is an arbitrary term, then `mk_pabs (M,N)` returns the paired abstraction `` `\(v1,..(..)..,vn).N` ``.

Failure

Fails unless `M` is an arbitrarily nested pair composed from variables, with no repetitions of variables.

See also

`pairSyntax.dest_pabs`, `pairSyntax.is_pabs`, `Term.mk_abs`

MK_PAIR (PairRules)

```
PairRules.MK_PAIR : thm * thm -> thm
```

Proves equality of pairs constructed from equal components.

When applied to theorems `A1 |- a = x` and `A2 |- b = y`, the inference rule `MK_PAIR` returns the theorem `A1 u A2 |- (a,b) = (x,y)`.

```
 A1 |- a = x   A2 |- b = y
--------------------------  MK_PAIR
 A1 u A2 |- (a,b) = (x,y)
```

Failure

Fails unless both theorems are equational.

mk_pair (pairSyntax)

```
pairSyntax.mk_pair : term * term -> term
```

Constructs object-level pair from a pair of terms.

`mk_pair (t1,t2)` returns `(t1,t2)`.

Failure

Never fails.

See also

`pairSyntax.dest_pair`, `pairSyntax.is_pair`, `pairSyntax.list_mk_pair`

MK_PEXISTS (PairRules)

```
PairRules.MK_PEXISTS : (thm -> thm)
```

Existentially quantifies both sides of a universally quantified equational theorem.

When applied to a theorem `A |- !p. t1 = t2`, the inference rule `MK_PEXISTS` returns the theorem `A |- (?x. t1) = (?x. t2)`.

```
     A |- !p. t1 = t2
--------------------------  MK_PEXISTS
 A |- (?p. t1) = (?p. t2)
```

Failure

Fails unless the theorem is a singly paired universally quantified equation.

See also

`PairRules.PEXISTS_EQ`, `PairRules.PGEN`, `PairRules.LIST_MK_PEXISTS`, `PairRules.MK_PABS`

MK_PFORALL (PairRules)

```
PairRules.MK_PFORALL : (thm -> thm)
```

Universally quantifies both sides of a universally quantified equational theorem.

When applied to a theorem `A |- !p. t1 = t2`, the inference rule `MK_PFORALL` returns the theorem `A |- (!x. t1) = (!x. t2)`.

```
      A |- !p. t1 = t2
--------------------------  MK_PFORALL
 A |- (!p. t1) = (!p. t2)
```

Failure

Fails unless the theorem is a singly paired universally quantified equation.

See also

`PairRules.PFORALL_EQ`, `PairRules.LIST_MK_PFORALL`, `PairRules.MK_PABS`

mk_primed_var (Term)

```
Term.mk_primed_var : string * hol_type -> term
```

Primes a variable name sufficiently to make it distinct from all constants.

When applied to a record made from a string `v` and a type `ty`, the function `mk_primed_var` constructs a variable whose name consists of `v` followed by however many primes are necessary to make it distinct from any constants in the current theory.

Failure

Never fails.

Example

```
> new_theory "wombat";
val it = (): unit

> mk_primed_var("x", bool);
val it = “x”: term

> new_constant("x", alpha);
val it = (): unit

> mk_primed_var("x", bool);
val it = “x'”: term
```

See also

`Term.genvar`, `Term.variant`

mk_prod (pairSyntax)

```
pairSyntax.mk_prod : hol_type * hol_type -> hol_type
```

Constructs a product type from two constituent types.

`mk_prod(ty1, ty2)` returns `ty1 # t2`.

Failure

Never fails.

See also

`pairSyntax.is_prod`, `pairSyntax.dest_prod`

MK_PSELECT (PairRules)

```
PairRules.MK_PSELECT : (thm -> thm)
```

Quantifies both sides of a universally quantified equational theorem with the choice quantifier.

When applied to a theorem `A |- !p. t1 = t2`, the inference rule `MK_PSELECT` returns the theorem `A |- (@x. t1) = (@x. t2)`.

```
      A |- !p. t1 = t2
  -------------------------  MK_PSELECT
   A |- (@p. t1) = (@p. t2)
```

Failure

Fails unless the theorem is a singly paired universally quantified equation.

See also

`PairRules.PSELECT_EQ`, `PairRules.MK_PABS`

mk_ptree (patriciaLib)

```
patriciaLib.mk_ptree : term_ptree -> term
```

Term constructor for Patricia trees.

The constructor `mk_ptree` will return a HOL term that corresponds with the supplied ML Patricia tree. The ML abstract data type `term_ptree` is defined in `patriciaLib`.

Failure

The conversion will fail if the terms stored in the supplied Patricia tree do not all have the same type.

Example

```
> patriciaLib.mk_ptree (patriciaLib.int_ptree_of_list [(1,``T``), (2, ``2``)]);
Exception- HOL_ERR at patriciaSyntax.mk_branch: raised

> patriciaLib.mk_ptree (patriciaLib.int_ptree_of_list [(1,``1``), (2, ``2``)]);
val it = “Branch 0 0 (Leaf 1 1) (Leaf 2 2)”: term
```

Comments

When working with large trees it is a good idea constrain term printing by setting Globals.max_print_depth.

See also

`patriciaLib.dest_ptree`, `patriciaLib.is_ptree`

mk_res_abstract (res_quanLib)

```
res_quanLib.mk_res_abstract : (term # term # term) -> term
```

Term constructor for restricted abstraction.

`mk_res_abstract("var","P","t")` returns `"\var :: P . t"`.

Failure

Fails with `mk_res_abstract` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also

`res_quanLib.dest_res_abstract`, `res_quanLib.is_res_abstract`

mk_res_exists (res_quanLib)

```
res_quanLib.mk_res_exists : ((term # term # term) -> term)
```

Term constructor for restricted existential quantification.

`mk_res_exists("var","P","t")` returns `"?var :: P . t"`.

Failure

Fails with `mk_res_exists` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also

`res_quanLib.dest_res_exists`, `res_quanLib.is_res_exists`, `res_quanLib.list_mk_res_exists`

mk_res_exists_unique (res_quanLib)

```
res_quanLib.mk_res_exists_unique : (term # term # term) -> term
```

Term constructor for restricted unique existential quantification.

`mk_res_exists_unique ("var","P","t")` returns `"?!var :: P . t"`.

Failure

Fails with `mk_res_exists_unique` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also

`res_quanLib.dest_res_exists_unique`, `res_quanLib.is_res_exists_unique`

mk_res_forall (res_quanLib)

```
res_quanLib.mk_res_forall : (term # term # term) -> term
```

Term constructor for restricted universal quantification.

`mk_res_forall("var","P","t")` returns `"!var :: P . t"`.

Failure

Fails with `mk_res_forall` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also

`res_quanLib.dest_res_forall`, `res_quanLib.is_res_forall`, `res_quanLib.list_mk_res_forall`

mk_res_select (res_quanLib)

```
res_quanLib.mk_res_select : (term # term # term) -> term
```

Term constructor for restricted choice quantification.

`mk_res_select("var","P","t")` returns `"@var :: P . t"`.

Failure

Fails with `mk_res_select` if the first term is not a variable or if `P` and `t` are not of type `":bool"`.

See also

`res_quanLib.dest_res_select`, `res_quanLib.is_res_select`

mk_select (boolSyntax)

```
boolSyntax.mk_select : term * term -> term
```

Constructs a choice-term.

If `v` is a variable and `t` is a term of type `bool`, then `mk_select (v,t)` returns `@var. t`.

Failure

Fails if `v` is not a variable or if `t` is not of type `bool`.

See also

`boolSyntax.dest_select`, `boolSyntax.is_select`

mk_set (Lib)

```
Lib.mk_set : ''a list -> ''a list
```

Transforms a list into one with distinct elements.

An invocation `mk_set list` returns a list consisting of the distinct members of `list`. In particular, the result list has no repeated elements.

Failure

Never fails.

Example

```
> mk_set [1,1,1,2,2,2,3,3,4];
val it = [1, 2, 3, 4]: int list
```

Comments

In some programming situations, it is convenient to implement sets by lists, in which case `mk_set` may be helpful. However, such an implementation is only suitable for small sets.

A high-performance implementation of finite sets may be found in structure `HOLset`.

ML equality types are used in the implementation of `mk_set` and its kin. This limits its applicability to types that allow equality. For other types, typically abstract ones, use the 'op_' variants.

See also

`Lib.op_mk_set`, `Lib.mem`, `Lib.insert`, `Lib.union`, `Lib.U`, `Lib.set_diff`, `Lib.subtract`, `Lib.intersect`, `Lib.null_intersection`, `Lib.set_eq`

mk_simpset (simpLib)

```
simpLib.mk_simpset : ssfrag list -> simpset
```

Creates a simpset by combining a list of `ssfrag` values.

This function creates a `simpset` value by repeatedly adding (as per the ++ operator) `simpset` fragment values to the base `empty_ss`.

Failure

Never fails.

Creates simpsets, which are a necessary argument to any simplification function.

See also

`simpLib.++`, `simpLib.rewrites`, `simpLib.SIMP_CONV`

mk_thm (Thm)

```
Thm.mk_thm : term list * term -> thm
```

Creates an arbitrary theorem (dangerous!).

The function `mk_thm` can be used to construct an arbitrary theorem. It is applied to a pair consisting of the desired assumption list (possibly empty) and conclusion. All the terms therein should be of type `bool`.

```
mk_thm([a1,...,an],c) = ({a1,...,an} |- c)
```

`mk_thm` is an application of `mk_oracle_thm`, and every application of it tags the resulting theorem with `MK_THM`.

Failure

Fails unless all the terms provided for assumptions and conclusion are of type `bool`.

Example

The following shows how to create a simple contradiction:

```
> val falsity = mk_thm([],boolSyntax.F);
val falsity = ⊢ F: thm

> Globals.show_tags := true;
val it = (): unit

> falsity;
val it = [oracles: MK_THM] [axioms: ] [] ⊢ F: thm
```

Comments

Although `mk_thm` can be useful for experimentation or temporarily plugging gaps, its use should be avoided if at all possible in important proofs, because it can be used to create

theorems leading to contradictions. The example above is a trivial case, but it is all too easy to create a contradiction by asserting 'obviously sound' theorems.

All theorems which are likely to be needed can be derived using only HOL's inbuilt axioms and primitive inference rules, which are provably sound (see the DESCRIPTION). Basing all proofs, normally via derived rules and tactics, on just these axioms and inference rules gives proofs which are (apart from bugs in HOL or the underlying system) completely secure. This is one of the great strengths of HOL, and it is foolish to sacrifice it to save a little work.

Because of the way tags are propagated during proof, a theorem proved with the aid of `mk_thm` is detectable by examining its tag.

See also

`Theory.new_axiom`, `Thm.mk_oracle_thm`, `Thm.tag`, `Globals.show_tags`

mk_thy_const (Term)

```
Term.mk_thy_const : {Thy:string, Name:string, Ty:hol_type} -> term
```

Constructs a constant.

If `n` is a string that has been previously declared to be a constant with type `ty` in theory `thy`, and `ty1` is an instance of `ty`, then `mk_thy_const{Name=n, Thy=thy, Ty=ty1}` returns the specified instance of the constant.

(A type `ty1` is an 'instance' of a type `ty2` when `match_type ty2 ty1` does not fail.)

Failure

Fails if `n` is not the name of a constant in theory `thy`, if `thy` is not in the ancestry of the current theory, or if `ty1` is not an instance of `ty`.

Example

```
- mk_thy_const {Name="T", Thy="bool", Ty=bool};
> val it = `T` : term
```

```
- try mk_thy_const {Name = "bar", Thy="foo", Ty=bool};
Exception raised at Term.mk_thy_const:
"foo$bar" not found
```

See also

`Term.dest_thy_const`, `Term.mk_const`, `Term.dest_const`, `Term.is_const`, `Term.mk_var`, `Term.mk_comb`, `Term.mk_abs`, `Type.match_type`

mk_thy_type (Type)

```
Type.mk_thy_type
       : {Thy:string, Tyop:string, Args:hol_type list} -> hol_type
```

Constructs a type.

If `s` is a string that has been previously declared to be a type with arity type `n` in theory `thy`, and the length of `tyl` is equal to `n`, then `mk_thy_type{Tyop=s, Thy=thy, Args=tyl}` returns the requested compound type.

Failure

Fails if `s` is not the name of a type in theory `thy`, if `thy` is not in the ancestry of the current theory, or if `n` is not the length of `tyl`.

Example

```
> mk_thy_type {Tyop="fun", Thy="min", Args = [alpha,bool]};
val it = ":α -> bool": hol_type

> try mk_thy_type {Tyop="bar", Thy="foo", Args = []};
Exception- HOL_ERR
  (at Type.mk_thy_type:
       the type operator "bar" has not been declared in theory "foo".) raised
```

Comments

In general, mk_thy_type is to be preferred over mk_type because HOL provides a fresh namespace for each theory (mk_type is a holdover from a time when there was only one namespace shared by all theories).

See also

Type.mk_type, Type.dest_thy_type, Term.mk_const, Term.mk_thy_const

mk_type (Type)

```
Type.mk_type : string * hol_type list -> hol_type
```

Constructs a compound type.

mk_type(tyop,[ty1,...,tyn]) returns the HOL type (ty1,...,tyn)tyop, provided tyop is the name of a known n-ary type constructor.

Failure

Fails if tyop is not the name of a known type, or if tyop is known, but the length of the list of argument types is not equal to the arity of tyop.

Example

```
> mk_type ("bool",[]);
val it = ":bool": hol_type

> mk_type ("fun",[alpha,it]);
val it = ":α -> bool": hol_type
```

Comments

Note that type operators with the same name (and arity) may be declared in different theories. If two theories having type operators with the same name s are in the ancestry of

the current theory, then `mk_type(s,tyl)` will issue a warning before arbitrarily selecting which type operator to use. In such situations, it is preferable to use `mk_thy_type` since it allows one to specify exactly which type operator to use.

See also

`Type.mk_thy_type`, `Type.dest_type`, `Type.mk_vartype`, `Type.-->`

mk_ucomb (boolSyntax)

```
boolSyntax.mk_ucomb : term * term -> term
```

Forms an application term, possibly instantiating both the function and the argument types.

A call to `mk_ucomb(f,x)` checks to see if the term `f` (which must have a function type) can have its type variables instantiated to make the domain of the function match some instantiation of the type of `x`. If so, then the call returns the application of the instantiated `f` to the instantiated `x`.

Failure

Fails if there is no way to instantiate the types to make the function domain match the argument type.

Example

Note how both the FOLDR combinator and the argument (the K combinator) have type variables invented for them when the two quotations are parsed.

```
   - val t = mk_ucomb(``FOLDR``, ``K``);
<<HOL message: inventing new type variable names: 'a, 'b>>
<<HOL message: inventing new type variable names: 'a, 'b>>
   > val t = ``FOLDR K`` : term
```

The resulting term `t` has only the type variable `:'a` left after instantiation.

```
- type_of t;
> val it = ``:'a -> 'a list -> 'a`` : hol_type
```

This term can now be combined with an argument and the final type variable instantiated:

```
- mk_ucomb(t, ``T``);
> val it = ``FOLDR K T`` : term

- type_of it;
> val it = ``:bool list -> bool``;
```

Attempting to use `mk_icomb` in the first example above results in immediate error because it can only instantiate the function type:

```
- mk_icomb(``FOLDR``, ``K``) handle e => Raise e;
<<HOL message: inventing new type variable names: 'a, 'b>>
<<HOL message: inventing new type variable names: 'a, 'b>>

Exception raised at HolKernel.list_mk_icomb:
double bind on type variable 'b
Exception-
   HOL_ERR
     {message = "double bind on type variable 'b", origin_function =
      "list_mk_icomb", origin_structure = "HolKernel"} raised
```

However it can be used in the second example, as only the function type requires instantiation:

```
- mk_icomb(t, ``T``);
> val it = ``FOLDR K T`` : term
```

Comments

`mk_ucomb` makes use of `sep_type_unify`.

See also

`boolSyntax.list_mk_ucomb`, `boolSyntax.sep_type_unify`, `Term.mk_icomb`, `Term.mk_comb`

MK_USING (markerSyntax)

```
markerSyntax.MK_USING : thm -> thm
```

Encodes a theorem so it can be ASSUME_TAC-ed into a goal.

A call to `MK_USING th` encodes `th` in such a way that it can safely be the argument to ASSUME_TAC, and detectable by other tactics operating on the resulting goal as a theorem that might be worthy of notice.

Failure

Fails if the theorem has no hypotheses, is polymorphic, and cannot be found by reverse lookup in the theorem database (using `DB.revlookup`).

See also

`bossLib.using`

mk_var (Term)

```
Term.mk_var : string * hol_type -> term
```

Constructs a variable of given name and type.

If `v` is a string and `ty` is a HOL type, then `mk_var(v, ty)` returns a HOL variable.

Failure

Never fails.

Comments

`mk_var` can be used to construct variables with names which are not acceptable to the term parser. In particular, a variable with the name of a known constant can be constructed using `mk_var`.

See also

Term.dest_var, Term.is_var, Term.mk_const, Term.mk_comb, Term.mk_abs

mk_vartype (Type)

```
Type.mk_vartype : string -> hol_type
```

Constructs a type variable of the given name.

Failure

Fails if the string does not begin with '.

Example

```
> mk_vartype "'giraffe";
val it = “:'giraffe”: hol_type

> try mk_vartype "test";
val it = “:test”: hol_type
```

See also

Type.dest_vartype, Type.is_vartype, Type.mk_type

mk_word_size (wordsLib)

```
wordsLib.mk_word_size : int -> unit
```

Adds a type abbreviation and theorems for a given word length.

An invocation of mk_word_size n introduces a type abbreviation for words of length n. Theorems for dimindex(:n), dimword(:n) and INT_MIN(:n) are generated and stored.

Example

```
> wordsLib.mk_word_size 128
val it = (): unit
> “:word128”
val it = “:word128”: hol_type
> theorem "dimword_128"
val it = ⊢ dimword (:128) = 340282366920938463463374607431768211456: thm
```

Comments

The type abbreviation will only print when `type_pp.pp_array_types` is set to `false`.

See also

`Parse.type_abbrev`, `wordsLib.SIZES_CONV`, `wordsLib.SIZES_ss`

mlquote (Lib)

```
Lib.mlquote : string -> string
```

Put quotation marks around a string.

Like `quote`, `mlquote s` puts quotation marks around a string. However, it also transforms the characters in a string so that, when printed, it would be a valid ML lexeme.

Failure

Never fails

Example

```
> quote "foo\nbar";
val it = "val it = foo \nbar ; \n": string
```

```
> mlquote "foo\nbar";
val it = "\"foo\\nbar\"": string
```

See also

Lib.quote

MOD_CONV (reduceLib)

```
reduceLib.MOD_CONV : conv
```

Calculates by inference the remainder after dividing one numeral by another.

If m and n are numerals (e.g. 0, 1, 2, 3,...), then MOD_CONV "m MOD n" returns the theorem:

```
   |- m MOD n = s
```

where s is the numeral that denotes the remainder after dividing, with truncation, the natural number denoted by m by the natural number denoted by n.

Failure

MOD_CONV tm fails unless tm is of the form "m MOD n", where m and n are numerals, or if n denotes zero.

Example

```
#MOD_CONV "0 MOD 0";;
evaluation failed     MOD_CONV

#MOD_CONV "0 MOD 12";;
|- 0 MOD 12 = 0

#MOD_CONV "2 MOD 0";;
evaluation failed     MOD_CONV
```

```
#MOD_CONV "144 MOD 12";;
|- 144 MOD 12 = 0

#MOD_CONV "7 MOD 2";;
|- 7 MOD 2 = 1
```

monitoring (computeLib)

```
computeLib.monitoring : (term -> bool) option ref
```

Monitoring support for evaluation.

The reference variable `monitoring` provides a simple way to view the operation of `EVAL`, `EVAL_RULE`, and `EVAL_TAC`. The initial value of `monitoring` is `NONE`. If one wants to monitor the expansion of a function, defined with constant `c`, then setting `monitoring` to `SOME (same_const c)` will tell the system to print out the expansion of `c` by the evaluation entrypoints. To monitor the expansions of a collection of functions, defined with `c1`,...,`cn`, then `monitoring` can be set to

```
SOME (fn x => same_const c1 x orelse ... orelse same_const cn x)
```

Failure

Never fails.

Example

```
> val [FACT] = decls "FACT";
val FACT = “FACT”: term

> computeLib.monitoring := SOME (same_const FACT);
val it = (): unit

> EVAL (Term `FACT 4`);
FACT 4 = 4 * FACT 3
FACT 3 = 3 * FACT (PRE 3)
```

```
FACT 2 = 2 * FACT 1
FACT 1 = 1 * FACT (PRE 1)
FACT 0 = 1
val it = ⊢ FACT 4 = 24: thm
```

See also

computeLib.RESTR_EVAL_CONV, Term.decls

MP (Thm)

```
Thm.MP : thm -> thm -> thm
```

Implements the Modus Ponens inference rule.

When applied to theorems `A1 |- t1 ==> t2` and `A2 |- t1`, the inference rule `MP` returns the theorem `A1 u A2 |- t2`.

```
 A1 |- t1 ==> t2   A2 |- t1
---------------------------  MP
       A1 u A2 |- t2
```

In common with the underlying `dest_imp` syntax function, `MP` treats theorems with conclusions of the form `~p` as implications `p ==> F`. This means that `MP` also has the following behaviour:

```
 A1 |- ~t1     A2 |- t1
------------------------  MP
      A1 u A2 |- F
```

Failure

Fails unless the first theorem is an implication (in the sense of `dest_imp`) whose antecedent is the same as the conclusion of the second theorem (up to alpha-conversion)

See also

boolSyntax.dest_imp, Thm.EQ_MP, Drule.LIST_MP, Drule.MATCH_MP, Tactic.MATCH_-MP_TAC, Tactic.MP_TAC

MP_CONV (Conv)

```
Conv.MP_CONV : conv -> conv
```

Eliminate the antecedent of a theorem using a conversion/proof rule.

If `c` is a conversion that when applied to `P` returns the theorem `|- P = T` or `|- P`, and `th` is a theorem of the general form `|- P ==> Q`, then `MP_CONV c th` will return the theorem `|- Q`, i.e. the antecedent of `th` is eliminated by the conversion `c`. This is done by calling `MP` on `|- P ==> Q` and `|- P`.

Failure

`MP_CONV c th` will fail if `th` is not of the form `|- P ==> Q` or if `c` fails when applied to `P`.

Example

```
> load "realLib"; open realTheory realLib;
<<HOL message: inventing new type variable names: 'a>>
val it = (): unit
> MP_CONV REAL_ARITH (Q.SPEC `1` REAL_DOWN);
val it = ⊢ ∃y. 0 < y ∧ y < 1: thm
```

Comments

This conversion is ported from HOL-Light (`drule.ml`). `MP_CONV` is useful when a universal theorem, after instantiating some of its quantifiers, the antecedent becomes a tautology that can be eliminated by a conversion.

See also

Thm.MP

MP_TAC (Tactic)

```
Tactic.MP_TAC : thm_tactic
```

Reduces a goal to implication from a known theorem.

When applied to the theorem `A' |- s` and the goal `A ?- t`, the tactic `MP_TAC` reduces the goal to `A ?- s ==> t`. Unless `A'` is a subset of `A`, this is an invalid tactic.

```
    A ?- t
==============  MP_TAC (A' |- s)
 A ?- s ==> t
```

Failure

Never fails.

See also

`Tactic.MATCH_MP_TAC`, `Thm.MP`, `Tactic.UNDISCH_TAC`

mp_then (Tactic)

```
Tactic.mp_then : match_position -> thm_tactic -> thm -> thm -> tactic
```

Matches two theorems against each other and then continues

The `mp_then` tactic combines two theorems (one or both of which will typically come from the current goal's assumptions) using modus ponens in a controlled way, and then passes the result of this to a continuation theorem-tactic.

Thus `mp_then ttac pos ith th` is a tactic in the usual "`_then`" fashion which produces a theorem and then applies the `ttac` continuation to that result. The theorems `ith` and `th` are the two theorems: `ith` contains the implication, and the other has a conclusion matching one of the antecedents of the implication.

An implication's antecedents are calculated by first normalising the implication so that it is of the form

```
!v1 .. vn. ant1 /\ ant2 .. /\ antn ==> concl
```

The `pos` parameter indicates how to find the antecedent. There are four possible forms for `pos` (constructors for the `match_position` type). It can be `Any`, which tells `mp_then` to search for anything that works. It can be `Pat q`, with `q` a quotation, which means to find anything matching `q` that works. It can be `Pos f`, where `f` is a function of type `term list -> term`, and is typically a value such as `hd`, `el n` for some number `n` or `last`. This function is passed the list of all `ith`'s antecedents to pick the right one. Finally, the `pos` parameter might be `Concl`, meaning that the conclusion of `ith` is treated as a negated assumption. This allows implications to be used in a contrapositive way.

In practice, there are some common patterns for obtaining `ith` and `th`. Apart from the fully applied version above, you might also see:

```
<sel>_assum (mp_then pos ttac ith)

<sel>_assum (<sel>_assum o mp_then pos ttac)

disch_then(<sel>_assum o mp_then pos ttac)
```

where `<sel>` is one of `first`, `last`, `qpat` and `goal`, possibly with an appended `_x`; the usual ways of obtaining theorems from the current goal. Where there are two selectors used, the outermost is used for the selection of the implicational theorem and the innermost selects `th`. In the first example, the `ith` value is provided in the call, and is presumably an existing theorem rather than an assumption from the goal.

The `goal_assum` selector is worth special mention since it's especially useful with `mp_then`: it turns an existentially quantified goal `?x. P x` into the assumption `!x. P x ==> F` thereby providing a theorem with antecedents to match on. In conjunction with `mp_tac` (which will put the revised implication back into the goal as an existential once more) it has the effect of instantiating the existential quantifier in order to match a provided subterm (similar to `part_match_exists_tac` or `asm_exists_tac`).

Finally, note that the `Pat` and `Any` position selectors will backtrack across the set of theorem antecedents to find a match that makes the whole application succeed.

Failure

If the provided implicational theorem doesn't have a match at a compatible position for the second provided theorem, or if no such match then allows the continuation `ttac` to

succeed.

See also

`Tactical.goal_assum`, `Tactic.resolve_then`

MUL_CONV (reduceLib)

`reduceLib.MUL_CONV : conv`

Calculates by inference the product of two numerals.

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `MUL_CONV "m * n"` returns the theorem:

```
|- m * n = s
```

where `s` is the numeral that denotes the product of the natural numbers denoted by `m` and `n`.

Failure

`MUL_CONV tm` fails unless `tm` is of the form `"m * n"`, where `m` and `n` are numerals.

Example

```
#MUL_CONV "0 * 12";;
|- 0 * 12 = 0

#MUL_CONV "1 * 1";;
|- 1 * 1 = 1

#MUL_CONV "6 * 11";;
|- 6 * 11 = 66
```

n2w_INTRO_TAC (wordsLib)

```
wordsLib.n2w_INTRO_TAC : int -> tactic
```

The tactic `n2w_INTRO_TAC i` attempts to recast finite problems (over `num`) of the form `m = n`, `m < n` and `m <= n` into problems over bit-vectors of size `i`.

Example

Given the goal:

```
?- w2n (a: word4) + w2n (b: word4) < 32
```

applying

```
e (wordsLib.n2w_INTRO_TAC 6)
```

gives the new goal

```
[ w2n a < 16, w2n b < 16 ] ?- w2w a + w2w b <+ 32w
```

This goal can be solved using `blastLib.BBLAST_TAC`. Any word length strictly greater than five would have sufficed here; it is generally best to pick as small a word size as is necessary.

See also

`blastLib.BBLAST_CONV`

namedCases (BasicProvers)

```
BasicProvers.namedCases : string list -> tactic
```

Re-exported from `bossLib.namedCases`. See that entry for full documentation.

namedCases (bossLib)

```
bossLib.namedCases : string list -> tactic
```

Also exported as `BasicProvers.namedCases`.

Case split on type of leading universally quantified variable in the goal, using given names for introduced constructor arguments.

An application of `namedCases [s1, ..., sn]` to a goal of the form `!x:ty. P` will perform a case split on the type `ty`, using the given names for the arguments of the introduced constructor terms. The type `ty` should be that of a datatype that has a so-called "nchotomy" theorem installed in the system database of declared datatypes, accessible via `TypeBase.nchotomy_of`.

For a datatype with `n` constructors, `n` strings are expected to be supplied. If no strings are supplied, the system will use a default naming scheme. If the `i`th constructor has no arguments, then `si` should be the empty string. If the `i`th constructor has `k` arguments, then `si` should consist of `k` space-separated names. In case a name does not need to be specified, an underscore _ or dash - can be supplied, in which case a default name will be conjured up.

In case `ty` is a product type `ty1 # ... # tyi`, `namedCases [s]` will iteratively case split on all product types in `ty`, thus replacing `x:ty` by a tuple with `i` variables, the names of which are taken from `s`.

Failure

Fails if there is not an nchotomy theorem installed for the topmost type constructor of `ty`. If `slist` is not the empty list, `namedCases slist` will fail if the length of `slist` is not equal to the number of constructors in the nchotomy theorem. Fails if the given names for arguments of an introduced constructor are not equinumerous with the arguments.

Example

Consider the goal

```
A ?- !x:num#num#bool. P x
```

Invoking `namedCases ["a b c"]` yields the goal

```
A ?- P (a,b,c)
```

while `namedCases ["a _ _"]` yields the goal

```
A ?- P (a,_gv0,_gv1)
```

Example

Consider a datatype of arithmetic expressions declared as

```
Datatype:
  arith
    = Var 'a
    | Const num
    | Add arith arith
    | Sub arith arith
    | Mult arith arith
End
```

and the goal

```
A ?- !x:'a arith. P x
```

Invoking `namedCases ["v","c","a1 a2", "s1 s2", "m1 m2"]` yields the following 5 goals

```
P (Mult m1 m2)

P (Sub s1 s2)

P (Add a1 a2)

P (Const c)

P (Var v)
```

See also

`bossLib.namedCases_on`, `bossLib.Cases_on`, `bossLib.Cases`, `TypeBase.nchotomy_of`

namedCases_on (BasicProvers)

```
BasicProvers.namedCases_on : term quotation -> string list -> tactic
```

Re-exported from `bossLib.namedCases_on`. See that entry for full documentation.

namedCases_on (bossLib)

```
bossLib.namedCases_on : term quotation -> string list -> tactic
```

Also exported as `BasicProvers.namedCases_on`.

Case split on type of given term, using given names for introduced constructor arguments.

An application of `namedCases_on q [s1, ..., sn]` to a goal `A ?- P` first parses `q` in the context of the goal to yield a term `tm:ty`, then uses `ty` to look up a a so-called "nchotomy" theorem installed in the system database of declared datatypes, then performs a case split on how `tm` can be constructed. The strings `s1, ..., sn` designate the names to be used as arguments of the constructor in each case. This yields the goals

```
(A, tm = <constr>1 <names>1 ?- P)
 ,...,
(A, tm = <constr>n <names>n ?- P)
```

For a datatype with `n` constructors, `n` strings are expected to be supplied. If no strings are supplied, the system will use a default naming scheme. If the `i`th constructor has no arguments, then `si` should be the empty string. If the `i`th constructor has `k` arguments, then `si` should consist of `k` space-separated names. In case a name does not need to be specified, an underscore `_` or dash `-` can be supplied, in which case a default name will be conjured up.

In case `ty` is a product type `ty1 # ... # tyi`, `namedCases_on q [s]` will iteratively case split on all product types in `ty`, thus replacing `x:ty` by a tuple with `i` variables, the names of which are taken from `s`.

Failure

Fails if there is not an nchotomy theorem installed for the topmost type constructor of `ty`. If `slist` is not the empty list, `namedCases_on q slist` will fail if the length of `slist` is not equal to the number of constructors in the nchotomy theorem. Fails if the given names for arguments of an introduced constructor are not equinumerous with the arguments.

Comments

This is a version of `namedCases` where the (free) term being split on is specified.

See also

`bossLib.namedCases`, `bossLib.Cases_on`, `bossLib.Cases`, `TypeBase.nchotomy_of`

NEG_DISCH (Drule)

```
Drule.NEG_DISCH : term -> thm -> thm
```

Discharges an assumption, transforming `|- s ==> F` into `|- ~s`.

When applied to a term `s` and a theorem `A |- t`, the inference rule `NEG_DISCH` returns the theorem `A - {s} |- s ==> t`, or if `t` is just `F`, returns the theorem `A - {s} |- ~s`.

```
        A |- F
--------------------  NEG_DISCH    [special case]
   A - {s} |- ~s

        A |- t
--------------------  NEG_DISCH    [general case]
 A - {s} |- s ==> t
```

Failure

Fails unless the supplied term has type `bool`.

See also

Thm.DISCH, Thm.NOT_ELIM, Thm.NOT_INTRO

NEGATE_CONV (Arith)

```
Arith.NEGATE_CONV : (conv -> conv)
```

Function for negating the operation of a conversion that proves a formula to be either true or false.

This function negates the operation of a conversion that proves a formula to be either true or false. For example, if conv proves "t" to be equal to "T" then NEGATE_CONV conv will prove "~t" to be "F".

Failure

Fails if the application of the conversion to the negation of the formula does not yield either "T" or "F".

Example

```
#ARITH_CONV "!n. 0 <= n";;
|- (!n. 0 <= n) = T

#NEGATE_CONV ARITH_CONV "~(!n. 0 <= n)";;
|- ~(!n. 0 <= n) = F

#NEGATE_CONV ARITH_CONV "?n. ~(0 <= n)";;
|- (?n. ~0 <= n) = F
```

negation (boolSyntax)

```
boolSyntax.negation : term
```

Constant denoting logical negation.

The ML variable `boolSyntax.negation` is bound to the term `bool$~`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`

NEQ_CONV (reduceLib)

```
reduceLib.NEQ_CONV : conv
```

Proves equality or inequality of two numerals.

If `m` and `n` are both numerals (e.g. `0`, `1`, `2`, `3`,...), then `NEQ_CONV "m = n"` returns the theorem:

```
|- (m = n) = T
```

if `m` and `n` are identical, or

```
|- (m = n) = F
```

if `m` and `n` are distinct.

Failure

`NEQ_CONV tm` fails unless `tm` is of the form `"m = n"`, where `m` and `n` are numerals.

Example

```
#NEQ_CONV "12 = 12";;
|- (12 = 12) = T

#NEQ_CONV "14 = 25";;
|- (14 = 25) = F
```

new_axiom (Theory)

```
Theory.new_axiom : string * term -> thm
```

Install a new axiom in the current theory.

If `M` is a term of type `bool`, a call `new_axiom(name,M)` creates a theorem

```
|- tm
```

and stores it away in the current theory segment under `name`.

Failure

Fails if the given term does not have type `bool`.

Example

```
> new_axiom("untrue", Term `!x. x = 1`);
val it = ⊢ ∀x. x = 1: thm
```

Comments

For most purposes, it is unnecessary to declare new axioms: all of classical mathematics can be derived by definitional extension alone. Proceeding by definition is not only more elegant, but also guarantees the consistency of the deductions made. However, there are certain entities which cannot be modelled in simple type theory without further axioms, such as higher transfinite ordinals.

See also

`Thm.mk_thm`, `Definition.new_definition`, `Definition.new_specification`

new_binder (boolSyntax)

```
boolSyntax.new_binder : string * hol_type -> unit
```

Sets up a new binder in the current theory.

A call `new_binder(bnd,ty)` declares a new binder `bnd` in the current theory. The type must be of the form `('a -> 'b) -> 'c`, because being a binder, `bnd` will apply to an abstraction; for example

```
!x:bool. (x=T) \/ (x=F)
```

is actually a prettyprinting of

```
$! (\x. (x=T) \/ (x=F))
```

Failure

Fails if the type does not correspond to the above pattern.

Example

```
- new_theory "anorak";
> val it = () : unit

- new_binder ("!!", (bool-->bool)-->bool);
> val it = () : unit

- Term `!!x. T`;
> val it = `!! x. T` : term
```

See also

`Theory.constants`, `Theory.new_constant`, `boolSyntax.new_infix`, `Definition.new_definition`, `boolSyntax.new_infixl_definition`, `boolSyntax.new_infixr_definition`, `boolSyntax.new_binder_definition`

new_binder_definition (boolSyntax)

```
boolSyntax.new_binder_definition : string * term -> thm
```

Defines a new constant, giving it the syntactic status of a binder.

The function `new_binder_definition` provides a facility for making definitional extensions to the current theory by introducing a constant definition. It takes a pair of arguments, consisting of the name under which the resulting theorem will be saved in the current theory segment and a term giving the desired definition. The value returned by `new_binder_definition` is a theorem which states the definition requested by the user.

Let `v1`, ..., `vn` be syntactically distinct tuples constructed from the variables `x1`,...,`xm`. A binder is defined by evaluating

```
new_binder_definition (name, `b v1 ... vn = t`)
```

where `b` does not occur in `t`, all the free variables that occur in `t` are a subset of `x1`,...,`xn`, and the type of `b` has the form `(ty1->ty2)->ty3`. This declares `b` to be a new constant with the syntactic status of a binder in the current theory, and with the definitional theorem

```
|- !x1...xn. b v1 ... vn = t
```

as its specification. This constant specification for `b` is saved in the current theory under the name `name` and is returned as a theorem.

The equation supplied to `new_binder_definition` may optionally have any of its free variables universally quantified at the outermost level. The constant `b` has binder status only after the definition has been made.

Failure

`new_binder_definition` fails if `t` contains free variables that are not in any one of the variable structures `v1`, ..., `vn` or if any variable occurs more than once in `v1`, ..., `v2`. Failure also occurs if the type of `b` is not of the form appropriate for a binder, namely a type of the form `(ty1->ty2)->ty3`. Finally, failure occurs if there is a type variable in `v1`, ..., `vn` or `t` that does not occur in the type of `b`.

Example

The unique-existence quantifier `?!` is defined as follows.

```
> new_binder_definition
   (`EXISTS_UNIQUE_DEF`,
    Term`$?! = \P:(*->bool). ($? P) /\ (!x y. ((P x) /\ (P y)) ==> (x=y))`);
Exception- Type error in function application.
   Function: new_binder_definition : string * term -> thm
   Argument:
      (
         [QUOTE " (*#loc 2 4*)EXISTS_UNIQUE_DEF"],
         Term
         [
            QUOTE
            " (*#loc 3 8*)$?! = \\P:(*->bool). ($? P) /\\ (!x y. ((P x) /\\ (P y)) ==> (
            ]
         ) : 'a frag list * term
   Reason:
      Can't unify string (*In Basis*) with 'a frag list (*In Basis*)
         (Different type constructors)
Fail "Static Errors" raised
```

Comments

It is a common practice among HOL users to write a `$` before the constant being defined as a binder to indicate that it will have a special syntactic status after the definition is made:

```
new_binder_definition(name, Term `$b = ... `);
```

This use of `$` is not necessary; but after the definition has been made `$` must, of course, be used if the syntactic status of `b` needs to be suppressed.

See also

`Definition.new_definition`, `boolSyntax.new_infixl_definition`, `boolSyntax.new_infixr_definition`, `Prim_rec.new_recursive_definition`, `TotalDefn.Define`

new_constant (Theory)

```
Theory.new_constant : string * hol_type -> unit
```

Declares a new constant in the current theory.

A call `new_constant(n,ty)` installs a new constant named `n` in the current theory. Note that `new_constant` does not specify a value for the constant, just a name and type. The constant may have a polymorphic type, which can be used in arbitrary instantiations.

Failure

Never fails, but issues a warning if the name is not a valid constant name. It will overwrite an existing constant with the same name in the current theory.

See also

`Theory.constants`, `boolSyntax.new_infix`, `boolSyntax.new_binder`, `Definition.new_definition`, `Definition.new_type_definition`, `Definition.new_specification`, `Theory.new_axiom`, `boolSyntax.new_infixl_definition`, `boolSyntax.new_infixr_definition`, `boolSyntax.new_binder_definition`

new_definition (Definition)

```
Definition.new_definition : string * term -> thm
```

Declare a new constant and install a definitional axiom in the current theory.

The function `new_definition` provides a facility for definitional extensions to the current theory. It takes a pair argument consisting of the name under which the resulting definition will be saved in the current theory segment, and a term giving the desired definition. The value returned by `new_definition` is a theorem which states the definition requested by the user.

Let `v_1`,...,`v_n` be tuples of distinct variables, containing the variables `x_1`,...,`x_m`. Evaluating `new_definition (name, c v_1 ... v_n = t)`, where `c` is not already a constant, declares the sequent `({},\v_1 ... v_n. t)` to be a definition in the current theory, and declares `c` to be a new constant in the current theory with this definition as its specification. This constant specification is returned as a theorem with the form

```
|- !x_1 ... x_m. c v_1 ... v_n = t
```

and is saved in the current theory under `name`. Optionally, the definitional term argument may have any of its variables universally quantified.

Failure

`new_definition` fails if `t` contains free variables that are not in `x_1`, ..., `x_m` (this is equivalent to requiring `\v_1 ... v_n. t` to be a closed term). Failure also occurs if any variable occurs more than once in `v_1, ..., v_n`. Finally, failure occurs if there is a type variable in `v_1`, ..., `v_n` or `t` that does not occur in the type of `c`.

Example

A NAND relation can be defined as follows.

```
- new_definition (
    "NAND2",
    Term`NAND2 (in_1,in_2) out = !t:num. out t = ~(in_1 t /\ in_2 t)`);

> val it =
    |- !in_1 in_2 out.
          NAND2 (in_1,in_2) out = !t. out t = ~(in_1 t /\ in_2 t)
    : thm
```

See also

`Definition.new_specification`, `boolSyntax.new_binder_definition`, `boolSyntax.new_infixl_definition`, `boolSyntax.new_infixr_definition`, `Prim_rec.new_recursive_definition`, `TotalDefn.Define`

new_infix (boolSyntax)

```
boolSyntax.new_infix : string * hol_type * int -> unit
```

Declares a new infix constant in the current theory.

A call `new_infix ("i", ty, n)` makes `i` a right associative infix constant in the current theory. It has binding strength of `n`, the larger this number, the more tightly the infix will attempt to "grab" arguments to its left and right. Note that the call to `new_infix` does not specify the value of the constant. The constant may have a polymorphic type, which may be arbitrarily instantiated. Like any other infix or binder, its special parse status may be suppressed by preceding it with a dollar sign.

Comments

Infixes defined with `new_infix` associate to the right, i.e., `A <op> B <op> C` is equivalent to `A op (B <op> C)`. Some standard infixes, with their precedences and associativities in the system are:

```
      $,  ---> 50     RIGHT
      $=  ---> 100    NONASSOC
    $==>  ---> 200    RIGHT
     $\/  ---> 300    RIGHT
     $/\  ---> 400    RIGHT
  $>, $<  ---> 450    RIGHT
$>=, $<=  ---> 450    RIGHT
  $+, $-  ---> 500    LEFT
$*, $DIV  ---> 600    LEFT
    $MOD  ---> 650    LEFT
    $EXP  ---> 700    RIGHT
    $o    ---> 800    RIGHT
```

Note that the arithmetic operators `+`, `-`, `*`, `DIV` and `MOD` are left associative in hol98 releases from Taupo onwards. Non-associative infixes (= above, for example) will cause parse errors if an attempt is made to group them (e.g., `x = y = z`).

Failure

Fails if the name is not a valid constant name.

Example

The following shows the use of the infix and the prefix form of an infix constant. It also shows binding resolution between infixes of different precedence.

```
- new_infix("orelse", Type`:bool->bool->bool`, 50);
val it = () : unit

- Term`T \/ T orelse F`;
val it = `T \/ T orelse F` : term

- “$orelse T F”;
val it = `T orelse F` : term

- dest_comb “T \/ T orelse F”;
> val it = (`$orelse (T \/ T)`,  `F`) : term * term
```

See also

`Parse.add_infix`, `Theory.constants`, `Theory.new_constant`, `boolSyntax.new_binder`, `Definition.new_definition`, `boolSyntax.new_binder_definition`

new_infixl_definition (boolSyntax)

```
boolSyntax.new_infixl_definition : string * term * int -> thm
```

Declares a new left associative infix constant and installs a definition in the current theory.

The function `new_infix_definition` provides a facility for definitional extensions to the current theory. It takes a triple consisting of the name under which the resulting definition will be saved in the current theory segment, a term giving the desired definition and an integer giving the precedence of the infix. The value returned by `new_infix_definition` is a theorem which states the definition requested by the user.

Let `v_1` and `v_2` be tuples of distinct variables, containing the variables `x_1,...,x_m`. Evaluating `new_infix_definition (name, ix v_1 v_2 = t)` declares the sequent `({},\v_1 v_2. t)` to be a definition in the current theory, and declares `ix` to be a new constant in the current theory with this definition as its specification. This constant specification is returned as a theorem with the form

```
|- !x_1 ... x_m. v_1 ix v_2 = t
```

and is saved in the current theory under (the name) `name`. Optionally, the definitional term argument may have any of its variables universally quantified. The constant `ix` has infix status only after the infix declaration has been processed. It is therefore necessary to use the constant in normal prefix position when making the definition.

Failure

`new_infixl_definition` fails if `t` contains free variables that are not in either of the variable structures `v_1` and `v_2` (this is equivalent to requiring `\v_1 v_2. t` to be a closed term); or if any variable occurs more than once in `v_1, v_2`. It also fails if the precedence level chosen for the infix is already home to parsing rules of a different form of fixity (infixes associating in a different way, or suffixes, prefixes etc). Finally, failure occurs if there is a type variable in `v_1`, ..., `v_n` or `t` that does not occur in the type of `ix`.

Example

The nand function can be defined as follows.

```
- new_infix_definition
 ("nand", “$nand in_1 in_2 = ~(in_1 /\ in_2)”, 500);;
> val it = |- !in_1 in_2. in_1 nand in_2 = ~(in_1 /\ in_2) : thm
```

Comments

It is a common practice among HOL users to write a `$` before the constant being defined as an infix to indicate that after the definition is made, it will have a special syntactic status; ie. to write:

```
new_infixl_definition("ix_DEF", Term `$ix m n = ... `)
```

This use of `$` is not necessary; but after the definition has been made `$` must, of course, be used if the syntactic status needs to be suppressed.

In releases of hol98 past Taupo 1, `new_infixl_definition` and its sister `new_infixr_definition` replace the old `new_infix_definition`, which has been superseded. Its behaviour was to define a right associative infix, so can be freely replaced by `new_infixr_definition`.

See also

`boolSyntax.new_binder_definition`, `Definition.new_definition`, `Definition.new_specification`, `boolSyntax.new_infixr_definition`, `Prim_rec.new_recursive_definition`, `TotalDefn.Define`

new_infixr_definition (boolSyntax)

```
boolSyntax.new_infixr_definition : string * term * int -> thm
```

Declares a new right associative infix constant and installs a definition in the current theory.

The function `new_infixr_definition` has exactly the same effect as `new_infixl_definition` except that the infix constant defined will associate to the right.

See also

`Definition.new_definition`, `Definition.new_specification`, `boolSyntax.new_infix`, `boolSyntax.new_infixl_definition`

new_recursive_definition (Prim_rec)

```
Prim_rec.new_recursive_definition : {name:string, def:term, rec_axiom:thm} -> thm
```

Defines a primitive recursive function over a concrete recursive type.

`new_recursive_definition` provides a facility for defining primitive recursive functions on arbitrary concrete recursive types. `name` is a name under which the resulting definition will be saved in the current theory segment. `def` is a term giving the desired primitive recursive function definition. `rec_axiom` is the primitive recursion theorem for the concrete type in question; this must be a theorem obtained from `define_type`. The value returned by `new_recursive_definition` is a theorem which states the primitive recursive definition requested by the user. This theorem is derived by formal proof from an instance of the general primitive recursion theorem given as the second argument.

A theorem `th` of the form returned by `define_type` is a primitive recursion theorem for an automatically-defined concrete type `ty`. Let `C1`, ..., `Cn` be the constructors of this type, and let '`(Ci vs)`' represent a (curried) application of the `i`th constructor to a sequence of variables. Then a curried primitive recursive function `fn` over `ty` can be specified by a conjunction of (optionally universally-quantified) clauses of the form:

```
fn v1 ... (C1 vs1) ... vm  =  body1   /\
fn v1 ... (C2 vs2) ... vm  =  body2   /\
                              .
                              .
fn v1 ... (Cn vsn) ... vm  =  bodyn
```

where the variables `v1`, ..., `vm`, `vs` are distinct in each clause, and where in the `i`th clause `fn` appears (free) in `bodyi` only as part of an application of the form:

```
fn t1 ... v ... tm
```

in which the variable `v` of type `ty` also occurs among the variables `vsi`.

If `tm` is a conjunction of clauses, as described above, then evaluating:

```
new_recursive_definition{name=name, rec_axiom=th, def=tm}
```

automatically proves the existence of a function `fn` that satisfies the defining equations supplied as the fourth argument, and then declares a new constant in the current theory with this definition as its specification. This constant specification is returned as a theorem and is saved in the current theory segment under the name `name`.

`new_recursive_definition` also allows the supplied definition to omit clauses for any number of constructors. If a defining equation for the `i`th constructor is omitted, then the value of `fn` at that constructor:

```
fn v1 ... (Ci vsi) ... vn
```

is left unspecified (`fn`, however, is still a total function).

Failure

A call to `new_recursive_definition` fails if the supplied theorem is not a primitive recursion theorem of the form returned by `define_type`; if the term argument supplied is not a well-formed primitive recursive definition; or if any other condition for making a constant specification is violated (see the failure conditions for `new_specification`).

Example

Given the following primitive recursion theorem for labelled binary trees:

```
|- !f0 f1.
     ?! fn.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

`new_recursive_definition` can be used to define primitive recursive functions over binary trees. Suppose the value of `th` is this theorem. Then a recursive function `Leaves`, which computes the number of leaves in a binary tree, can be defined recursively as shown below:

```
- val Leaves = new_recursive_definition
        {name = "Leaves",
         rec_axiom = th,
         def= “(Leaves (LEAF (x:'a)) = 1) /\
                 (Leaves (NODE t1 t2) = (Leaves t1) + (Leaves t2))”};
 > val Leaves =
     |- (!x. Leaves (LEAF x) = 1) /\
        !t1 t2. Leaves (NODE t1 t2) = Leaves t1 + Leaves t2 : thm
```

The result is a theorem which states that the constant `Leaves` satisfies the primitive-recursive defining equations supplied by the user.

The function defined using `new_recursive_definition` need not, in fact, be recursive. Here is the definition of a predicate `IsLeaf`, which is true of binary trees which are leaves, but is false of the internal nodes in a binary tree:

```
- val IsLeaf = new_recursive_definition
        {name = "IsLeaf",
         rec_axiom = th,
         def = “(IsLeaf (NODE t1 t2) = F) /\
                  (IsLeaf (LEAF (x:'a)) = T)”};
> val IsLeaf = |- (!t1 t2. IsLeaf (NODE t1 t2) = F) /\
                !x. IsLeaf (LEAF x) = T : thm
Exception- unknown symbol .
unknown symbol .
Fail "Static Errors" raised
```

Note that the equations defining a (recursive or non-recursive) function on binary trees by cases can be given in either order. Here, the NODE case is given first, and the LEAF case second. The reverse order was used in the above definition of Leaves.

new_recursive_definition also allows the user to partially specify the value of a function defined on a concrete type, by allowing defining equations for some of the constructors to be omitted. Here, for example, is the definition of a function Label which extracts the label from a leaf node. The value of Label applied to an internal node is left unspecified:

```
- val Label = new_recursive_definition
                {name = "Label",
                 rec_axiom = th,
                 def = “Label (LEAF (x:'a)) = x”};
> val Label = |- !x. Label (LEAF x) = x : thm
```

Curried functions can also be defined, and the recursion can be on any argument. The next definition defines an infix function << which expresses the idea that one tree is a proper subtree of another.

```
- val _ = set_fixity ("<<", Infixl 231);

- val Subtree = new_recursive_definition
       {name = "Subtree",
        rec_axiom = th,
        def = “($<< (t:'a bintree) (LEAF (x:'a)) = F) /\
                 ($<< t (NODE t1 t2) = (t = t1) \/
                                       (t = t2) \/
                                       ($<< t t1) \/
                                       ($<< t t2))”};
> val Subtree =
    |- (!t x. t << LEAF x = F) /\
       !t t1 t2.
         t << NODE t1 t2 = (t = t1) \/ (t = t2) \/
                           (t << t1) \/ (t << t2) : thm
```

Note that the fixity of the identifier << is set independently of the definition.

See also

bossLib.Hol_datatype, Prim_rec.prove_rec_fn_exists, TotalDefn.Define, Parse.set_fixity

new_specification (Definition)

```
Definition.new_specification : string * string list * thm -> thm
```

Introduce a constant or constants satisfying a given property.

The ML function `new_specification` implements the primitive rule of constant specification for the HOL logic. Evaluating:

```
new_specification (name, ["c1",...,"cn"], |- ?x1...xn. t)
```

simultaneously introduces new constants named `c1`,...,`cn` satisfying the property:

```
|- t[c1,...,cn/x1,...,xn]
```

This theorem is stored, with name `name`, as a definition in the current theory segment. It is also returned by the call to `new_specification`.

Failure

`new_specification` fails if the theorem argument has assumptions or free variables. It also fails if the supplied constant names `c1`, ..., `cn` are not distinct. It also fails if the length of the existential prefix of the theorem is not at least `n`. Finally, failure occurs if some `ci` does not contain all the type variables that occur in the term `?x1...xn. t`.

`new_specification` can be used to introduce constants that satisfy a given property without having to make explicit equational constant definitions for them. For example, the built-in constants `MOD` and `DIV` are defined in the system by first proving the theorem:

```
th |- ?MOD DIV.
        !n. 0 < n ==> !k. (k = (DIV k n * n) + MOD k n) /\ MOD k n < n
```

and then making the constant specification:

```
new_specification ("DIVISION", ["MOD","DIV"], th)
```

This introduces the constants `MOD` and `DIV` with the defining property shown above.

Comments

The introduced constants have a prefix parsing status. To alter this, use `set_fixity`. Typical fixity values are `Binder`, `Infixl n`, `Infixr n`, `Prefix n`, `Suffix n`, or `Closefix`.

See also

`Definition.gen_new_specification`, `Definition.new_definition`, `boolSyntax.new_binder_definition`, `boolSyntax.new_infixl_definition`, `boolSyntax.new_infixr_definition`, `TotalDefn.Define`, `Parse.set_fixity`

new_theory (Theory)

```
Theory.new_theory : string -> unit
```

Creates a new theory segment.

A theory consists of a hierarchy of named parts called 'theory segments'. All theory segments have a 'theory' of the same name associated with them consisting of the theory segment itself together with the contents of all its ancestors. Each axiom, definition, specification and theorem belongs to a particular theory segment.

Calling `new_theory thy` creates a new, and empty, theory segment having name `thy`. The theory segment which was current before the call becomes a parent of the new theory segment. The new theory therefore consists of the current theory extended with the new theory segment. The new theory segment replaces its parent as the current theory segment. The parent segment is exported to disk.

In the interests of interactive usability, the behaviour of `new_theory` has some special cases. First, if `new_theory thy` is called in a situation where the current theory segment is already called `thy`, then this is interpreted as the user wanting to restart the current segment. In that case, the current segment is wiped clean (types and constants declared in it are removed from the signature, and all definitions, theorems and axioms are deleted) but is otherwise unchanged (it keeps the same parents, for example).

Second, if the current theory segment is empty and named `"scratch"`, then `new_theory thy` creates a new theory segment the parents of which are the parents of `"scratch"`. (This situation is almost never visible to users.)

Failure

A call `new_theory thy` fails if the name `thy` is unsuitable for use as a filename. In particular, it should be an alphanumeric identifier.

Failure also occurs if `thy` is the name of a currently loaded theory segment or if is in a set of reserved words that includes all SML and HOL keywords (*e.g.,* `if`, `while`, `Definition`). In general, all theory names, whether loaded or not, should be distinct. Moreover, the names should be distinct even when case distinctions are ignored.

Example

In the following, we follow a standard progression: start HOL up and declare a new theory segment.

```
- current_theory();
> val it = "scratch" : string

- parents "-";
> val it = ["list", "option"] : string list

- new_theory "foo";
<<HOL message: Created theory "foo">>
> val it = () : unit

- parents "-";
> val it = ["list", "option"] : string list
```

Next we make a definition, prove and store a theorem, and then change our mind about the name of the defined constant. Restarting the current theory keeps the static theory context fixed but clears the current segment so that we have a clean slate to work from.

```
- val def = new_definition("foo", Term `foo x = x + x`);
> val def = |- !x. foo x = x + x : thm

val thm = Q.store_thm("foo_thm", `foo x = 2 * x`,
                                  RW_TAC arith_ss [def]);
> val thm = |- foo x = 2 * x : thm

- new_theory "foo";
```

```
<<HOL message: Restarting theory "foo">>
> val it = () : unit

val def = new_definition("twice", Term `twice x = x + x`);
> val def = |- !x. twice x = x + x : thm

- curr_defs();
> val it = [("twice", |- !x. twice x = x + x)]
           : (string * thm) list
```

Comments

The theory file in which the data of the new theory segment is ultimately stored will have name `thyTheory` in the directory in which `export_theory` is called.

Modularizing large formalizations. By splitting a formalization effort into theory segments by use of `new_theory`, the work required if definitions, etc., need to be changed is minimized. Only the associated segment and its descendants need be redefined.

See also

`Theory.current_theory`, `Theory.new_axiom`, `Theory.parents`, `boolSyntax.new_binder`, `Theory.new_constant`, `Definition.new_definition`, `boolSyntax.new_infix`, `Definition.new_specification`, `Theory.new_type`, `Theory.save_thm`, `Theory.export_theory`, `Hol_pp.print_theory`

new_type (Theory)

```
Theory.new_type : string * int -> unit
```

Declares a new type or type constructor.

A call `new_type(t,n)` declares a new `n`-ary type constructor called `t` in the current theory segment. If `n` is zero, this is just a new base type.

Failure

Never fails, but issues a warning if the name is not a valid type name. It will overwrite an existing type operator with the same name in the current theory.

Example

A non-definitional version of ZF set theory might declare a new type `set` and start using it as follows:

```
- new_theory "ZF";
<<HOL message: Created theory "ZF">>
> val it = () : unit

- new_type ("set", 0);
> val it = () : unit

- new_constant ("mem", Type`:set->set->bool`);
> val it = () : unit

- new_axiom ("EXT", Term`(!z. mem z x = mem z y) ==> (x = y)`);
> val it = |- (!z. mem z x = mem z y) ==> (x = y) : thm
```

See also

`Theory.types`, `Theory.new_constant`, `Theory.new_axiom`

new_type_definition (Definition)

```
Definition.new_type_definition : string * thm -> thm
```

Defines a new type constant or type operator.

The ML function `new_type_definition` implements the primitive HOL rule of definition for introducing new type constants or type operators into the logic. If `t` is a term of type `ty->bool` containing `n` distinct type variables, then evaluating:

```
new_type_definition (tyop, |- ?x. t x)
```

results in `tyop` being declared as a new `n`-ary type operator in the current theory and returned by the call to `new_type_definition`. This new type operator is characterized by a definitional axiom of the form:

```
|- ?rep:('a,...,'n)op->tyop. TYPE_DEFINITION t rep
```

which is stored as a definition in the current theory segment under the automatically-generated name `op_TY_DEF`. The arguments to the new type operator occur in the order given by an alphabetic ordering of the name of the corresponding type variables. The constant `TYPE_DEFINITION` in this axiomatic characterization of `tyop` is defined by:

```
|- TYPE_DEFINITION (P:'a->bool) (rep:'b->'a) =
      (!x' x''. (rep x' = rep x'') ==> (x' = x'')) /\
      (!x. P x = (?x'. x = rep x'))
```

Thus `|- ?rep. TYPE_DEFINITION P rep` asserts that there is a bijection between the newly defined type `('a,...,'n)tyop` and the set of values of type `ty` that satisfy `P`.

Failure

Executing `new_type_definition(tyop,th)` fails if `th` is not an assumption-free theorem of the form `|- ?x. t x`, if the type of `t` is not of the form `ty->bool`, or if there are free variables in the term `t`.

Example

In this example, a type containing three elements is defined. The predicate defining the type is over the type `bool # bool`.

```
app load ["PairedLambda", "Q"]; open PairedLambda pairTheory;

- val tyax =
   new_type_definition ("three",
     Q.prove(`?p. (\(x,y). ~(x /\ y)) p`,
             Q.EXISTS_TAC `(F,F)` THEN GEN_BETA_TAC THEN REWRITE_TAC []));

> val tyax = |- ?rep. TYPE_DEFINITION (\(x,y). ~(x /\ y)) rep : thm
```

Comments

Usually, once a type has been defined, maps between the representation type and the new type need to be proved. This may be accomplished using `define_new_type_bijections`. In the example, the two functions are named `abs3` and `rep3`.

```
- val three_bij = define_new_type_bijections
                    {name="three_tybij", ABS="abs3", REP="rep3", tyax=tyax};
> val three_bij =
    |- (!a. abs3 (rep3 a) = a) /\
       (!r. (\(x,y). ~(x /\ y)) r = (rep3 (abs3 r) = r))
```

Properties of the maps may be conveniently proved with `prove_abs_fn_one_one`, `prove_abs_fn_onto`, `prove_rep_fn_one_one`, and `prove_rep_fn_onto`. In this case, we need only `prove_abs_fn_one_one`.

```
- val abs_11 = GEN_BETA_RULE (prove_abs_fn_one_one three_bij);

> val abs_11 =
    |- !r r'.
         ~(FST r /\ SND r) ==>
         ~(FST r' /\ SND r') ==>
         ((abs3 r = abs3 r') = (r = r')) : thm
```

Now we address how to define constants designating the three elements of our example type. We will use `new_specification` to create these constants (say `e1`, `e2`, and `e3`) and their characterizing property, which is

```
~(e1 = e2) /\ ~(e2 = e3) /\ ~(e3 = e1)
```

A simple lemma stating that the abstraction function doesn't confuse any of the representations is required:

```
- val abs_distinct =
    REWRITE_RULE (PAIR_EQ::pair_rws)
      (LIST_CONJ (map (C Q.SPECL abs_11)
                      [[`(F,F)`,`(F,T)`],
                       [`(F,T)`,`(T,F)`],
                       [`(T,F)`,`(F,F)`]]));
```

```
> val abs_distinct =
   |- ~(abs3 (F,F) = abs3 (F,T)) /\
      ~(abs3 (F,T) = abs3 (T,F)) /\
      ~(abs3 (T,F) = abs3 (F,F)) : thm
```

Finally, we can introduce the constants and their property.

```
- val THREE = new_specification
    ("THREE", ["e1", "e2", "e3"],
     PROVE [abs_distinct]
      (Term`?x y z:three. ~(x=y) /\ ~(y=z) /\ ~(z=x)`));

> val THREE = |- ~(e1 = e2) /\ ~(e2 = e3) /\ ~(e3 = e1) : thm
```

See also

`Drule.define_new_type_bijections`, `Prim_rec.prove_abs_fn_one_one`, `Prim_rec.prove_abs_fn_onto`, `Drule.prove_rep_fn_one_one`, `Drule.prove_rep_fn_onto`, `Definition.new_specification`

next (Lib)

```
Lib.next : ('a,'b) istream -> ('a,'b) istream
```

Move to the next element in the stream.

An application `next istrm` moves to the next element in the stream.

Failure

If the transition function supplied when building the stream fails on the current state.

Example

```
> val istrm = mk_istream (fn x => x+1) 0 (strcat "gsym" o int_to_string);
val istrm = ?: (int, string) istream
```

```
> next istrm;
val it = ?: (int, string) istream
```

Comments

Perhaps the type of `next` should be `('a,'b) istream -> unit`.

See also

`Lib.mk_istream`, `Lib.state`, `Lib.reset`

NO_CONV (Conv)

```
Conv.NO_CONV : conv
```

Conversion that always fails.

Failure

`NO_CONV` always fails.

See also

`Conv.ALL_CONV`

NO_LT (Tactical)

```
Tactical.NO_LT : list_tactic
```

List-tactic which always fails.

Whatever goal list it is applied to, `NO_LT` always fails with string `` `NO_LT` ``.

Failure

Always fails.

See also

`Tactical.NO_TAC`, `Tactical.ALL_LT`, `Tactical.FAIL_LT`

NO_TAC (Tactical)

```
Tactical.NO_TAC : tactic
```

Tactic which always fails.

Whatever goal it is applied to, `NO_TAC` always fails with string `` `NO_TAC` ``.

Failure

Always fails.

See also

`Tactical.ALL_TAC`, `Thm_cont.ALL_THEN`, `Tactical.FAIL_TAC`, `Thm_cont.NO_THEN`

NO_THEN (Thm_cont)

```
Thm_cont.NO_THEN : thm_tactical
```

Theorem-tactical which always fails.

When applied to a theorem-tactic and a theorem, the theorem-tactical `NO_THEN` always fails with string `` `NO_THEN` ``.

Failure

Always fails when applied to a theorem-tactic and a theorem (note that it never gets as far as being applied to a goal!)

Writing compound tactics or tacticals.

See also

`Tactical.ALL_TAC`, `Thm_cont.ALL_THEN`, `Tactical.FAIL_TAC`, `Tactical.NO_TAC`

NoAsms (bossLib)

```
bossLib.NoAsms : thm
```

A special marker theorem that makes the simplifier ignore a goal's assumptions

The `NoAsms` theorem is a special value that causes a variety of simplification tactics (those ultimately based on `simpLib.ASM_SIMP_TAC`) to ignore a goal's assumptions, even if those tactics might otherwise attempt to use those assumptions when modifying the goal.

Failure

Never fails.

Example

```
> simp[] ([“x = T”], “p ∧ x”);
val it = ([([“x ⇔ T”], “p”)], fn): goal list * validation

> simp[NoAsms] ([“x = T”], “p ∧ x”);
val it = ([([“x ⇔ T”], “p ∧ x”)], fn): goal list * validation
```

See also

`bossLib.IgnAsm`

non_presburger_subterms (Arith)

```
Arith.non_presburger_subterms : (term -> term list)
```

Computes the subterms of a term that are not in the Presburger subset of arithmetic.

This function computes a list of subterms of a term that are not in the Presburger subset of natural number arithmetic. All numeric variables in the term are included in the result. Presburger natural arithmetic is the subset of arithmetic formulae made up from natural number constants, numeric variables, addition, multiplication by a constant, the natural number relations <, <=, =, >=, > and the logical connectives ~, /\, \/, ==>, = (if-and-only-if), ! (‘forall’) and ? (‘there exists’).

Products of two expressions which both contain variables are not included in the subset, so such products will appear in the result list. However, the function SUC which is not normally included in a specification of Presburger arithmetic is allowed in this HOL implementation. This function also considers subtraction and the predecessor function, PRE, to be part of the subset.

Failure

Never fails.

Example

```
#non_presburger_subterms "!m n p. m < (2 * n) /\ (n + n) <= p ==> m < SUC p";;
["m"; "n"; "p"] : term list

#non_presburger_subterms "!m n p q. m < (n * p) /\ (n * p) < q ==> m < q";;
["m"; "n * p"; "q"] : term list

#non_presburger_subterms "(m + n) - m = f n";;
["m"; "n"; "f n"] : term list
```

See also

Arith.INSTANCE_T_CONV, Arith.is_presburger

non_type_definitions (EmitTeX)

```
EmitTeX.non_type_definitions : string -> (string * thm) list
```

A versions of `definitions` that attempts to filter out definitions created by `Hol_datatype`.

An invocation `non_type_definitions thy`, where `thy` is the name of a currently loaded theory segment, will return a list of the definitions stored in that theory. Each definition is paired with its name in the result.

Failure

Never fails. If `thy` is not the name of a currently loaded theory segment, the empty list is returned.

Example

```
> new_theory "example";
<<HOL message: Restarting theory "example">>
<<HOL warning: Theory.callhooks: Hook Parse.clear_consts_from_grammar failed on event Ne
Exception raised at Term.dest_thy_const: not a const
>>
val it = (): unit
> val _ = Hol_datatype `example = First | Second`;
<<HOL message: Defined type: "example">>
> val example_def = Define
   `(example First = Second) /\ (example Second = First)`;
Definition has been stored under "example_def"
val example_def = ⊢ example First = Second ∧ example Second = First: thm

> definitions "example";
val it =
   [("example_TY_DEF", ⊢ ∃rep. TYPE_DEFINITION (λn. n < 2) rep),
    ("example_size_def", ⊢ ∀x. example_size x = 0),
    ("example_def", ⊢ example First = Second ∧ example Second = First),
```

```
    ("example_CASE",
     ⊢ ∀x v0 v1.
         (case x of First => v0 | Second => v1) =
         (λm. if m = 0 then v0 else v1) (example2num x)),
    ("example_BIJ",
     ⊢ (∀a. num2example (example2num a) = a) ∧
       ∀r. (λn. n < 2) r ⇔ example2num (num2example r) = r)]:
   (string * thm) list

> EmitTeX.non_type_definitions "example";
val it =
   [("example_def", ⊢ example First = Second ∧ example Second = First)]:
   (string * thm) list
```

See also

DB.definitions, bossLib.Hol_datatype

non_type_theorems (EmitTeX)

```
EmitTeX.non_type_theorems : string -> (string * thm) list
```

A versions of theorems that attempts to filter out theorems created by Hol_datatype.

An invocation non_type_theorems thy, where thy is the name of a currently loaded theory segment, will return a list of the theorems stored in that theory. Axioms and definitions are excluded. Each theorem is paired with its name in the result.

Failure

Never fails. If thy is not the name of a currently loaded theory segment, the empty list is returned.

Example

```
> new_theory "example";
<<HOL message: Restarting theory "example">>
<<HOL warning: Theory.callhooks: Hook Parse.clear_consts_from_grammar failed on event Ne
Exception raised at Term.dest_thy_const: not a const
>>
val it = (): unit
> val _ = Hol_datatype `example = First | Second`;
<<HOL message: Defined type: "example">>
> val example_def = Define
   `(example First = Second) /\ (example Second = First)`;
Definition has been stored under "example_def"
val example_def = ⊢ example First = Second ∧ example Second = First: thm
> save_thm("example_thm",
  METIS_PROVE [example_def, theorem "example_nchotomy"]
    ``!x. example (example x) = x``);
metis: r[+0+5]+1+0+0+0+6+2+1+0+1+1#
val it = ⊢ ∀x. example (example x) = x: thm

> theorems "example";
val it =
   [("num2example_thm", ⊢ num2example 0 = First ∧ num2example 1 = Second),
    ("num2example_ONTO", ⊢ ∀a. ∃r. a = num2example r ∧ r < 2),
    ("num2example_example2num", ⊢ ∀a. num2example (example2num a) = a),
    ("num2example_11",
     ⊢ ∀r r'. r < 2 ⇒ r' < 2 ⇒ (num2example r = num2example r' ⇔ r = r')),
    ("example_thm", ⊢ ∀x. example (example x) = x),
    ("example_nchotomy", ⊢ ∀a. a = First ∨ a = Second),
    ("example_induction", ⊢ ∀P. P First ∧ P Second ⇒ ∀a. P a),
    ("example_EQ_example", ⊢ ∀a a'. a = a' ⇔ example2num a = example2num a'),
    ("example_distinct", ⊢ First ≠ Second),
    ("example_case_eq",
     ⊢ (case x of First => v0 | Second => v1) = v ⇔
       x = First ∧ v0 = v ∨ x = Second ∧ v1 = v),
    ("example_case_def",
     ⊢ (∀v0 v1. (case First of First => v0 | Second => v1) = v0) ∧
       ∀v0 v1. (case Second of First => v0 | Second => v1) = v1),
    ("example_case_cong",
     ⊢ ∀M M' v0 v1.
         M = M' ∧ (M' = First ⇒ v0 = v0') ∧ (M' = Second ⇒ v1 = v1') ⇒
         (case M of First => v0 | Second => v1) =
```

```
         case M' of First => v0' | Second => v1'),
    ("example_Axiom", ⊢ ∀x0 x1. ∃f. f First = x0 ∧ f Second = x1),
    ("example2num_thm", ⊢ example2num First = 0 ∧ example2num Second = 1),
    ("example2num_ONTO", ⊢ ∀r. r < 2 ⇔ ∃a. r = example2num a),
    ("example2num_num2example",
     ⊢ ∀r. r < 2 ⇔ example2num (num2example r) = r),
    ("example2num_11", ⊢ ∀a a'. example2num a = example2num a' ⇔ a = a'),
    ("datatype_example", ⊢ DATATYPE (example First Second))]:
   (string * thm) list

> EmitTeX.non_type_theorems "example";
val it =
   [("example_case_eq",
     ⊢ (case x of First => v0 | Second => v1) = v ⇔
       x = First ∧ v0 = v ∨ x = Second ∧ v1 = v),
    ("example_thm", ⊢ ∀x. example (example x) = x)]: (string * thm) list
```

See also

DB.theorems, bossLib.Hol_datatype

norm_subst (Term)

```
Term.norm_subst : ((term, term) subst * term set) *
     ((hol_type, hol_type) subst * hol_type list) ->
   (term, term) subst * (hol_type, hol_type) subst
```

Instantiate term substitution by a type substitution.

Given a term substitution and a type substitution, norm_subst applies the type substitution to the redexes of the term substitution.

The substitutions coming from raw_match need to be normalized before they can be applied by inference rules like INST_TY_TERM. An invocation raw_match avoid_tys avoid_tms pat ob A returns a pair of substitutions ((S,Ids),(T,Idt)). The Id components can be ignored. The S component is a substitution for term variables, but it has to be instantiated by the type substitution T in order to be suitable for use by INST_TY_-

TERM. In this case, one uses `norm_subst ((S,Ids),(T,Idt))` as the first argument for `INST_TY_TERM`.

`norm_subst ((S,Ids),(T,Idt))` ignores `Ids` and `Idt`, and returns `T` unchanged. Where a type-substituted term redex becomes equal to the corresponding residue, that term redex-residue pair is omitted from the term substitution returned.

Failure

Never fails.

Example

```
> val ((tmS,Ids),(tyS,Idt)) = raw_match [] empty_varset
                    “\x:'a. x = f (y:'b)”
                    “\a.    a = ~p” ([],[]);
val Ids = HOLset{}: term set
val Idt = []: hol_type list
val tmS = [{redex = “y”, residue = “p”}, {redex = “f”, residue = “$¬”}]:
   (term, term) Term.subst
val tyS =
   [{redex = “:β”, residue = “:bool”}, {redex = “:α”, residue = “:bool”}]:
   (hol_type, hol_type) Term.subst

> val (tmS',_) = norm_subst ((tmS,Ids),(tyS,Idt)) ;
val tmS' = [{redex = “f”, residue = “$¬”}, {redex = “y”, residue = “p”}]:
   (term, term) Term.subst
```

Comments

Higher level matching routines, like `match_term` and `match_terml` already return normalized substitutions.

See also

`Term.raw_match`, `Term.match_term`, `Term.match_terml`, `Drule.INST_TY_TERM`

NORMEQ_ss (boolSimps)

```
boolSimps.NORMEQ_ss : ssfrag
```

A simpset fragment that reorients equalities.

The `NORMEQ_ss` simpset fragment embodies a conversion that flips terms of the form `l = r` when `l` contains no free variables, and `r` contains at least one variable. To flip an equality is to rewrite it so that `l = r` becomes `r = l`.

Failure

As a static value, this cannot fail.

Example

In this example, the simplifier flips the `3 = x` term, making it useful as a rewrite when attacking the consequent of the implication.

```
SIMP_CONV (bool_ss ++ boolSimps.NORMEQ_ss) [] ``(3 = x) ==> x + 1 < y``;
> val it =
    |- (3 = x) ==> x + 1 < y <=> (x = 3) ==> 3 + 1 < y : thm
```

See also

`simpLib.SIMP_CONV`

NOT_CONV (reduceLib)

```
reduceLib.NOT_CONV : conv
```

Simplifies certain boolean negation expressions.

If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `NOT_CONV tm` returns the corresponding theorem.

```
   NOT_CONV "~F"  = |-  ~F = T
   NOT_CONV "~T"  = |-  ~T = F
   NOT_CONV "~~t" = |- ~~t = t
```

Failure

`NOT_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
#NOT_CONV "~~~~T";;
|- ~~~~T = ~~T

#NOT_CONV "~~T";;
|- ~~T = T

#NOT_CONV "~T";;
|- ~T = F
```

NOT_ELIM (Thm)

```
Thm.NOT_ELIM : thm -> thm
```

Transforms `|- ~t` into `|- t ==> F`.

When applied to a theorem `A |- ~t`, the inference rule `NOT_ELIM` returns the theorem `A |- t ==> F`.

```
      A |- ~t
   --------------  NOT_ELIM
    A |- t ==> F
```

Failure

Fails unless the theorem has a negated conclusion.

See also

Drule.IMP_ELIM, Thm.NOT_INTRO

NOT_EQ_SYM (Drule)

```
Drule.NOT_EQ_SYM : (thm -> thm)
```

Swaps left-hand and right-hand sides of a negated equation.

When applied to a theorem `A |- ~(t1 = t2)`, the inference rule `NOT_EQ_SYM` returns the theorem `A |- ~(t2 = t1)`.

```
 A |- ~(t1 = t2)
-----------------  NOT_EQ_SYM
 A |- ~(t2 = t1)
```

Failure

Fails unless the theorem's conclusion is a negated equation.

See also

Conv.DEPTH_CONV, Thm.REFL, Thm.SYM

NOT_EXISTS_CONV (Conv)

```
Conv.NOT_EXISTS_CONV : conv
```

Moves negation inwards through an existential quantification.

When applied to a term of the form `~(?x.P)`, the conversion `NOT_EXISTS_CONV` returns the theorem:

```
|- ~(?x.P) = !x.~P
```

Failure

Fails if applied to a term not of the form `~(?x.P)`.

See also

`Conv.EXISTS_NOT_CONV`, `Conv.FORALL_NOT_CONV`, `Conv.NOT_FORALL_CONV`

NOT_FORALL_CONV (Conv)

```
Conv.NOT_FORALL_CONV : conv
```

Moves negation inwards through a universal quantification.

When applied to a term of the form `~(!x.P)`, the conversion `NOT_FORALL_CONV` returns the theorem:

```
|- ~(!x.P) = ?x.~P
```

It is irrelevant whether `x` occurs free in `P`.

Failure

Fails if applied to a term not of the form `~(!x.P)`.

See also

`Conv.EXISTS_NOT_CONV`, `Conv.FORALL_NOT_CONV`, `Conv.NOT_EXISTS_CONV`

NOT_INTRO (Thm)

```
Thm.NOT_INTRO : (thm -> thm)
```

Transforms `|- t ==> F` into `|- ~t`.

When applied to a theorem `A |- t ==> F`, the inference rule `NOT_INTRO` returns the theorem `A |- ~t`.

```
 A |- t ==> F
--------------  NOT_INTRO
   A |- ~t
```

Failure

Fails unless the theorem has an implicative conclusion with `F` as the consequent.

See also

`Drule.IMP_ELIM`, `Thm.NOT_ELIM`

NOT_PEXISTS_CONV (PairRules)

`PairRules.NOT_PEXISTS_CONV : conv`

Moves negation inwards through a paired existential quantification.

When applied to a term of the form `~(?p. t)`, the conversion `NOT_PEXISTS_CONV` returns the theorem:

```
|- ~(?p. t) = (!p. ~t)
```

Failure

Fails if applied to a term not of the form `~(?p. t)`.

See also

`Conv.NOT_EXISTS_CONV`, `PairRules.PEXISTS_NOT_CONV`, `PairRules.PFORALL_NOT_CONV`, `PairRules.NOT_PFORALL_CONV`

NOT_PFORALL_CONV (PairRules)

```
PairRules.NOT_PFORALL_CONV : conv
```

Moves negation inwards through a paired universal quantification.

When applied to a term of the form `~(!p. t)`, the conversion `NOT_PFORALL_CONV` returns the theorem:

```
|- ~(!p. t) = (?p. ~t)
```

It is irrelevant whether any variables in `p` occur free in `t`.

Failure

Fails if applied to a term not of the form `~(!p. t)`.

See also

`Conv.NOT_FORALL_CONV`, `PairRules.PEXISTS_NOT_CONV`, `PairRules.PFORALL_NOT_CONV`, `PairRules.NOT_PEXISTS_CONV`

note_tac (goalStack)

```
goalStack.note_tac : string -> tactic
```

Prints a message; does not change the goal.

The tactic `note_tac s` prints the string `s` followed by a newline (using the standard SML `print` function). The effect on the goal is as if the tactic `ALL_TAC` had been applied (i.e., the state of the goal is not changed).

Failure

Never fails.

Comments

This is useful for tracking progress through a proof by printing messages at various stages. Unlike `print_tac`, this function only prints the provided message without printing the goal state.

See also

`goalStack.print_tac`, `Tactical.ALL_TAC`.

notify_on_word_length_guess (wordsLib)

```
wordsLib.notify_on_word_length_guess : bool ref
```

Controls notification of word length guesses.

When the reference `notify_on_word_length_guess` is true a HOL message is printed (in interactive sessions) when the function `inst_word_lengths` instantiates types in a term.

Example

```
> load "wordsLib";
val it = (): unit
> wordsLib.notify_on_word_length_guess := false;
val it = (): unit
> wordsLib.inst_word_lengths “(7 >< 5) a @@ (4 >< 0) a”;
val it = “(7 >< 5) a @@ (4 >< 0) a”: term
> type_of it;
val it = “:word8”: hol_type
```

Comments

By default `notify_on_word_length_guess` is true.

See also

`wordsLib.guess_lengths`, `wordsLib.inst_word_lengths`

NTAC (Tactic)

```
Tactic.NTAC : int -> tactic -> tactic
```

Apply tactic a specified number of times.

An invocation `NTAC n tac` applies the tactic `tac` exactly `n` times. If `n <= 0` then the goal is unchanged.

Failure

Fails if `tac` fails.

Example

Suppose we have the following goal:

```
?- x = y
```

We apply a tactic for symmetry of equality 3 times:

```
NTAC 3 (PURE_ONCE_REWRITE_TAC [EQ_SYM_EQ])
```

and obtain

```
?- y = x
```

Controlling iterated application tactics.

See also

`Rewrite.PURE_ONCE_REWRITE_TAC`, `Tactical.REPEAT`, `Conv.REPEATC`

NTH_GOAL (Tactical)

```
Tactical.NTH_GOAL : tactic -> int -> list_tactic
```

The list-tactic which applies a tactic to the n'th member of a list of goals.

If tac is a tactic, NTH_GOAL n tac is a list-tactic which applies the tactic tac to the n'th member of a list of goals.

Note that in the interactive system, subgoals are printed in reverse order of their numbering.

Failure

The application of NTH_GOAL to a tactic and integer never fails. The resulting list-tactic fails if n is less than 1 or greater than the length of the goal list, or finally if tac fails when applied to the n'th member of the goal list.

Applying a tactic to a particular subgoal.

Example

Where tac1 and tac2 are tactics, tac1 THEN_LT NTH_GOAL n tac2 applies tac1 to a goal, and then applies tac2 to the n'th resulting subgoal.

See also

Tactical.THEN_LT, Tactical.THEN1, Tactical.HEADGOAL, Tactical.LASTGOAL

Ntimes (BoundedRewrites)

```
BoundedRewrites.Ntimes : thm -> int -> thm
```

Rewriting control.

When used as an argument to the rewriter or simplifier, `Ntimes th n` is a directive saying that `th` should be used at most `n` times in the rewriting process. This is useful for controlling looping rewrites.

Failure

Never fails.

Example

Suppose factorial was defined as follows:

```
- val fact_def = Define `fact n = if n=0 then 1 else n * fact (n-1)`;
Equations stored under "fact_def".
Induction stored under "fact_ind".
> val fact_def = |- fact n = (if n = 0 then 1 else n * fact (n - 1)) : thm
```

The theorem `fact_def` is a looping rewrite since the recursive call `fac (n-1)` matches the left-hand side of `fact_def`. Thus, a naive application of the simplifier will loop:

```
- SIMP_CONV arith_ss [fact_def] ``fact 6``;
(* looping *)
> Interrupted.
```

In order to expand the definition of `fact_def` three times, the following invocation can be made

```
- SIMP_CONV arith_ss [Ntimes Fact_def 3] ``fact 6``;
> val it = |- fact 6 = 6 * (5 * (4 * fact 3)) : thm
```

Comments

Use of `Ntimes` does not compose well. For example,

```
tac1 THENL [SIMP_TAC std_ss [Ntimes th 1],
            SIMP_TAC std_ss [Ntimes th 1]]
```

is not equivalent in behaviour to

```
tac1 THEN SIMP_TAC std_ss [Ntimes th 1].
```

In the first call two rewrites using `th` can occur; in the second, only one can occur.

See also

BoundedRewrites.Once, Tactical.THEN, simpLib.SIMP_TAC, bossLib.RW_TAC, Rewrite.REWRITE_TAC

null_intersection (Lib)

```
Lib.null_intersection : ''a list -> ''a list -> bool
```

Tells if two lists have no common elements.

An invocation `null_intersection l1 l2` is equivalent to `null(intersect l1 l2)`, but is more efficient in the case where the intersection is not empty.

Failure

Never fails.

Example

```
> null_intersection [1,2,3,4] [5,6,7,8];
val it = true: bool

> null_intersection [1,2,3,4] [8,5,3];
val it = false: bool
```

Comments

A high-performance implementation of finite sets may be found in structure `HOLset`.

See also

Lib.intersect, Lib.union, Lib.U, Lib.mk_set, Lib.mem, Lib.insert, Lib.set_eq, Lib.set_diff

NULL_OK_LT (Tactical)

```
Tactical.NULL_OK_LT : list_tactic -> list_tactic
```

Makes a list-tactic succeed with no effect when applied to the empty goal list.

For any list-tactic `ltac`, the application `NULL_OK_LT ltac` gives a new list-tactic which has the same effect as `ltac` when applied to a non-empty goal list. Applied to the empty goal list it succeeds with no effect.

Failure

The application of `NULL_OK_LT` to a list-tactic `ltac` never fails. The resulting list-tactic fails if applied to a non-empty goal list on which `ltac` fails.

See also

`Tactical.ALL_LT`, `Tactical.THENL`

num_CONV (numLib)

```
numLib.num_CONV : conv
```

Equates a non-zero numeral with the form `SUC x` for some `x`.

Example

```
> numLib.num_CONV ``1203``;
val it = ⊢ 1203 = SUC 1202: thm
```

Failure

Fails if the argument term is not a numeral of type ``` ``:num`` ```, or if the argument is ``` ``0`` ```.

See also

numLib.SUC_TO_NUMERAL_DEFN_CONV

NUM_DEPTH_CONSEQ_CONV (ConseqConv)

```
ConseqConv.NUM_DEPTH_CONSEQ_CONV : directed_conseq_conv -> int -> directed_conseq_c
```

Applies a consequence conversion at most a given number of times to the sub-terms of a term, in bottom-up order.

While `DEPTH_CONSEQ_CONV c tm` applies `c` repeatedly, `NUM_DEPTH_CONSEQ_CONV c n tm` applies it at most n-times.

See also

Conv.DEPTH_CONV, ConseqConv.ONCE_DEPTH_CONSEQ_CONV, ConseqConv.DEPTH_CONSEQ_CONV, ConseqConv.DEPTH_STRENGTHEN_CONSEQ_CONV

Once (BoundedRewrites)

```
BoundedRewrites.Once : thm -> thm
```

Rewriting control.

When used as an argument to the rewriter or simplifier, `Once th` is a directive saying that `th` should be used at most once in the rewriting process. This is useful for controlling looping rewrites.

Failure

Never fails.

Example

Suppose factorial was defined as follows:

```
- val fact_def = Define `fact n = if n=0 then 1 else n * fact (n-1)`;
Equations stored under "fact_def".
Induction stored under "fact_ind".
> val fact_def = |- fact n = (if n = 0 then 1 else n * fact (n - 1)) : thm
```

The theorem `fact_def` is a looping rewrite since the recursive call `fac (n-1)` matches the left-hand side of `fact_def`. Thus, a naive application of the simplifier will loop:

```
- SIMP_CONV arith_ss [fact_def] ``fact 6``;
(* looping *)
> Interrupted.
```

In order to expand the definition of `fact_def`, the following invocation can be made

```
- SIMP_CONV arith_ss [Once fact_def] ``fact 6``;
> val it = |- fact 6 = 6 * fact 5 : thm
```

Comments

Use of `Once` does not compose well. For example,

```
tac1 THENL [SIMP_TAC std_ss [Once th],
            SIMP_TAC std_ss [Once th]]
```

is not equivalent in behaviour to

```
tac1 THEN SIMP_TAC std_ss [Once th].
```

In the first call two rewrites using `th` can occur; in the second, only one can occur.

See also

`BoundedRewrites.Ntimes`, `Tactical.THEN`, `simpLib.SIMP_TAC`, `bossLib.RW_TAC`, `Rewrite.ONCE_REWRITE_TAC`

ONCE_ASM_REWRITE_RULE (Rewrite)

```
Rewrite.ONCE_ASM_REWRITE_RULE : (thm list -> thm -> thm)
```

Rewrites a theorem once including built-in rewrites and the theorem's assumptions.

ONCE_ASM_REWRITE_RULE applies all possible rewrites in one step over the subterms in the conclusion of the theorem, but stops after rewriting at most once at each subterm. This strategy is specified as for ONCE_DEPTH_CONV. For more details see ASM_REWRITE_RULE, which does search recursively (to any depth) for matching subterms. The general strategy for rewriting theorems is described under GEN_REWRITE_RULE.

Failure

Never fails.

This tactic is used when rewriting with the hypotheses of a theorem (as well as a given list of theorems and implicit_rewrites), when more than one pass is not required or would result in divergence.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.FILTER_ASM_REWRITE_RULE, Rewrite.FILTER_ONCE_ASM_REWRITE_RULE, Rewrite.GEN_REWRITE_RULE, Conv.ONCE_DEPTH_CONV, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_ASM_REWRITE_RULE, Rewrite.PURE_ONCE_ASM_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.REWRITE_RULE

ONCE_ASM_REWRITE_TAC (Rewrite)

```
Rewrite.ONCE_ASM_REWRITE_TAC : (thm list -> tactic)
```

Rewrites a goal once including built-in rewrites and the goal's assumptions.

ONCE_ASM_REWRITE_TAC behaves in the same way as ASM_REWRITE_TAC, but makes one pass only through the term of the goal. The order in which the given theorems are applied is an implementation matter and the user should not depend on any ordering. See GEN_REWRITE_TAC for more information on rewriting a goal in HOL.

Failure

ONCE_ASM_REWRITE_TAC does not fail and, unlike ASM_REWRITE_TAC, does not diverge. The resulting tactic may not be valid, if the rewrites performed add new assumptions to the theorem eventually proved.

Example

The use of ONCE_ASM_REWRITE_TAC to control the amount of rewriting performed is illustrated below:

```
- ONCE_ASM_REWRITE_TAC []
    ([Term` (a:'a) = b`, Term `(b:'a) = c`], Term `P (a:'a): bool`);

> val it = ([([`a = b`, `b = c`], `P b`)], fn)
   : (term list * term) list * (thm list -> thm)



- (ONCE_ASM_REWRITE_TAC [] THEN ONCE_ASM_REWRITE_TAC [])
  ([Term`(a:'a) = b`, Term`(b:'a) = c`], Term `P (a:'a): bool`);

> val it = ([([`a = b`, `b = c`], `P c`)], fn)
   : (term list * term) list * (thm list -> thm)
```

ONCE_ASM_REWRITE_TAC can be applied once or iterated as required to give the effect of ASM_REWRITE_TAC, either to avoid divergence or to save inference steps.

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.FILTER_ASM_REWRITE_TAC, Rewrite.FILTER_ONCE_ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_REWRITE_TAC, Rewrite.PURE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST_TAC

ONCE_DEPTH_CONSEQ_CONV (ConseqConv)

```
ConseqConv.ONCE_DEPTH_CONSEQ_CONV : directed_conseq_conv -> directed_conseq_conv
```

Applies a consequence conversion at most once to a sub-terms of a term.

While `DEPTH_CONSEQ_CONV c tm` applies `c` repeatedly, `ONCE_DEPTH_CONSEQ_CONV c tm` applies `c` at most once.

See also

`Conv.DEPTH_CONV`, `ConseqConv.NUM_DEPTH_CONSEQ_CONV`, `ConseqConv.DEPTH_CONSEQ_CONV`, `ConseqConv.DEPTH_STRENGTHEN_CONSEQ_CONV`

ONCE_DEPTH_CONV (Conv)

```
Conv.ONCE_DEPTH_CONV : (conv -> conv)
```

Applies a conversion once to the first suitable sub-term(s) encountered in top-down order.

`ONCE_DEPTH_CONV c tm` applies the conversion `c` once to the first subterm or subterms encountered in a top-down 'parallel' search of the term `tm` for which `c` succeeds. If the conversion `c` fails on all subterms of `tm`, the theorem returned is `|- tm = tm`.

Failure

Never fails.

Example

The following example shows how `ONCE_DEPTH_CONV` applies a conversion to only the first suitable subterm(s) found in a top-down search:

```
- ONCE_DEPTH_CONV BETA_CONV (Term `(\x. (\y. y + x) 1) 2`);
> val it = |- (\x. (\y. y + x)1)2 = (\y. y + 2) 1 : thm
```

Here, there are two beta-redexes in the input term. One of these occurs within the other, so `BETA_CONV` is applied only to the outermost one.

Note that the supplied conversion is applied by ONCE_DEPTH_CONV to all independent subterms at which it succeeds. That is, the conversion is applied to every suitable subterm not contained in some other subterm for which the conversions also succeeds, as illustrated by the following example:

```
- ONCE_DEPTH_CONV numLib.num_CONV (Term `(\x. (\y. y + x) 1) 2`);
> val it = |- (\x. (\y. y + x)1)2 = (\x. (\y. y + x)(SUC 0))(SUC 1) : thm
```

Here num_CONV is applied to both 1 and 2, since neither term occurs within a larger subterm for which the conversion num_CONV succeeds.

ONCE_DEPTH_CONV is frequently used when there is only one subterm to which the desired conversion applies. This can be much faster than using other functions that attempt to apply a conversion to all subterms of a term (e.g. DEPTH_CONV). If, for example, the current goal in a goal-directed proof contains only one beta-redex, and one wishes to apply BETA_CONV to it, then the tactic

```
CONV_TAC (ONCE_DEPTH_CONV BETA_CONV)
```

may, depending on where the beta-redex occurs, be much faster than

```
CONV_TAC (TOP_DEPTH_CONV BETA_CONV)
```

ONCE_DEPTH_CONV c may also be used when the supplied conversion c never fails, in which case using a conversion such as DEPTH_CONV c, which applies c repeatedly would never terminate.

Comments

The implementation of this function uses failure to avoid rebuilding unchanged subterms. That is to say, during execution the exception QConv.UNCHANGED may be generated and later trapped. The behaviour of the function is dependent on this use of failure. So, if the conversion given as an argument happens to generate the same exception, the operation of ONCE_DEPTH_CONV will be unpredictable.

See also

Conv.DEPTH_CONV, Conv.REDEPTH_CONV, Conv.TOP_DEPTH_CONV

ONCE_REWRITE_CONV (Rewrite)

```
Rewrite.ONCE_REWRITE_CONV : (thm list -> conv)
```

Rewrites a term, including built-in tautologies in the list of rewrites.

ONCE_REWRITE_CONV searches for matching subterms and applies rewrites once at each subterm, in the manner specified for ONCE_DEPTH_CONV. The rewrites which are used are obtained from the given list of theorems and the set of tautologies stored in implicit_rewrites. See GEN_REWRITE_CONV for the general method of using theorems to rewrite a term.

Failure

ONCE_REWRITE_CONV does not fail; it does not diverge.

ONCE_REWRITE_CONV can be used to rewrite a term when recursive rewriting is not desired.

See also

Rewrite.GEN_REWRITE_CONV, Rewrite.PURE_ONCE_REWRITE_CONV, Rewrite.PURE_REWRITE_CONV, Rewrite.REWRITE_CONV

ONCE_REWRITE_RULE (Rewrite)

```
Rewrite.ONCE_REWRITE_RULE : (thm list -> thm -> thm)
```

Rewrites a theorem, including built-in tautologies in the list of rewrites.

ONCE_REWRITE_RULE searches for matching subterms and applies rewrites once at each subterm, in the manner specified for ONCE_DEPTH_CONV. The rewrites which are used are obtained from the given list of theorems and the set of tautologies stored in implicit_rewrites. See GEN_REWRITE_RULE for the general method of using theorems to rewrite an object theorem.

Failure

ONCE_REWRITE_RULE does not fail; it does not diverge.

ONCE_REWRITE_RULE can be used to rewrite a theorem when recursive rewriting is not desired.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_ASM_REWRITE_RULE, Rewrite.PURE_ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.REWRITE_RULE

ONCE_REWRITE_TAC (Rewrite)

```
Rewrite.ONCE_REWRITE_TAC : thm list -> tactic
```

Rewrites a goal only once with implicit_rewrites and the supplied list of theorems.

A set of equational rewrites is generated from the theorems supplied by the user and the set of basic tautologies, and these are used to rewrite the goal at all subterms at which a match is found in one pass over the term part of the goal. The result is returned without recursively applying the rewrite theorems to it. The order in which the given theorems are applied is an implementation matter and the user should not depend on any ordering. More details about rewriting can be found under GEN_REWRITE_TAC.

Failure

ONCE_REWRITE_TAC does not fail and does not diverge. It results in an invalid tactic if any of the applied rewrites introduces new assumptions to the theorem eventually proved.

Example

Given a theorem list:

```
thl = [ |- a = b, |- b = c, |- c = a]
```

the tactic ONCE_REWRITE_TAC thl can be iterated as required without diverging:

```
- ONCE_REWRITE_TAC thl ([], Term `P (a:'a) :bool`);
> val it = ([([], `P b`)], fn)
    : (term list * term) list * (thm list -> thm)



- (ONCE_REWRITE_TAC thl THEN ONCE_REWRITE_TAC thl)
  ([], Term `P a`);
> val it = ([([], `P c`)], fn)
    : (term list * term) list * (thm list -> thm)



- (NTAC 3 (ONCE_REWRITE_TAC thl)) ([], Term `P a`);
> val it = ([([], `P a`)], fn)
    : (term list * term) list * (thm list -> thm)
```

ONCE_REWRITE_TAC can be used iteratively to rewrite when recursive rewriting would diverge. It can also be used to save inference steps.

See also

Rewrite.ASM_REWRITE_TAC, BoundedRewrites.Once, Rewrite.ONCE_ASM_REWRITE_TAC, Rewrite.PURE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST_TAC

oneline (bossLib)

```
bossLib.oneline : thm -> thm
```

Collapse a theorem representing a single definition into a single line.

Given a theorem which consists of equations defining constants, derive a reformulation where any pattern matching clauses have been combined and replaced by a single case expression. This produces left-hand sides consisting of the constant applied only to

variables. When supplied equations for several constants (e.g., for mutually recursive functions), `oneline` returns a theorem with one equation per constant.

To advance a proof by unfolding a function defined by pattern-matching, but where the pattern is not yet constrained enough.

Example

```
> listTheory.MAP;
val it = ⊢ (∀f. MAP f [] = []) ∧ ∀f h t. MAP f (h::t) = f h::MAP f t: thm

> oneline it;
val it = ⊢ MAP f v = case v of [] => [] | h::t => f h::MAP f t: thm
```

Failure

Fails on theorems of the wrong form, including definition of multiple constants.

Comments

Shorthand for `DefnBase.one_line_ify NONE`.

See also

`bossLib.lambdify`, `bossLib.AllCaseEqs`

op_arity (Type)

```
Type.op_arity : {Thy:string, Tyop:string} -> int option
```

Return the arity of a type operator.

An invocation `op_arity{Tyop,Thy}` returns `NONE` if the given record does not identify a type operator in the current type signature. Otherwise, it returns `SOME n`, where `n` identifies the number of arguments the specified type operator takes.

Failure

Never fails.

Example

```
> op_arity{Tyop="fun", Thy="min"};
val it = SOME 2: int option

> op_arity{Tyop="foo", Thy="min"};
val it = NONE: int option
```

See also

`Type.decls`

op_insert (Lib)

```
Lib.op_insert ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list
```

Add an element to a list if it is not already there.

If there exists an element `y` in `list`, such that `eq x y`, then `insert eq x list` equals `list`. Otherwise, `x` is added to `list`.

Failure

Never fails.

Example

```
> op_insert (fn x => fn y => x = y mod 2) 1 [3,2];
val it = [3, 2]: int list

> op_insert aconv (Term `\x. x /\ y`)
                  [T, Term `\z. z /\ y`, F];
```

```
val it = [“T”, “λz. z ∧ y”, “F”]: term list

> op_insert aconv (Term `\x. x /\ y`)
                  [T, Term `\z. z /\ a`, F];
val it = [“λx. x ∧ y”, “T”, “λz. z ∧ a”, “F”]: term list
```

Comments

There is no requirement that `eq` be recognizable as a kind of equality (it could be implemented by an order relation, for example).

One should not write code that depends on the arrangement of elements in the result.

A high-performance implementation of finite sets may be found in structure `HOLset`.

See also

`Lib.insert`, `Lib.op_mem`, `Lib.op_union`, `Lib.op_mk_set`, `Lib.op_U`, `Lib.op_intersect`, `Lib.op_set_diff`

op_intersect (Lib)

```
Lib.op_intersect : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

Computes the intersection of two ‘sets’.

`op_intersect eq l1 l2` returns a list consisting of those elements of `l1` that are `eq` to some element in `l2`.

Failure

Fails if an application of `eq` fails.

Example

```
> op_intersect aconv [Term `\x:bool.x`, Term `\x y. x /\ y`]
                     [Term `\y:bool.y`, Term `\x y. x /\ z`];
val it = [“λx. x”]: term list
```

Comments

The order of items in the list returned by `op_intersect` is not dependable.

A high-performance implementation of finite sets may be found in structure `HOLset`.

There is no requirement that `eq` be recognizable as a kind of equality (it could be implemented by an order relation, for example).

See also

`Lib.intersect`, `Lib.op_mem`, `Lib.op_insert`, `Lib.op_mk_set`, `Lib.op_union`, `Lib.op_U`, `Lib.op_set_diff`

op_mem (Lib)

```
Lib.op_mem : ('a -> 'a -> bool) -> 'a -> 'a list -> bool
```

Tests whether a list contains a certain element.

An invocation `op_mem eq x [x1,...,xn]` returns `true` if, for some `xi` in the list, `eq xi x` evaluates to `true`. Otherwise it returns `false`.

Failure

Only fails if an application of `eq` fails.

Example

```
> op_mem aconv (Term `\x. x /\ y`) [T, Term `\z. z /\ y`, F];
val it = true: bool
```

Comments

A high-performance implementation of finite sets may be found in structure `HOLset`.

See also

Lib.mem, Lib.op_insert, Lib.tryfind, Lib.exists, Lib.all, Lib.assoc, Lib.rev_assoc, Lib.assoc1, Lib.assoc2, Lib.op_union, Lib.op_mk_set, Lib.op_U, Lib.op_intersect, Lib.op_set_diff

op_mk_set (Lib)

```
Lib.op_mk_set : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Transforms a list into one with elements that are distinct modulo the supplied relation.

An invocation `op_mk_set eq list` returns a list consisting of the `eq`-distinct members of `list`. In particular, the result list will not contain elements `x` and `y` at different positions such that `eq x y` evaluates to `true`.

Failure

If an application of `eq` fails when applied to two elements of `list`.

Example

```
> op_mk_set aconv [Term `\x y. x /\ y`,
                   Term `\y x. y /\ x`,
                   Term `\z a. z /\ a`];
val it = [“λz a. z ∧ a”]: term list
```

Comments

The order of items in the list returned by `op_mk_set` is not dependable.

A high-performance implementation of finite sets may be found in structure `HOLset`.

There is no requirement that `eq` be recognizable as a kind of equality (it could be implemented by an order relation, for example).

See also

Lib.mk_set, Lib.op_mem, Lib.op_insert, Lib.op_union, Lib.op_U, Lib.op_intersect, Lib.op_set_diff

op_set_diff (Lib)

```
Lib.op_set_diff : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

Computes the set-theoretic difference of two 'sets', modulo a supplied relation.

`op_set_diff eq l1 l2` returns a list consisting of those elements of `l1` that are not `eq` to some element of `l2`.

Failure

Fails if an application of `eq` fails.

Example

```
> op_set_diff (fn x => fn y => x mod 2 = y mod 2) [1,2,3] [4,5,6];
val it = []: int list

> op_set_diff (fn x => fn y => x mod 2 = y mod 2) [1,2,3] [2,4,6,8];
val it = [1, 3]: int list
```

Comments

The order in which the elements occur in the resulting list should not be depended upon.

A high-performance implementation of finite sets may be found in structure `HOLset`.

See also

Lib.set_diff, Lib.op_mem, Lib.op_insert, Lib.op_union, Lib.op_U, Lib.op_mk_set, Lib.op_intersect

op_U (Lib)

```
Lib.op_U : ('a -> 'a -> bool) -> 'a list list -> 'a list
```

Takes the union of a list of sets, modulo the supplied relation.

An application `op_U eq [l1, ..., ln]` is equivalent to `op_union eq l1 (... (op_union eq ln-1, ln)...)`. Thus, every element that occurs in one of the lists will appear in the result. However, if there are two elements `x` and `y` from different lists such that `eq x y`, then only one of `x` and `y` will appear in the result.

Failure

If an application of `eq` fails when applied to two elements from the lists.

Example

```
> op_U (fn x => fn y => x mod 2 = y mod 2)
      [[1,2,3], [4,5,6], [2,4,6,8,10]];
val it = [5, 2, 4, 6, 8, 10]: int list
```

Comments

The order in which the elements occur in the resulting list should not be depended upon.

A high-performance implementation of finite sets may be found in structure `HOLset`.

There is no requirement that `eq` be recognizable as a kind of equality (it could be implemented by an order relation, for example).

See also

`Lib.U`, `Lib.op_mem`, `Lib.op_insert`, `Lib.op_union`, `Lib.op_mk_set`, `Lib.op_intersect`, `Lib.op_set_diff`

op_union (Lib)

```
Lib.op_union : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

Computes the union of two 'sets'.

If `l1` and `l2` are both 'sets' (lists with no repeated members), `union eq l1 l2` returns the set union of `l1` and `l2`, using `eq` as the implementation of element equality. If one or both of `l1` and `l2` have repeated elements, there may be repeated elements in the result.

Failure

If some application of `eq` fails.

Example

```
> op_union (fn x => fn y => x mod 2 = y mod 2) [1,2,3] [5,4,7];
val it = [5, 4, 7]: int list
```

Comments

Do not make the assumption that the order of items in the list returned by `op_union` is fixed. Later implementations may use different algorithms, and return a different concrete result while still meeting the specification.

There is no requirement that `eq` be recognizable as a kind of equality (it could be implemented by an order relation, for example).

A high-performance implementation of finite sets may be found in structure `HOLset`.

See also

`Lib.union`, `Lib.op_mem`, `Lib.op_insert`, `Lib.op_mk_set`, `Lib.op_U`, `Lib.op_intersect`, `Lib.op_set_diff`

OR_CONV (reduceLib)

```
reduceLib.OR_CONV : conv
```

Simplifies certain boolean disjunction expressions.

If `tm` corresponds to one of the forms given below, where `t` is an arbitrary term of type `bool`, then `OR_CONV tm` returns the corresponding theorem. Note that in the last case the disjuncts need only be alpha-equivalent rather than strictly identical.

```
OR_CONV "T \/ t" = |- T \/ t = T
OR_CONV "t \/ T" = |- t \/ T = T
OR_CONV "F \/ t" = |- F \/ t = t
OR_CONV "t \/ F" = |- t \/ F = t
OR_CONV "t \/ t" = |- t \/ t = t
```

Failure

`OR_CONV tm` fails unless `tm` has one of the forms indicated above.

Example

```
#OR_CONV "F \/ T";;
|- F \/ T = T

#OR_CONV "X \/ F";;
|- X \/ F = X

#OR_CONV "(!n. n + 1 = SUC n) \/ (!m. m + 1 = SUC m)";;
|- (!n. n + 1 = SUC n) \/ (!m. m + 1 = SUC m) = (!n. n + 1 = SUC n)
```

OR_EL_CONV (listLib)

```
listLib.OR_EL_CONV : conv
```

Computes by inference the result of taking the disjunction of the elements of a boolean list.

For any object language list of the form "[x1;x2;...;xn]", where x1, x2, ..., xn are boolean expressions, the result of evaluating

```
OR_EL_CONV "OR_EL [x1;x2;...;xn]"
```

is the theorem

```
|- OR_EL [x1;x2;...;xn] = b
```

where b is either the boolean constant that denotes the disjunction of the elements of the list, or a disjunction of those xi that are not boolean constants.

Example

```
> listLib.OR_EL_CONV "OR_EL [T;F;F;T]";
val it = ⊢ OR_EL [T; F; F; T] ⇔ T: thm

> listLib.OR_EL_CONV "OR_EL [F;F;F]";
val it = ⊢ OR_EL [F; F; F] ⇔ F: thm

> listLib.OR_EL_CONV "OR_EL [F;x;y]";
val it = ⊢ OR_EL [F; x; y] ⇔ x ∨ y: thm

> listLib.OR_EL_CONV "OR_EL [x;T;y]";
val it = ⊢ OR_EL [x; T; y] ⇔ T: thm
```

Failure

OR_EL_CONV tm fails if tm is not of the form described above.

OR_EXISTS_CONV (Conv)

```
Conv.OR_EXISTS_CONV : conv
```

Moves an existential quantification outwards through a disjunction.

When applied to a term of the form `(?x.P) \/ (?x.Q)`, the conversion `OR_EXISTS_CONV` returns the theorem:

```
|- (?x.P) \/ (?x.Q) = (?x. P \/ Q)
```

Failure

Fails if applied to a term not of the form `(?x.P) \/ (?x.Q)`.

See also

`Conv.EXISTS_OR_CONV`, `Conv.LEFT_OR_EXISTS_CONV`, `Conv.RIGHT_OR_EXISTS_CONV`

OR_FORALL_CONV (Conv)

```
Conv.OR_FORALL_CONV : conv
```

Moves a universal quantification outwards through a disjunction.

When applied to a term of the form `(!x.P) \/ (!x.Q)`, where `x` is free in neither `P` nor `Q`, `OR_FORALL_CONV` returns the theorem:

```
|- (!x. P) \/ (!x. Q) = (!x. P \/ Q)
```

Failure

`OR_FORALL_CONV` fails if it is applied to a term not of the form `(!x.P) \/ (!x.Q)`, or if it is applied to a term `(!x.P) \/ (!x.Q)` in which the variable `x` is free in either `P` or `Q`.

See also

`Conv.FORALL_OR_CONV`, `Conv.LEFT_OR_FORALL_CONV`, `Conv.RIGHT_OR_FORALL_CONV`

OR_PEXISTS_CONV (PairRules)

```
PairRules.OR_PEXISTS_CONV : conv
```

Moves a paired existential quantification outwards through a disjunction.

When applied to a term of the form `(?p. t) \/ (?p. u)`, the conversion `OR_PEXISTS_CONV` returns the theorem:

```
|- (?p. t) \/ (?p. u) = (?p. t \/ u)
```

Failure

Fails if applied to a term not of the form `(?p. t) \/ (?p. u)`.

See also

`Conv.OR_EXISTS_CONV`, `PairRules.PEXISTS_OR_CONV`, `PairRules.LEFT_OR_PEXISTS_CONV`, `PairRules.RIGHT_OR_PEXISTS_CONV`

OR_PFORALL_CONV (PairRules)

```
PairRules.OR_PFORALL_CONV : conv
```

Moves a paired universal quantification outwards through a disjunction.

When applied to a term of the form `(!p. t) \/ (!p. u)`, where no variables from `p` are free in either `t` nor `u`, `OR_PFORALL_CONV` returns the theorem:

```
|- (!p. t) \/ (!p. u) = (!p. t \/ u)
```

Failure

`OR_PFORALL_CONV` fails if it is applied to a term not of the form `(!p. t) \/ (!p. u)`, or if it is applied to a term `(!p. t) \/ (!p. u)` in which the variables from `p` are free in either `t` or `u`.

See also

Conv.OR_FORALL_CONV, PairRules.PFORALL_OR_CONV, PairRules.LEFT_OR_PFORALL_-CONV, PairRules.RIGHT_OR_PFORALL_CONV

ORELSE (Tactical)

```
op Tactical.ORELSE : tactic * tactic -> tactic
```

Applies first tactic, and if it fails, applies the second instead.

If T1 and T2 are tactics, T1 ORELSE T2 is a tactic which applies T1 to a goal, and if it fails, applies T2 to the goal instead.

Failure

The application of ORELSE to a pair of tactics never fails. The resulting tactic fails if both T1 and T2 fail when applied to the relevant goal.

See also

Tactical.EVERY, Tactical.FIRST, Tactical.IF, Tactical.THEN

ORELSE_CONSEQ_CONV (ConseqConv)

```
ConseqConv.ORELSE_CONSEQ_CONV : (conseq_conv -> conseq_conv -> conseq_conv)
```

Applies the first of two consequence conversions that succeeds.

See also

Conv.ORELSEC, ConseqConv.FIRST_CONSEQ_CONV

ORELSE_LT (Tactical)

```
op Tactical.ORELSE_LT : list_tactic * list_tactic -> list_tactic
```

Applies first list-tactic, and if it fails, applies the second instead.

If `ltac1` and `ltac2` are list-tactics, `ltac1 ORELSE_LT ltac2` is a list-tactic which applies `ltac1` to a goal list, and if it fails, applies `ltac2` to the goals.

Failure

The application of `ORELSE_LT` to a pair of list-tactics never fails. The resulting list-tactic fails if both `ltac1` and `ltac2` fail when applied to the relevant goals.

See also

`Tactical.ORELSE`, `Tactical.THEN_LT`

ORELSE_TCL (Thm_cont)

```
Thm_cont.ORELSE_TCL : (thm_tactical * thm_tactical -> thm_tactical)
```

Applies a theorem-tactical, and if it fails, tries a second.

When applied to two theorem-tacticals, `ttl1` and `ttl2`, a theorem-tactic `ttac`, and a theorem `th`, if `ttl1 ttac th` succeeds, that gives the result. If it fails, the result is `ttl2 ttac th`, which may itself fail.

Failure

`ORELSE_TCL` fails if both the theorem-tacticals fail when applied to the given theorem-tactic and theorem.

See also

`Thm_cont.EVERY_TCL`, `Thm_cont.FIRST_TCL`, `Thm_cont.THEN_TCL`

ORELSEC (Conv)

```
op Conv.ORELSEC : (conv -> conv -> conv)
```

Applies the first of two conversions that succeeds.

`(c1 ORELSEC c2) ``t``` returns the result of applying the conversion `c1` to the term ``` ``t`` ``` if this succeeds. Otherwise `(c1 ORELSEC c2) ``t``` returns the result of applying the conversion `c2` to the term ``` ``t`` ```. If either conversion raises the `UNCHANGED` exception when applied, this is passed on to `ORELSEC`'s caller.

Failure

`(c1 ORELSEC c2) ``t``` fails if both `c1` and `c2` fail when applied to ``` ``t`` ```. (This refers to failure other than by raising `UNCHANGED`).

See also

`Conv.UNCHANGED`, `Conv.FIRST_CONV`

origins_of (Feedback)

```
origins_of : hol_error -> origin list
```

Extract `origin` stack from a `hol_error`.

An invocation `origins_of holerr` returns the list of `origin` elements held in `holerr`.

Example

```
> origins_of (mk_hol_error "Foo" "bar" locn.Loc_Unknown "bad input");
val it =
   [{origin_function = "bar", origin_structure = "Foo",
     source_location = <??>}]: origin list
```

Failure

Never fails.

See also

`Feedback`, `Feedback.mk_hol_error`, `Feedback.message_of`, `Feedback.top_structure_of`, `Feedback.top_function_of`

output_words_as_bin (wordsLib)

`wordsLib.output_words_as_bin : unit -> unit`

Makes word literals pretty-print as binary.

A call to `output_words_as_bin` will make word literals output in binary format.

Example

```
> wordsLib.output_words_as_bin();
val it = (): unit
> EVAL “$FCP ODD : word8”;
val it = ⊢ $FCP ODD = 0b10101010w: thm
```

See also

`wordsLib.remove_word_printer`, `wordsLib.output_words_as_dec`, `wordsLib.output_words_as_oct`, `wordsLib.output_words_as_hex`

output_words_as_dec (wordsLib)

```
wordsLib.output_words_as_dec : unit -> unit
```

Makes word literals pretty-print as decimal.

A call to `output_words_as_dec` will make word literals output in decimal format.

Example

```
> “0x100000w”;
val it = “0b10000000000000000000w”: term
> wordsLib.output_words_as_dec();
val it = (): unit
> “0x100000w”;
val it = “1048576w”: term
```

See also

`wordsLib.remove_word_printer`, `wordsLib.output_words_as_hex`, `wordsLib.output_words_as_bin`, `wordsLib.output_words_as_oct`

output_words_as_hex (wordsLib)

```
wordsLib.output_words_as_hex : unit -> unit
```

Makes word literals pretty-print as hexadecimal.

A call to `output_words_as_hex` will make word literals output in hexadecimal format.

Example

```
> wordsLib.output_words_as_hex();
val it = (): unit
> EVAL “44w : word32 << 3”
val it = ⊢ 0x2Cw ≪ 3 = 0x160w: thm
```

See also

wordsLib.remove_word_printer, wordsLib.output_words_as_dec, wordsLib.output_words_as_bin, wordsLib.output_words_as_oct

output_words_as_oct (wordsLib)

```
wordsLib.output_words_as_oct : unit -> unit
```

Makes word literals pretty-print as octal.

A call to output_words_as_oct will make word literals output in octal format.

Example

```
> “032w:word5”;
val it = “0x20w”: term
> wordsLib.output_words_as_oct();
val it = (): unit
> “032w:word5”;
val it = “032w”: term
> wordsLib.output_words_as_dec();
val it = (): unit
> “032w:word5”;
val it = “26w”: term
```

Comments

Printing and parsing in octal is controlled by the reference base_tokens.allow_octal_input. A call to output_words_as_oct sets this value to true.

See also

`wordsLib.remove_word_printer`, `wordsLib.output_words_as_dec`, `wordsLib.output_words_as_bin`, `wordsLib.output_words_as_hex`

overload_info_for (Parse)

```
Parse.overload_info_for: string -> unit
```

Prints overload information for a string.

A call to `overload_info_for s` will cause the system to print (to standard out) some information about the way in which the string `s` may be overloaded in the current global grammar. The system will print first the terms that `s` may parse to, and then the terms that might prompt the printing of `s`. Typically, both sets of terms will be the same, but they don't have to be.

Failure

Never fails.

Example

```
> overload_info_for "<=>";
<=> parses to:
  ($= :bool -> bool -> bool)
<=> might be printed from:
  ($= :bool -> bool -> bool)
val it = (): unit
```

Comments

Pretty-printed grammar values (such as returned by `term_grammar()`) include some of this information for all the constants that the grammar parses.

See also

Parse.overload_on, Parse.term_grammar

overload_on (Parse)

```
Parse.overload_on : string * term -> unit
```

Establishes a term as one of the overloading possibilities for a string.

Calling overload_on(name,tm) establishes tm as a possible resolution of the overloaded name. The call to overload_on also ensures that tm is the first in the list of possible resolutions chosen when a string might be parsed into a term in more than one way, and this is the only effect if this combination is already recorded as a possible overloading.

When printing, this call causes tm to be seen as the operator name. The string name may prompt further pretty-printing if it is involved in any of the relevant grammar's rules for concrete syntax.

If tm is an abstraction, then the parser will perform beta-reductions if the term is the function part of a redex position.

Failure

Never fails.

Example

We define the equivalent of intersection over predicates:

```
- val inter = new_definition("inter", Term`inter p q x = p x /\ q x`);
<<HOL message: inventing new type variable names: 'a.>>
> val inter = |- !p q x. inter p q x = p x /\ q x : thm
```

We overload on our new intersection constant, and can be sure that in ambiguous situations, it will be preferred:

```
- overload_on ("/\\", Term`inter`);
<<HOL message: inventing new type variable names: 'a.>>
> val it = () : unit
- Term`p /\ q`;
<<HOL message: more than one resolution of overloading was possible.>>
<<HOL message: inventing new type variable names: 'a.>>
> val it = `p /\ q` : term
- type_of it;
> val it = `:'a -> bool` : hol_type
```

Note that the original constant is considered overloaded to itself, so that our one call to `overload_on` now allows for two possibilities whenever the identifier `/\` is seen. In order to make normal conjunction the preferred choice, we can call `overload_on` with the original constant:

```
- overload_on ("/\\", Term`bool$/\`);
> val it = () : unit
- Term`p /\ q`;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = `p /\ q` : term
- type_of it;
> val it = `:bool` : hol_type
```

Note that in order to specify the original conjunction constant, we used the qualified identifier syntax, with the `$`. If we'd used just `/\`, the overloading would have ensured that this was parsed as `inter`. Instead of the qualified identifier syntax, we could have also constrained the type of conjunction explicitly so that the original constant would be the only possibility. Thus:

```
- overload_on ("/\\", Term`/\ :bool->bool->bool`);
> val it = () : unit
```

The ability to overload to abstractions allows the use of simple symbols for "complicated" effects, without needing to actually define new constants.

```
- overload_on ("|<", Term`\x y. ~(x < y)`);
> val it = () : unit

- set_fixity "|<" (Infix(NONASSOC, 450));
> val it = () : unit
```

```
- val t = Term`p |< q`;
> val t = `p |< q` : term

- dest_neg t;
> Val it = `p < q` : term
```

This facility is used to provide symbols for “is-not-equal” (<>), and “is-not-a-member” (NOTIN).

Comments

Overloading with abandon can lead to input that is very hard to make sense of, and so should be used with caution. There is a temporary version of this function: temp_overload_on.

See also

Parse.clear_overloads_on, Parse.set_fixity

p (proofManagerLib)

```
proofManagerLib.p : unit -> proof
```

Prints the top levels of the subgoal package goal stack.

The function p is part of the subgoal package. For a description of the subgoal package, see set_goal.

Failure

Never fails.

Examining the proof state during an interactive proof session.

See also

proofManagerLib.set_goal, proofManagerLib.restart, proofManagerLib.backup, proofManagerLib.redo, proofManagerLib.restore, proofManagerLib.save, proofManagerLib.set_backup, proofManagerLib.expand, proofManagerLib.expandf, proofManagerLib.p, proofManagerLib.top_thm, proofManagerLib.top_goal

P_FUN_EQ_CONV (PairRules)

```
PairRules.P_FUN_EQ_CONV : (term -> conv)
```

Performs extensionality conversion for functions (function equality).

The conversion P_FUN_EQ_CONV embodies the fact that two functions are equal precisely when they give the same results for all values to which they can be applied. For any paired variable structure "p" and equation "f = g", where p is of type ty1 and f and g are functions of type ty1->ty2, a call to P_FUN_EQ_CONV "p" "f = g" returns the theorem:

```
|- (f = g) = (!p. f p = g p)
```

Failure

P_FUN_EQ_CONV p tm fails if p is not a paired structure of variables or if tm is not an equation f = g where f and g are functions. Furthermore, if f and g are functions of type ty1->ty2, then the pair x must have type ty1; otherwise the conversion fails. Finally, failure also occurs if any of the variables in p is free in either f or g.

See also

Conv.FUN_EQ_CONV, PairRules.PEXT

P_PCHOOSE_TAC (PairRules)

```
PairRules.P_PCHOOSE_TAC : (term -> thm_tactic)
```

Assumes a theorem, with existentially quantified pair replaced by a given witness.

P_PCHOOSE_TAC expects a pair q and theorem with a paired existentially quantified conclusion. When applied to a goal, it adds a new assumption obtained by introducing the pair q as a witness for the pair p whose existence is asserted in the theorem.

```
         A ?- t
==================  P_CHOOSE_TAC "q" (A1 |- ?p. u)
 A u {u[q/p]} ?- t          ("y" not free anywhere)
```

Failure

Fails if the theorem's conclusion is not a paired existential quantification, or if the first argument is not a paired structure of variables. Failures may arise in the tactic-generating function. An invalid tactic is produced if the introduced variable is free in u or t, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

See also

Tactic.X_CHOOSE_TAC, PairRules.PCHOOSE, PairRules.PCHOOSE_THEN, PairRules.P_-PCHOOSE_THEN

P_PCHOOSE_THEN (PairRules)

```
PairRules.P_PCHOOSE_THEN : (term -> thm_tactical)
```

Replaces existentially quantified pair with given witness, and passes it to a theorem-tactic.

P_PCHOOSE_THEN expects a pair q, a tactic-generating function f:thm->tactic, and a theorem of the form (A1 |- ?p. u) as arguments. A new theorem is created by introducing the given pair q as a witness for the pair p whose existence is asserted in the original theorem, (u[q/p] |- u[q/p]). If the tactic-generating function f applied to this theorem produces results as follows when applied to a goal (A ?- u):

```
 A ?- t
=========  f ({u[q/p]} |- u[q/p])
 A ?- t1
```

then applying `(P_PCHOOSE_THEN "q" f (A1 |- ?p. u))` to the goal `(A ?- t)` produces the subgoal:

```
 A ?- t
=========  P_PCHOOSE_THEN "q" f (A1 |- ?p. u)
 A ?- t1          ("q" not free anywhere)
```

Failure

Fails if the theorem's conclusion is not existentially quantified, or if the first argument is not a paired structure of variables. Failures may arise in the tactic-generating function. An invalid tactic is produced if the introduced variable is free in `u` or `t`, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

See also

`Thm_cont.X_CHOOSE_THEN`, `PairRules.PCHOOSE`, `PairRules.PCHOOSE_THEN`, `PairRules.P_PCHOOSE_TAC`

P_PGEN_TAC (PairRules)

```
PairRules.P_PGEN_TAC : (term -> tactic)
```

Specializes a goal with the given paired structure of variables.

When applied to a paired structure of variables `p'`, and a goal `A ?- !p. t`, the tactic `P_PGEN_TAC` returns the goal `A ?- t[p'/p]`.

```
  A ?- !p. t
==============  P_PGEN_TAC "p'"
 A ?- t[p'/x]
```

Failure

Fails unless the goal's conclusion is a paired universal quantification and the term a paired structure of variables of the appropriate type. It also fails if any of the variables of the supplied structure occurs free in either the assumptions or (initial) conclusion of the goal.

See also

Tactic.X_GEN_TAC, PairRules.FILTER_PGEN_TAC, PairRules.PGEN, PairRules.PGENL, PairRules.PSPEC, PairRules.PSPECL, PairRules.PSPEC_ALL, PairRules.PSPEC_TAC

P_PSKOLEM_CONV (PairRules)

```
PairRules.P_PSKOLEM_CONV : (term -> conv)
```

Introduces a user-supplied Skolem function.

P_PSKOLEM_CONV takes two arguments. The first is a variable f, which must range over functions of the appropriate type, and the second is a term of the form !p1...pn. ?q. t (where pi and q may be pairs). Given these arguments, P_PSKOLEM_CONV returns the theorem:

```
|- (!p1...pn. ?q. t) = (?f. !p1...pn. tm[f p1 ... pn/q])
```

which expresses the fact that a skolem function f of the universally quantified variables p1...pn may be introduced in place of the the existentially quantified pair p.

Failure

P_PSKOLEM_CONV f tm fails if f is not a variable, or if the input term tm is not a term of the form !p1...pn. ?q. t, or if the variable f is free in tm, or if the type of f does not match its intended use as an n-place curried function from the pairs p1...pn to a value having the same type as p.

See also

Conv.X_SKOLEM_CONV, PairRules.PSKOLEM_CONV

PABS (PairRules)

```
PairRules.PABS : (term -> thm -> thm)
```

Paired abstraction of both sides of an equation.

```
      A |- t1 = t2
------------------------  ABS "p"            [Where p is not free in A]
 A |- (\p.t1) = (\p.t2)
```

Failure

If the theorem is not an equation, or if any variable in the paired structure of variables `p` occurs free in the assumptions `A`.

EXAMPLE

```
> PairRules.PABS (Term `(x:'a,y:'b)`) (REFL (Term `(x:'a,y:'b)`));
val it = ⊢ (λ(x,y). (x,y)) = (λ(x,y). (x,y)): thm
```

See also

`Thm.ABS`, `PairRules.PABS_CONV`, `PairRules.PETA_CONV`, `PairRules.PEXT`, `PairRules.MK_PABS`

PABS_CONV (PairRules)

```
PairRules.PABS_CONV : conv -> conv
```

Applies a conversion to the body of a paired abstraction.

If `c` is a conversion that maps a term `t` to the theorem `|- t = t'`, then the conversion `PABS_CONV c` maps abstractions of the form `\p.t` to theorems of the form:

```
|- (\p.t) = (\p.t')
```

That is, `ABS_CONV c "\p.t"` applies `p` to the body of the paired abstraction `"\p.t"`.

Failure

PABS_CONV c tm fails if tm is not a paired abstraction or if tm has the form "\p.t" but the conversion c fails when applied to the term t. The function returned by ABS_CONV p may also fail if the ML function c:term->thm is not, in fact, a conversion (i.e. a function that maps a term t to a theorem |- t = t').

Example

```
> PairRules.PABS_CONV SYM_CONV (Term `\(x,y). (1,2) = (x,y)`);
val it = ⊢ (λ(x,y). (1,2) = (x,y)) = (λ(x,y). (x,y) = (1,2)): thm
```

See also

Conv.ABS_CONV, PairRules.PSUB_CONV

pair (Lib)

```
Lib.pair : 'a -> 'b -> 'a * 'b
```

Makes two values into a pair.

pair x y returns (x, y).

Failure

Never fails.

See also

Lib.rpair, Lib.swap, Lib.fst, Lib.snd, Lib.pair_of_list, Lib.triple, Lib.quadruple, Lib.curry, Lib.uncurry

PAIR_CONV (PairRules)

```
PairRules.PAIR_CONV : (conv -> conv)
```

Applies a conversion to all the components of a pair structure.

For any conversion `c`, the function returned by `PAIR_CONV c` is a conversion that applies `c` to all the components of a pair. If the term `t` is not a pair, them `PAIR_CONV c t` applies `c` to `t`. If the term `t` is the pair `(t1,t2)` then `PAIR c t` recursively applies `PAIR_CONV c` to `t1` and `t2`.

Failure

The conversion returned by `PAIR_CONV c` will fail for the pair structure `t` if the conversion `c` would fail for any of the components of `t`.

See also

`Conv.RAND_CONV`, `Conv.RATOR_CONV`

pair_of_list (Lib)

```
Lib.pair_of_list : 'a list -> 'a * 'a
```

Turns a two-element list into a pair.

`pair_of_list [x, y]` returns `(x, y)`.

Failure

Fails if applied to a list that is not of length 2.

See also

`Lib.singleton_of_list`, `Lib.triple_of_list`, `Lib.quadruple_of_list`

pairarg_tac (bossLib)

```
bossLib.pairarg_tac : tactic
```

Adds a "splitting" equation for a pair term to goal assumptions.

A call to `pairarg_tac` will search the goal (starting with the conclusion and moving onto each assumption in turn), for a sub-term of the form `(\(x,y,...). body) arg`, where the variables appearing in `arg` are free in the goal. The tactic will then introduce a new assumption in the goal of the form

```
arg = (x,y,...)
```

where the variables `x`, `y` etc., are chosen to be as close as possible to the names in the paired abstraction. In other words, they will vary only if those names are already free in the goal.

Failure

Fails if there is no such sub-term.

Example

```
> pairarg_tac ([], ``(\(x,y). x + y) p = 10``);
val it = ([(["p = (x,y)"], "(λ(x,y). x + y) p = 10")], fn):
   goal list * validation
```

See also

`pairLib.PairCases_on`, `bossLib.split_pair_case_tac`

PairCases_on (pairLib)

```
pairLib.PairCases_on : term quotation -> tactic
```

Recursively split variables of product type.

An application `PairCases_on q` first parses `q` in the context of the goal to obtain `v`, which should be a variable of product type. Then, it introduces new variables of the form `v`n, where n is a number, representing the atomic components of `v` after all nested pair structure is expanded away. Finally, all occurrences of `v` in the goal (including in the assumptions) are replaced by the explicit pair structure (with the new variables at its leaves).

The new variables are numbered from zero according to a depth-first traversal. (Therefore, they should appear in increasing order from left to right when the tree is pretty-printed.) Primed variants of the new numbered variables are used if necessary (i.e. `v`n already occurs free in the goal).

Failure

Fails if `v` is not a variable of product type.

Example

```
val PairCases_on = pairLib.PairCases_on; g(‘(x = y) ==> ((x:((bool#bool)#bool#(bool#((bool#bo
```

Initial goal:

```
(x = y) ==> (x = z)
```

```
e(DISCH_TAC); OK.. 1 subgoal:
```

```
x = z                                                    (pairLib)
```

```
x = y
```

```
e(PairCases_on ‘y’); OK.. 1 subgoal:
```

```
x = z                                                    (pairLib)
```

```
x = ((y0,y1),y2,y3,(y4,y5),y6)
```

```
e(PairCases_on’x’); OK.. 1 subgoal:
```

((x0,x1),x2,x3,(x4,x5),x6) = z (pairLib)

((x0,x1),x2,x3,(x4,x5),x6) = ((y0,y1),y2,y3,(y4,y5),y6)

See also

Tactic.FULL_STRUCT_CASES_TAC, Conv.RENAME_VARS_CONV

PAIRED_BETA_CONV (PairedLambda)

PairedLambda.PAIRED_BETA_CONV : conv

Performs generalized beta conversion for tupled beta-redexes.

The conversion PAIRED_BETA_CONV implements beta-reduction for certain applications of tupled lambda abstractions called 'tupled beta-redexes'. Tupled lambda abstractions have the form \<vs>.tm, where <vs> is an arbitrarily-nested tuple of variables called a 'varstruct'. For the purposes of PAIRED_BETA_CONV, the syntax of varstructs is given by:

```
<vs>  ::=  (v1,v2)  |  (<vs>,v)  |  (v,<vs>)  |  (<vs>,<vs>)
```

where v, v1, and v2 range over variables. A tupled beta-redex is an application of the form (\<vs>.tm) t, where the term t is a nested tuple of values having the same structure as the varstruct <vs>. For example, the term:

```
(\((a,b),(c,d)). a + b + c + d)  ((1,2),(3,4))
```

is a tupled beta-redex, but the term:

```
(\((a,b),(c,d)). a + b + c + d)  ((1,2),p)
```

is not, since p is not a pair of terms.

Given a tupled beta-redex (\<vs>.tm) t, the conversion PAIRED_BETA_CONV performs generalized beta-reduction and returns the theorem

```
|-  (\<vs>.tm) t = t[t1,...,tn/v1,...,vn]
```

where `ti` is the subterm of the tuple `t` that corresponds to the variable `vi` in the varstruct `<vs>`. In the simplest case, the varstruct `<vs>` is flat, as in the term:

```
(\(v1,...,vn).t) (t1,...,tn)
```

When applied to a term of this form, `PAIRED_BETA_CONV` returns:

```
|- (\(v1, ... ,vn).t) (t1, ... ,tn) = t[t1,...,tn/v1,...,vn]
```

As with ordinary beta-conversion, bound variables may be renamed to prevent free variable capture. That is, the term `t[t1,...,tn/v1,...,vn]` in this theorem is the result of substituting `ti` for `vi` in parallel in `t`, with suitable renaming of variables to prevent free variables in `t1`, ..., `tn` becoming bound in the result.

Failure

`PAIRED_BETA_CONV tm` fails if `tm` is not a tupled beta-redex, as described above. Note that ordinary beta-redexes are specifically excluded: `PAIRED_BETA_CONV` fails when applied to `(\v.t)u`. For these beta-redexes, use `BETA_CONV`, or `GEN_BETA_CONV`.

Example

The following is a typical use of the conversion:

```
- PairedLambda.PAIRED_BETA_CONV
     (Term `(\((a,b),(c,d)). a + b + c + d)  ((1,2),(3,4))`);
> val it = |- (\((a,b),c,d). a+b+c+d) ((1,2),3,4) = 1+2+3+4 : thm
```

Note that the term to which the tupled lambda abstraction is applied must have the same structure as the varstruct. For example, the following succeeds:

```
- PairedLambda.PAIRED_BETA_CONV
      (Term `(\((a,b),p). a + b)  ((1,2),(3+5,4))`);
> val it = |- (\((a,b),p). a + b)((1,2),3 + 5,4) = 1 + 2 : thm
```

but the following call fails:

```
- PairedLambda.PAIRED_BETA_CONV
    (Term `(\((a,b),(c,d)). a + b + c + d)  ((1,2),p)`);
! Uncaught exception:
! HOL_ERR
```

because `p` is not a pair.

See also

Thm.BETA_CONV, Conv.BETA_RULE, Tactic.BETA_TAC, Drule.LIST_BETA_CONV, Drule.RIGHT_-BETA, Drule.RIGHT_LIST_BETA

PAIRED_ETA_CONV (PairedLambda)

PairedLambda.PAIRED_ETA_CONV : conv

Performs generalized eta conversion for tupled eta-redexes.

The conversion PAIRED_ETA_CONV generalizes ETA_CONV to eta-redexes with tupled abstractions.

```
PAIRED_ETA_CONV   \(v1..(..)..vn). f (v1..(..)..vn)
 = |- \(v1..(..)..vn). f (v1..(..)..vn) = f
```

Failure

Fails unless the given term is a paired eta-redex as illustrated above.

Comments

Note that this result cannot be achieved by ordinary eta-reduction because the tupled abstraction is a surface syntax for a term which does not correspond to a normal pattern for eta reduction. Taking the term apart reveals the true form of a paired eta redex:

```
- dest_comb (Term `\(x:num,y:num). FST (x,y)`)
> val it = (`UNCURRY`, `\x y. FST (x,y)`) : term * term
```

Example

The following is a typical use of the conversion:

```
val SELECT_PAIR_EQ = Q.prove
 (`(@(x:'a,y:'b). (a,b) = (x,y)) = (a,b)`,
  CONV_TAC (ONCE_DEPTH_CONV PairedLambda.PAIRED_ETA_CONV) THEN
  ACCEPT_TAC (SYM (MATCH_MP SELECT_AX (REFL (Term `(a:'a,b:'b)`)))));
```

See also

`Drule.ETA_CONV`

PALPHA (PairRules)

```
PairRules.PALPHA : term -> term -> thm
```

Proves equality of paired alpha-equivalent terms.

When applied to a pair of terms `t1` and `t1'` which are alpha-equivalent, `ALPHA` returns the theorem `|- t1 = t1'`.

```
-------------  PALPHA "t1" "t1'"
 |- t1 = t1'
```

The difference between `PALPHA` and `ALPHA` is that `PALPHA` is prepared to consider pair structures of different structure to be alpha-equivalent. In its most trivial case this means that `PALPHA` can consider a variable and a pair to alpha-equivalent.

Failure

Fails unless the terms provided are alpha-equivalent.

Example

```
> PairRules.PALPHA (Term `\(x:'a,y:'a). (x,y)`) (Term`\xy:'a#'a. xy`);
val it = ⊢ (λ(x,y). (x,y)) = (λxy. xy): thm
```

Comments

Alpha-converting a paired abstraction to a nonpaired abstraction can introduce instances of the terms `FST` and `SND`. A paired abstraction and a nonpaired abstraction will be considered equivalent by `PALPHA` if the nonpaired abstraction contains all those instances of `FST` and `SND` present in the paired abstraction, plus the minimum additional instances of `FST` and `SND`. For example:

```
- PALPHA
    (Term `\(x:'a,y:'b). (f x y (x,y)):'c`)
    (Term `\xy:'a#'b. (f (FST xy) (SND xy) xy):'c`);
> val it = |- (\(x,y). f x y (x,y)) = (\xy. f (FST xy) (SND xy) xy) : thm

- PALPHA
    (Term `\(x:'a,y:'b). (f x y (x,y)):'c`)
    (Term `\xy:'a#'b. (f (FST xy) (SND xy) (FST xy, SND xy)):'c`)
  handle e => Raise e;

Exception raised at ??.failwith:
PALPHA
! Uncaught exception:
! HOL_ERR
```

See also

Thm.ALPHA, Term.aconv, PairRules.PALPHA_CONV, PairRules.GEN_PALPHA_CONV

PALPHA_CONV (PairRules)

```
PairRules.PALPHA_CONV : term -> conv
```

Renames the bound variables of a paired lambda-abstraction.

If q is a variable of type ty and \p.t is a paired abstraction in which the bound pair p also has type ty, then ALPHA_CONV q "\p.t" returns the theorem:

```
|- (\p.t) = (\q'. t[q'/p])
```

where the pair q':ty is a primed variant of q chosen so that none of its components are free in \p.t. The pairs p and q need not have the same structure, but they must be of the same type.

Example

PALPHA_CONV renames the variables in a bound pair:

```
- PALPHA_CONV
    (Term `((w:'a,x:'a),(y:'a,z:'a))`)
    (Term `\((a:'a,b:'a),(c:'a,d:'a)). (f a b c d):'a`);
> val it = |- (\((a,b),c,d). f a b c d) = (\((w,x),y,z). f w x y z) : thm
```

The new bound pair and the old bound pair need not have the same structure.

```
- PALPHA_CONV
    (Term `((wx:'a#'a),(y:'a,z:'a))`)
    (Term `\((a:'a,b:'a),(c:'a,d:'a)). (f a b c d):'a`);
> val it = |- (\((a,b),c,d). f a b c d) =
              (\(wx,y,z). f (FST wx) (SND wx) y z) : thm
```

PALPHA_CONV recognises subpairs of a pair as variables and preserves structure accordingly.

```
- PALPHA_CONV
   (Term `((wx:'a#'a),(y:'a,z:'a))`)
   (Term `\((a:'a,b:'a),(c:'a,d:'a)). (f (a,b) c d):'a`);
> val it = |- (\((a,b),c,d). f (a,b) c d) = (\(wx,y,z). f wx y z) : thm
```

Comments

PALPHA_CONV will only ever add the terms FST and SND, i.e., it will never remove them. This means that while `\(x,y). x + y` can be converted to `\xy. (FST xy) + (SND xy)`, it can not be converted back again.

Failure

`PALPHA_CONV q tm` fails if `q` is not a variable, if `tm` is not an abstraction, or if `q` is a variable and `tm` is the lambda abstraction `\p.t` but the types of `p` and `q` differ.

See also

Drule.ALPHA_CONV, PairRules.PALPHA, PairRules.GEN_PALPHA_CONV

parents (Theory)

```
Theory.parents : string -> string list
```

Lists the parent theories of a named theory.

If `s` is the name of the current theory or an ancestor of the current theory, the call `parents s` returns a list of strings that identify the parent theories of `s`. The shorthand `"-"` may be used to denote the name of the current theory segment.

Failure

Fails if the named theory is not an ancestor of the current theory.

Example

```
> parents "bool";
val it = ["min"]: string list

> parents "min";
val it = []: string list

> current_theory();
val it = "foo": string

> parents "-";
val it = ["wombat"]: string list
```

See also

`Theory.ancestry`, `Theory.current_theory`

parse_from_grammars (Parse)

```
Parse.parse_from_grammars :
  (type_grammar.grammar * term_grammar.grammar) ->
  ((hol_type frag list -> hol_type) * (term frag list -> term))
```

Returns parsing functions based on the supplied grammars.

When given a pair consisting of a type and a term grammar, this function returns parsing functions that use those grammars to turn strings (strictly, quotations) into types and terms respectively.

Failure

Can't fail immediately. However, when the precedence matrix for the term parser is built on first application of the term parser, this may generate precedence conflict errors depending on the rules in the grammar.

Example

First the user loads `arithmeticTheory` to augment the built-in grammar with the ability to lex numerals and deal with symbols such as + and −:

```
- load "arithmeticTheory";
> val it = () : unit
- val t = Term`2 + 3`;
> val t = `2 + 3` : term
```

Then the `parse_from_grammars` function is used to make the values `Type` and `Term` use the grammar present in the simpler theory of booleans. Using this function fails to parse numerals or even the + infix:

```
- val (Type,Term) = parse_from_grammars boolTheory.bool_grammars;
> val Type = fn : hol_type frag list -> hol_type
  val Term = fn : term frag list -> term
- Term`2 + 3`;
<<HOL message: No numerals currently allowed.>>
! Uncaught exception:
! HOL_ERR <poly>
- Term`x + y`;
<<HOL message: inventing new type variable names: 'a, 'b.>>
> val it = `x $+ y` : term
```

But, as the last example above also demonstrates, the installed pretty-printer is still dependent on the global grammar, and the global value of `Term` can still be accessed through the `Parse` structure:

```
- t;
> val it = `2 + 3` : term

- Parse.Term`2 + 3`;
> val it = `2 + 3` : term
```

This function is used to ensure that library code has access to a term parser that is a known quantity. In particular, it is not good form to have library code that depends on the default parsers `Term` and `Type`. When the library is loaded, which may happen at any stage, these global values may be such that the parsing causes quite unexpected results or failures.

See also

`Parse.add_rule`, `Parse.print_from_grammars`, `Parse.Term`

parse_in_context (Parse)

```
Parse.parse_in_context : term list -> term quotation -> term
```

Parses a quotation into a term, using the terms as typing context.

Where the `Term` function parses a quotation in isolation of all possible contexts (except inasmuch as the global grammar provides a form of context), this function uses the additional parameter, a list of terms, to help in giving variables in the quotation types.

Thus, `Term`x`` will either guess the type ``` ``:'a`` ``` for this quotation, or refuse to parse it at all, depending on the value of the `guessing_tyvars` flag. The `parse_in_context` function, in contrast, will attempt to find a type for `x` from the list of free variables.

If the quotation already provides enough context in itself to determine a type for a variable, then the context is not consulted, and a conflicting type there for a given variable is ignored.

Failure

Fails if the quotation doesn't make syntactic sense, or if the assignment of context types to otherwise unconstrained variables in the quotation causes overloading resolution to

fail. The latter would happen if the variable `x` was given boolean type in the context, if `+` was overloaded to be over either `:num` or `:int`, and if the quotation was `x + y`.

Example

```
<< There should be an example here >>
```

Used in many of the `Q` module's variants of the standard tactics in order to have a goal provide contextual information to the parsing of arguments to tactics.

See also

`Parse.Term`

PART_MATCH (Drule)

```
Drule.PART_MATCH : (term -> term) -> thm -> term -> thm
```

Instantiates a theorem by matching part of it to a term.

When applied to a 'selector' function of type `term -> term`, a theorem and a term:

```
PART_MATCH fn (A |- !x1...xn. t) tm
```

the function `PART_MATCH` applies `fn` to `t'` (the result of specializing universally quantified variables in the conclusion of the theorem), and attempts to match the resulting term to the argument term `tm`. If it succeeds, the appropriately instantiated version of the theorem is returned.

Failure

Fails if the selector function `fn` fails when applied to the instantiated theorem, or if the match fails with the term it has provided.

Since `PART_MATCH` will not instantiate variables which appear in the hypotheses of the given theorem, it fails if the attempted match would require instantiating these variables. To allow instantiation of these variables, use `PART_MATCH_A`.

Example

Suppose that we have the following theorem:

```
th = |- !x. x==>x
```

then the following:

```
PART_MATCH (fst o dest_imp) th "T"
```

results in the theorem:

```
|- T ==> T
```

because the selector function picks the antecedent of the implication (the inbuilt specialization gets rid of the universal quantifier), and matches it to `T`.

See also

`Drule.PART_MATCH'`, `Drule.PART_MATCH_A`, `Thm.INST_TYPE`, `Drule.INST_TY_TERM`, `Term.match_term`

PART_MATCH' (Drule)

```
Drule.PART_MATCH' : (term -> term) -> thm -> term -> thm
```

Version of `PART_MATCH` that only specialises necessary variables in input

`PART_MATCH' selfn th tm` behaves similarly to `PART_MATCH selfn th tm`, except that outermost, universally quantified variables in `th` are retained in the result unless they are part of the matching.

Failure

Fails when `PART_MATCH` would fail.

Example

```
> IMP_DISJ_THM;
val it = ⊢ ∀A B. A ⇒ B ⇔ ¬A ∨ B: thm

> PART_MATCH (rand o lhs) IMP_DISJ_THM “p /\ A”;
val it = ⊢ A ⇒ p ∧ A ⇔ ¬A ∨ p ∧ A: thm

> PART_MATCH' (rand o lhs) IMP_DISJ_THM “p /\ A”;
val it = ⊢ ∀A'. A' ⇒ p ∧ A ⇔ ¬A' ∨ p ∧ A: thm
```

See also

`Drule.PART_MATCH`

PART_MATCH_A (Drule)

```
Drule.PART_MATCH_A : (term -> term) -> thm -> term -> thm
```

Instantiates a theorem by matching part of its conclusion to a term.

`PART_MATCH_A` behaves as `PART_MATCH` except that it permits instantiating variables which appear in the assumptions of the given theorem.

See also

`Drule.PART_MATCH`, `Conv.REWR_CONV_A`

PART_PMATCH (PairRules)

```
PairRules.PART_PMATCH : ((term -> term) -> thm -> term -> thm)
```

Instantiates a theorem by matching part of it to a term.

When applied to a ‘selector’ function of type `term -> term`, a theorem and a term:

```
PART_MATCH fn (A |- !p1...pn. t) tm
```

the function PART_PMATCH applies fn to t' (the result of specializing universally quantified pairs in the conclusion of the theorem), and attempts to match the resulting term to the argument term tm. If it succeeds, the appropriately instantiated version of the theorem is returned.

Failure

Fails if the selector function fn fails when applied to the instantiated theorem, or if the match fails with the term it has provided.

See also

Drule.PART_MATCH

partial (Lib)

```
Lib.partial : exn -> ('a -> 'b option) -> 'a -> 'b
```

Converts a total function to a partial function.

In ML, there are two main ways for a function to signal that it has been called on an element outside of its intended domain of application: exceptions and options. The function partial maps a function returning an element in an option type to one that may raise an exception. Thus, if f x returns NONE, then partial e f x results in the exception e being raised. If f x returns SOME y, then partial e f x returns y.

The function partial has an inverse total. Generally speaking, (partial err o total) f equals f, provided that err is the only exception that f raises. Similarly, (total o partial err) f is equal to f.

Failure

When application of the second argument to the third argument returns NONE.

Example

```
> Int.fromString "foo";
val it = NONE: int option

> partial (Fail "not convertable") Int.fromString "foo";
Exception- Fail "not convertable" raised

> (total o partial (Fail "not convertable")) Int.fromString "foo";
val it = NONE: int option
```

See also

`Lib.total`

partition (Lib)

```
Lib.partition : ('a -> bool) -> 'a list -> 'a list * 'a list
```

Split a list by a predicate.

An invocation `partition P l` divides `l` into a pair of lists `(l1,l2)`. `P` holds of each element of `l1`, and `P` does not hold of any element of `l2`.

Failure

If applying `P` to any element of `l` results in failure.

Example

```
> partition (fn i => i mod 2 = 0) [1,2,3,4,5,6,7,8,9];
val it = ([2, 4, 6, 8], [1, 3, 5, 7, 9]): int list * int list

> partition (fn _ => true) [1,2,3];
val it = ([1, 2, 3], []): int list * int list
```

```
> partition (fn _ => raise Fail "") ([]:int list);
val it = ([], []): int list * int list

> partition (fn _ => raise Fail "") [1];
Exception- Fail "" raised
```

See also

`Lib.split_after`, `Lib.pluck`

PAT_ABBREV_TAC (Q)

```
Q.PAT_ABBREV_TAC : term quotation -> tactic
```

Introduces an abbreviation within the goal statement.

When applied to the goal `(asl, w)`, the tactic `Q.PAT_ABBREV_TAC q` parses the quotation `q` in the context of the goal, producing an equality term `v = p`. The tactic then uses `HolKernel.gen_find_term` to search for a sub-term of `w` that `p` matches against. If such a sub-term `t` is found then all occurrences of `t` (in `asl` and `w`) will be replaced by `v` and an assumption `Abbrev(v = t)` is added to the goal.

The trace variable `"PAT_ABBREV_TAC: match var/const"` controls whether or not trivial matches (with constants or variables) are allowed. By default trivial matches are permitted but when the trace variable is false the tactic will ignore trivial matches, which could result in failure.

Failure

`PAT_ABBREV_TAC` fails if `q` does not successfully parse as an equality or if no matching sub-term is found in the goal, or if the only matching sub-terms would bind pattern variables to variables that are bound in the goal.

Example

If the current goal is

```
?- (n + 10) * y <= 42315
```

then applying the tactic `Q.PAT_ABBREV_TAC `X = a * b: num`` results in the goal

```
Abbrev (X = (n + 10) * y)
   ?-
X <= 42315
```

By default trivial matches are permitted. If the current goal is

```
?- y <= 42315
```

then `Q.PAT_ABBREV_TAC `X = a: num`` will give

```
Abbrev (X = y)
   ?-
X <= 42315
```

However, if this is not desirable then setting

```
Feedback.set_trace "PAT_ABBREV_TAC: match var/const" 0
```

and applying `Q.PAT_ABBREV_TAC `X = a: num`` will give

```
Abbrev (X = 42315)
   ?-
y <= X
```

If the current goal is

```
?- !x. x < 3
```

then applying `Q.PAT_ABBREV_TAC `v = (a < b)`` will result in a failure because the pattern's variable `a` would have to bind the bound variable `x` in the goal.

See also

`Q.ABBREV_TAC`, `HolKernel.gen_find_term`, `Q.MATCH_ABBREV_TAC`

PAT_ASSUM (Tactical)

```
Tactical.PAT_ASSUM : term -> thm_tactic -> tactic
```

Finds the first assumption that matches the term argument, applies the theorem tactic to it. The matching assumption remains in the assumption list.

The tactic

```
PAT_ASSUM tm ttac ([A1, ..., An], g)
```

finds the first `Ai` which matches `tm` using higher-order pattern matching in the sense of `ho_match_term`. Free variables in the pattern that are also free in the assumptions or the goal must not be bound by the match. In effect, these variables are being treated as local constants.

Failure

Fails if the term doesn't match any of the assumptions, or if the theorem-tactic fails when applied to the first assumption that does match the term.

Example

The tactic

```
PAT_ASSUM ``x:num = y`` SUBST_ALL_TAC
```

searches the assumptions for an equality over numbers and causes its right hand side to be substituted for its left hand side throughout the goal and assumptions. It also removes the equality from the assumption list. Trying to use `FIRST_ASSUM` above (i.e., replacing `PAT_ASSUM` with `FIRST_ASSUM` and dropping the term argument entirely) would require that the desired equality was the first such on the list of assumptions.

If one is trying to solve the goal

```
{ !x. f x = g (x + 1), !x. g x = f0 (f x)} ?- f x = g y
```

rewriting with the assumptions directly will cause a loop. Instead, one might want to rewrite with the formula for `f`. This can be done in an assumption-order-independent way with

```
PAT_ASSUM (Term`!x. f x = f' x`) (fn th => REWRITE_TAC [th])
```

This use of the tactic exploits higher order matching to match the RHS of the assumption, and the fact that `f` is effectively a local constant in the goal to find the correct assumption.

Comments

The behavior of `PAT_ASSUM` changed in Kananaskis 12. The old `PAT_ASSUM` (and `qpat_assum`, `Q.PAT_ASSUM`) was changed to include an extra `_X_` (or `_x_`), indicating that the matching assumption is pulled out of the assumption list. The old name now provides a version that doesn't pull the assumption out of the list.

See also

`Tactical.PAT_X_ASSUM`, `Tactical.ASSUM_LIST`, `Tactical.EVERY`, `Tactical.EVERY_ASSUM`, `Tactical.FIRST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`, `Thm_cont.UNDISCH_THEN`, `Term.match_term`

PAT_CONV (Conv)

```
Conv.PAT_CONV : term -> conv -> conv
```

Applies a conversion at specific sub-terms, following a pattern

The call `PAT_CONV ``\x1 ... xn. t[x1,...,xn]`` cnv` returns a new conversion that applies `cnv` to subterms of the target term corresponding to the free instances of any `xi` in the pattern `t[x1,...,xn]`. The fact that the pattern is a function has no logical significance; it is just used as a convenient format for the pattern.

Failure

Never fails until applied to a term, but then it may fail if the core conversion does on the chosen subterms, or if the pattern doesn't match the structure of the term.

Example

Here we choose to evaluate just two subterms:

```
> PAT_CONV ``\x. x + a + x`` numLib.REDUCE_CONV
           ``(1 + 2) + (3 + 4) + (5 + 6)``;
val it : thm = |- 1 + 2 + (3 + 4) + (5 + 6) = 3 + (3 + 4) + 11
```

while here we swap two particular quantifiers in a long chain:

```
> PAT_CONV ``\x. !x1 x2 x3 x4 x5. x`` SWAP_FORALL_CONV
           ``!a b c d e f g h. something``
<<HOL message: inventing new type variable names: ...>>
val it =
  |- (!a b c d e f g h. something) <=>
     !a b c d e g f h. something: thm
```

Comments

Multiple bound variables will only be necessary if the conversion needs to be applied to sub-terms of different types.

See also

`Conv.ABS_CONV`, `Conv.COMB_CONV`, `Conv.PATH_CONV`, `Conv.RAND_CONV`, `Conv.RATOR_CONV`, `Conv.SUB_CONV`

PAT_X_ASSUM (Tactical)

```
Tactical.PAT_X_ASSUM : term -> thm_tactic -> tactic
```

Finds the first assumption that matches the term argument, applies the theorem tactic to it, and removes this assumption.

The tactic

```
PAT_X_ASSUM tm ttac ([A1, ..., An], g)
```

finds the first `Ai` which matches `tm` using higher-order pattern matching in the sense of `ho_match_term`. Free variables in the pattern that are also free in the assumptions or the goal must not be bound by the match. In effect, these variables are being treated as local constants.

Failure

Fails if the term doesn't match any of the assumptions, or if the theorem-tactic fails when applied to the first assumption that does match the term.

See also

`Tactical.PAT_ASSUM`, `Tactical.ASSUM_LIST`, `Tactical.EVERY`, `Tactical.EVERY_ASSUM`, `Tactical.FIRST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`, `Thm_cont.UNDISCH_THEN`, `Term.match_term`

PATH_CONV (Conv)

```
Conv.PATH_CONV : string -> conv -> conv
```

Applies a conversion to the subterm indicated by a path string.

A call to `PATH_CONV p c` returns a new conversion that applies `c` to the subterm of a term identified by the path string `p`. This path string is interpreted as a sequence of direction indications: `"a"`: take the body of an abstraction; `"b"`: take the body of an abstraction or binder (such as universal or existential quantification); `"l"`: take the left (rator) path in an application; `"r"`: take the right (rand) path in an application.

Failure

The call to the path string and conversion fails if the provided string includes characters other than `a`, `b`, `l` or `r`. When applied to a term the resulting conversion will fail if the path is not meaningful or if the conversion itself fails on the indicated subterm.

Example

```
> PATH_CONV "lrr" numLib.REDUCE_CONV ``(1 + 2) + (3 + 4) + (5 + 6)``;
```

```
val it = ⊢ 1 + 2 + (3 + 4) + (5 + 6) = 1 + 2 + 7 + (5 + 6): thm

> PATH_CONV "br" numLib.REDUCE_CONV ``!x. x > 10 + 3``;
val it = ⊢ (∀x. x > 10 + 3) ⇔ ∀x. x > 13: thm
```

Comments

This function provides a more concise indication of sub-conversion application than by composing RATOR_CONV, RAND_CONV and ABS_CONV.

See also

Conv.ABS_CONV, Conv.BINDER_CONV, Conv.RAND_CONV, Conv.RATOR_CONV

PBETA_CONV (PairRules)

```
PairRules.PBETA_CONV : conv
```

Performs a general beta-conversion.

The conversion PBETA_CONV maps a paired beta-redex "(\p.t)q" to the theorem

```
|- (\p.t)q = t[q/p]
```

where u[q/p] denotes the result of substituting q for all free occurrences of p in t, after renaming sufficient bound variables to avoid variable capture. Unlike PAIRED_BETA_CONV, PBETA_CONV does not require that the structure of the argument match the structure of the pair bound by the abstraction. However, if the structure of the argument does match the structure of the pair bound by the abstraction, then PAIRED_BETA_CONV will do the job much faster.

Failure

PBETA_CONV tm fails if tm is not a paired beta-redex.

Example

PBETA_CONV will reduce applications with arbitrary structure.

```
- PBETA_CONV
     (Term `((\((a:'a,b:'a),(c:'a,d:'a)). f a b c d) ((w,x),(y,z))):'a`);
> val it = |- (\((a,b),c,d). f a b c d) ((w,x),y,z) = f w x y z : thm
```

PBETA_CONV does not require the structure of the argument and the bound pair to match.

```
- PBETA_CONV
    (Term `((\((a:'a,b:'a),(c:'a,d:'a)). f a b c d) ((w,x),yz)):'a`);
> val it = |- (\((a,b),c,d). f a b c d) ((w,x),yz) =
              f w x (FST yz) (SND yz) : thm
```

PBETA_CONV regards component pairs of the bound pair as variables in their own right and preserves structure accordingly:

```
- PBETA_CONV
    (Term `((\((a:'a,b:'a),(c:'a,d:'a)). f (a,b) (c,d)) (wx,(y,z))):'a`);
> val it = |- (\((a,b),c,d). f (a,b) (c,d)) (wx,y,z) = f wx (y,z) : thm
```

See also

Thm.BETA_CONV, PairedLambda.PAIRED_BETA_CONV, PairRules.PBETA_RULE, PairRules.PBETA_-TAC, PairRules.LIST_PBETA_CONV, PairRules.RIGHT_PBETA, PairRules.RIGHT_LIST_-PBETA, PairRules.LEFT_PBETA, PairRules.LEFT_LIST_PBETA

PBETA_RULE (PairRules)

```
PairRules.PBETA_RULE : (thm -> thm)
```

Beta-reduces all the paired beta-redexes in the conclusion of a theorem.

When applied to a theorem `A |- t`, the inference rule PBETA_RULE beta-reduces all beta-redexes, at any depth, in the conclusion `t`. Variables are renamed where necessary to avoid free variable capture.

```
 A |- ....((\p. s1) s2)....
--------------------------  BETA_RULE
   A |- ....(s1[s2/p])....
```

Failure

Never fails, but will have no effect if there are no paired beta-redexes.

See also

`Conv.BETA_RULE`, `PairRules.PBETA_CONV`, `PairRules.PBETA_TAC`, `PairRules.RIGHT_PBETA`, `PairRules.LEFT_PBETA`

PBETA_TAC (PairRules)

```
PairRules.PBETA_TAC : tactic
```

Beta-reduces all the paired beta-redexes in the conclusion of a goal.

When applied to a goal `A ?- t`, the tactic `PBETA_TAC` produces a new goal which results from beta-reducing all paired beta-redexes, at any depth, in `t`. Variables are renamed where necessary to avoid free variable capture.

```
 A ?- ...((\p. s1) s2)...
==========================  PBETA_TAC
  A ?- ...(s1[s2/p])...
```

Failure

Never fails, but will have no effect if there are no paired beta-redexes.

See also

`Tactic.BETA_TAC`, `PairRules.PBETA_CONV`, `PairRules.PBETA_RULE`

pbody (pairSyntax)

```
pairSyntax.pbody : (term -> term)
```

Returns the body of a paired abstraction.

`pbody "\pair. t"` returns `"t"`.

Failure

Fails unless the term is a paired abstraction.

See also

`Term.body`, `pairSyntax.dest_pabs`

PCHOOSE (PairRules)

```
PairRules.PCHOOSE : term * thm -> thm -> thm
```

Eliminates paired existential quantification using deduction from a particular witness.

When applied to a term-theorem pair `(q,A1 |- ?p. s)` and a second theorem of the form `A2 u {s[q/p]} |- t`, the inference rule `PCHOOSE` produces the theorem `A1 u A2 |- t`.

```
 A1 |- ?p. s   A2 u {s[q/p]} |- t
-----------------------------------  PCHOOSE ("q",(A1 |- ?q. s))
            A1 u A2 |- t
```

Where no variable in the paired variable structure `q` is free in `A1`, `A2` or `t`.

Failure

Fails unless the terms and theorems correspond as indicated above; in particular `q` must have the same type as the pair existentially quantified over, and must not contain any variable free in `A1`, `A2` or `t`.

See also

`Thm.CHOOSE`, `PairRules.PCHOOSE_TAC`, `PairRules.PEXISTS`, `PairRules.PEXISTS_TAC`, `PairRules.PSELECT_ELIM`

PCHOOSE_TAC (PairRules)

```
PairRules.PCHOOSE_TAC : thm_tactic
```

Adds the body of a paired existentially quantified theorem to the assumptions of a goal.

When applied to a theorem `A' |- ?p. t` and a goal, `CHOOSE_TAC` adds `t[p'/p]` to the assumptions of the goal, where `p'` is a variant of the pair `p` which has no components free in the assumption list; normally `p'` is just `p`.

```
        A ?- u
====================  CHOOSE_TAC (A' |- ?q. t)
 A u {t[p'/p]} ?- u
```

Unless `A'` is a subset of `A`, this is not a valid tactic.

Failure

Fails unless the given theorem is a paired existential quantification.

See also

`Tactic.CHOOSE_TAC`, `PairRules.PCHOOSE_THEN`, `PairRules.P_PCHOOSE_TAC`

PCHOOSE_THEN (PairRules)

```
PairRules.PCHOOSE_THEN : thm_tactical
```

Applies a tactic generated from the body of paired existentially quantified theorem.

When applied to a theorem-tactic `ttac`, a paired existentially quantified theorem:

```
A' |- ?p. t
```

and a goal, `CHOOSE_THEN` applies the tactic `ttac (t[p'/p] |- t[p'/p])` to the goal, where `p'` is a variant of the pair `p` chosen to have no components free in the assumption list of the goal. Thus if:

```
 A ?- s1
=========  ttac (t[q'/q] |- t[q'/q])
 B ?- s2
```

then

```
 A ?- s1
==========  CHOOSE_THEN ttac (A' |- ?q. t)
 B ?- s2
```

This is invalid unless `A'` is a subset of `A`.

Failure

Fails unless the given theorem is a paired existential quantification, or if the resulting tactic fails when applied to the goal.

See also

`Thm_cont.CHOOSE_THEN`, `PairRules.PCHOOSE_TAC`, `PairRules.P_PCHOOSE_THEN`

permahide (Parse)

```
Parse.permahide : term -> unit
```

Hide a constant so that its name doesn't overload to it.

A call to `permahide c` where `c` is a constant removes any mapping from `c`'s name to that string in the overloading map. This is done by calling `remove_ovl_mapping`, which see.

Failure

Fails if the term argument is not a constant.

Comments

This is a convenience wrapper for `remove_ovl_mapping`. It is not the same as a "permanent" form of the related `hide` function. A call to `hide s`, with `s` a string, clears all overloads to the string `s`, making that string parse to a variable when name resolution is performed. By contrast, `permahide c` only adjusts the overloading maps to and from `c`.

The intention is that `permahide` can be used in theory developments where a constant is needed but contaminating the namespace with that constant's name is not desired.

See also

`Parse.hide`, `Parse.remove_ovl_mapping`

PETA_CONV (PairRules)

```
PairRules.PETA_CONV : conv
```

Performs a top-level paired eta-conversion.

`PETA_CONV` maps an eta-redex `\p. t p`, where none of variables in the paired structure of variables `p` occurs free in `t`, to the theorem `|- (\p. t p) = t`.

Failure

Fails if the input term is not a paired eta-redex.

PEXISTENCE (PairRules)

```
PairRules.PEXISTENCE : (thm -> thm)
```

Deduces paired existence from paired unique existence.

When applied to a theorem with a paired unique-existentially quantified conclusion, `EXISTENCE` returns the same theorem with normal paired existential quantification over the same pair.

```
 A |- ?!p. t
-------------  PEXISTENCE
 A |- ?p. t
```

Failure

Fails unless the conclusion of the theorem is a paired unique-existential quantification.

See also

`Conv.EXISTENCE`, `PairRules.PEXISTS_UNIQUE_CONV`

PEXISTS (PairRules)

```
PairRules.PEXISTS : term * term -> thm -> thm
```

Introduces paired existential quantification given a particular witness.

When applied to a pair of terms and a theorem, where the first term a paired existentially quantified pattern indicating the desired form of the result, and the second a witness whose substitution for the quantified pair gives a term which is the same as the conclusion of the theorem, `PEXISTS` gives the desired theorem.

```
 A |- t[q/p]
-------------  EXISTS ("?p. t","q")
 A |- ?p. t
```

Failure

Fails unless the substituted pattern is the same as the conclusion of the theorem.

Example

The following examples illustrate the various uses of PEXISTS:

```
- PEXISTS (Term`?x. x + 2 = x + 2`, Term`1`) (REFL (Term`1 + 2`));
> val it = |- ?x. x + 2 = x + 2 : thm

- PEXISTS (Term`?y. 1 + y = 1 + y`, Term`2`) (REFL (Term`1 + 2`));
> val it = |- ?y. 1 + y = 1 + y : thm

- PEXISTS (Term`?(x,y). x + y = x + y`, Term`(1,2)`) (REFL (Term`1 + 2`));
> val it = |- ?(x,y). x + y = x + y : thm

- PEXISTS (Term`?(a:'a,b:'a). (a,b) = (a,b)`, Term`ab:'a#'a`)
          (REFL (Term `ab:'a#'a`));
> val it = |- ?(a,b). (a,b) = (a,b) : thm
```

See also

Thm.EXISTS, PairRules.PCHOOSE, PairRules.PEXISTS_TAC

PEXISTS_AND_CONV (PairRules)

```
PairRules.PEXISTS_AND_CONV : conv
```

Moves a paired existential quantification inwards through a conjunction.

When applied to a term of the form `?p. t /\ u`, where variables in `p` are not free in both `t` and `u`, PEXISTS_AND_CONV returns a theorem of one of three forms, depending on

occurrences of variables from p in t and u. If p contains variables free in t but none in u, then the theorem:

```
|- (?p. t /\ u) = (?p. t) /\ u
```

is returned. If p contains variables free in u but none in t, then the result is:

```
|- (?p. t /\ u) = t /\ (?x. u)
```

And if p does not contain any variable free in either t nor u, then the result is:

```
|- (?p. t /\ u) = (?x. t) /\ (?x. u)
```

Failure

PEXISTS_AND_CONV fails if it is applied to a term not of the form ?p. t /\ u, or if it is applied to a term ?p. t /\ u in which variables in p are free in both t and u.

See also

Conv.EXISTS_AND_CONV, PairRules.AND_PEXISTS_CONV, PairRules.LEFT_AND_PEXISTS_CONV, PairRules.RIGHT_AND_PEXISTS_CONV

PEXISTS_CONV (PairRules)

PairRules.PEXISTS_CONV : conv

Eliminates paired existential quantifier by introducing a paired choice-term.

The conversion PEXISTS_CONV expects a boolean term of the form (?p. t[p]), where p may be a paired structure or variables, and converts it to the form (t [@p. t[p]]).

```
--------------------------------  PEXISTS_CONV "(?p. t[p])"
 (|- (?p. t[p]) = (t [@p. t[p]])
```

Failure

Fails if applied to a term that is not a paired existential quantification.

See also

PairRules.PSELECT_RULE, PairRules.PSELECT_CONV, PairRules.PEXISTS_RULE, PairRules.PSEL
INTRO, PairRules.PSELECT_ELIM

PEXISTS_EQ (PairRules)

```
PairRules.PEXISTS_EQ : (term -> thm -> thm)
```

Existentially quantifies both sides of an equational theorem.

When applied to a paired structure of variables p and a theorem whose conclusion is equational:

```
A |- t1 = t2
```

the inference rule PEXISTS_EQ returns the theorem:

```
A |- (?p. t1) = (?p. t2)
```

provided the none of the variables in p is not free in any of the assumptions.

```
       A |- t1 = t2
--------------------------  PEXISTS_EQ "p"      [where p is not free in A]
 A |- (?p. t1) = (?p. t2)
```

Failure

Fails unless the theorem is equational with both sides having type bool, or if the term is not a paired structure of variables, or if any variable in the pair to be quantified over is free in any of the assumptions.

See also

Drule.EXISTS_EQ, PairRules.PEXISTS_IMP, PairRules.PFORALL_EQ, PairRules.MK_-PEXISTS, PairRules.PSELECT_EQ

PEXISTS_IMP (PairRules)

```
PairRules.PEXISTS_IMP : (term -> thm -> thm)
```

Existentially quantifies both the antecedent and consequent of an implication.

When applied to a paired structure of variables p and a theorem `A |- t1 ==> t2`, the inference rule PEXISTS_IMP returns the theorem `A |- (?p. t1) ==> (?p. t2)`, provided no variable in p is free in the assumptions.

```
       A |- t1 ==> t2
--------------------------  EXISTS_IMP "x"   [where x is not free in A]
 A |- (?x.t1) ==> (?x.t2)
```

Failure

Fails if the theorem is not implicative, or if the term is not a paired structure of variables, of if any variable in the pair is free in the assumption list.

See also

Drule.EXISTS_IMP, PairRules.PEXISTS_EQ

PEXISTS_IMP_CONV (PairRules)

```
PairRules.PEXISTS_IMP_CONV : conv
```

Moves a paired existential quantification inwards through an implication.

When applied to a term of the form `?p. t ==> u`, where variables from `p` are not free in both `t` and `u`, `PEXISTS_IMP_CONV` returns a theorem of one of three forms, depending on occurrences of variable from `p` in `t` and `u`. If variables from `p` are free in `t` but none are in `u`, then the theorem:

```
|- (?p. t ==> u) = (!p. t) ==> u
```

is returned. If variables from `p` are free in `u` but none are in `t`, then the result is:

```
|- (?p. t ==> u) = t ==> (?p. u)
```

And if no variable from `p` is free in either `t` nor `u`, then the result is:

```
|- (?p. t ==> u) = (!p. t) ==> (?p. u)
```

Failure

`PEXISTS_IMP_CONV` fails if it is applied to a term not of the form `?p. t ==> u`, or if it is applied to a term `?p. t ==> u` in which the variables from `p` are free in both `t` and `u`.

See also

`Conv.EXISTS_IMP_CONV`, `PairRules.LEFT_IMP_PFORALL_CONV`, `PairRules.RIGHT_IMP_-PEXISTS_CONV`

PEXISTS_NOT_CONV (PairRules)

`PairRules.PEXISTS_NOT_CONV : conv`

Moves a paired existential quantification inwards through a negation.

When applied to a term of the form `?p. ~t`, the conversion `PEXISTS_NOT_CONV` returns the theorem:

```
|- (?p. ~t) = ~(!p. t)
```

Failure

Fails if applied to a term not of the form `?p. ~t`.

See also

`Conv.EXISTS_NOT_CONV`, `PairRules.PFORALL_NOT_CONV`, `PairRules.NOT_PEXISTS_CONV`, `PairRules.NOT_PFORALL_CONV`

PEXISTS_OR_CONV (PairRules)

```
PairRules.PEXISTS_OR_CONV : conv
```

Moves a paired existential quantification inwards through a disjunction.

When applied to a term of the form `?p. t \/ u`, the conversion `PEXISTS_OR_CONV` returns the theorem:

```
|- (?p. t \/ u) = (?p. t) \/ (?p. u)
```

Failure

Fails if applied to a term not of the form `?p. t \/ u`.

See also

`Conv.EXISTS_OR_CONV`, `PairRules.OR_PEXISTS_CONV`, `PairRules.LEFT_OR_PEXISTS_CONV`, `PairRules.RIGHT_OR_PEXISTS_CONV`

PEXISTS_RULE (PairRules)

```
PairRules.PEXISTS_RULE : (thm -> thm)
```

Introduces a paired existential quantification in place of a paired choice.

The inference rule PEXISTS_RULE expects a theorem asserting that (@p. t) denotes a pair for which t holds. The equivalent assertion that there exists a p for which t holds is returned.

```
 A |- t[(@p. t)/p]
-------------------  PEXISTS_RULE
    A |- ?p. t
```

Failure

Fails if applied to a theorem the conclusion of which is not of the form (t[(@p.t)/p]).

See also

PairRules.PEXISTS_CONV, PairRules.PSELECT_RULE, PairRules.PSELECT_CONV, PairRules.PSELECT_INTRO, PairRules.PSELECT_ELIM

PEXISTS_TAC (PairRules)

```
PairRules.PEXISTS_TAC : (term -> tactic)
```

Reduces paired existentially quantified goal to one involving a specific witness.

When applied to a term q and a goal ?p. t, the tactic PEXISTS_TAC reduces the goal to t[q/p].

```
 A ?- ?p. t
=============  EXISTS_TAC "q"
 A ?- t[q/p]
```

Failure

Fails unless the goal's conclusion is a paired existential quantification and the term supplied has the same type as the quantified pair in the goal.

Example

The goal:

```
?- ?(x,y). (x,y)=(1,2)
```

can be solved by:

```
PEXISTS_TAC "(1,2)" THEN REFL_TAC
```

See also

`Tactic.EXISTS_TAC`, `PairRules.PEXISTS`

PEXISTS_UNIQUE_CONV (PairRules)

`PairRules.PEXISTS_UNIQUE_CONV : conv`

Expands with the definition of paired unique existence.

Given a term of the form `"?!p. t[p]"`, the conversion `PEXISTS_UNIQUE_CONV` proves that this assertion is equivalent to the conjunction of two statements, namely that there exists at least one pair `p` such that `t[p]`, and that there is at most one value `p` for which `t[p]` holds. The theorem returned is:

```
|- (?!p. t[p]) = (?p. t[p]) /\ (!p p'. t[p] /\ t[p'] ==> (p = p'))
```

where `p'` is a primed variant of the pair `p` none of the components of which appear free in the input term. Note that the quantified pair `p` need not in fact appear free in the body of the input term. For example, `PEXISTS_UNIQUE_CONV "?!(x,y). T"` returns the theorem:

```
|- (?!(x,y). T) =
   (?(x,y). T) /\ (!(x,y) (x',y'). T /\ T ==> ((x,y) = (x',y')))
```

Failure

`PEXISTS_UNIQUE_CONV tm` fails if `tm` does not have the form `"?!p.t"`.

See also

Conv.EXISTS_UNIQUE_CONV, PairRules.PEXISTENCE

PEXT (PairRules)

```
PairRules.PEXT : (thm -> thm)
```

Derives equality of functions from extensional equivalence.

When applied to a theorem `A |- !p. t1 p = t2 p`, the inference rule `PEXT` returns the theorem `A |- t1 = t2`.

```
 A |- !p. t1 p = t2 p
---------------------  PEXT          [where p is not free in t1 or t2]
     A |- t1 = t2
```

Failure

Fails if the theorem does not have the form indicated above, or if any of the component variables in the paired variable structure `p` is free either of the functions `t1` or `t2`.

Example

```
> PairRules.PEXT (ASSUME (Term `!(x,y). ((f:('a#'a)->'a) (x,y)) = (g (x,y))`));
val it =  [.] ⊢ f = g: thm
```

See also

Drule.EXT, Thm.AP_THM, PairRules.PETA_CONV, Conv.FUN_EQ_CONV, PairRules.P_-FUN_EQ_CONV

PFORALL_AND_CONV (PairRules)

```
PairRules.PFORALL_AND_CONV : conv
```

Moves a paired universal quantification inwards through a conjunction.

When applied to a term of the form `!p. t /\ u`, the conversion `PFORALL_AND_CONV` returns the theorem:

```
|- (!p. t /\ u) = (!p. t) /\ (!p. u)
```

Failure

Fails if applied to a term not of the form `!p. t /\ u`.

See also

`Conv.FORALL_AND_CONV`, `PairRules.AND_PFORALL_CONV`, `PairRules.LEFT_AND_PFORALL_CONV`, `PairRules.RIGHT_AND_PFORALL_CONV`

PFORALL_EQ (PairRules)

```
PairRules.PFORALL_EQ : (term -> thm -> thm)
```

Universally quantifies both sides of an equational theorem.

When applied to a paired structure of variables `p` and a theorem

```
A |- t1 = t2
```

whose conclusion is an equation between boolean terms:

```
PFORALL_EQ
```

returns the theorem:

```
A |- (!p. t1) = (!p. t2)
```

unless any of the variables in `p` is free in any of the assumptions.

```
        A |- t1 = t2
--------------------------  PFORALL_EQ "p"      [where p is not free in A]
 A |- (!p. t1) = (!p. t2)
```

Failure

Fails if the theorem is not an equation between boolean terms, or if the supplied term is not a paired structure of variables, or if any of the variables in the supplied pair is free in any of the assumptions.

See also

`Drule.FORALL_EQ`, `PairRules.PEXISTS_EQ`, `PairRules.PSELECT_EQ`

PFORALL_IMP_CONV (PairRules)

```
PairRules.PFORALL_IMP_CONV : conv
```

Moves a paired universal quantification inwards through an implication.

When applied to a term of the form `!p. t ==> u`, where variables from `p` are not free in both `t` and `u`, `PFORALL_IMP_CONV` returns a theorem of one of three forms, depending on occurrences of the variables from `p` in `t` and `u`. If variables from `p` are free in `t` but none are in `u`, then the theorem:

```
|- (!p. t ==> u) = (?p. t) ==> u
```

is returned. If variables from `p` are free in `u` but none are in `t`, then the result is:

```
|- (!p. t ==> u) = t ==> (!p. u)
```

And if no variable from `p` is free in either `t` nor `u`, then the result is:

```
|- (!p. t ==> u) = (?p. t) ==> (!p. u)
```

Failure

`PFORALL_IMP_CONV` fails if it is applied to a term not of the form `!p. t ==> u`, or if it is applied to a term `!p. t ==> u` in which variables from `p` are free in both `t` and `u`.

See also

Conv.FORALL_IMP_CONV, PairRules.LEFT_IMP_PEXISTS_CONV, PairRules.RIGHT_IMP_-PFORALL_CONV

PFORALL_NOT_CONV (PairRules)

```
PairRules.PFORALL_NOT_CONV : conv
```

Moves a paired universal quantification inwards through a negation.

When applied to a term of the form `!p. ~t`, the conversion PFORALL_NOT_CONV returns the theorem:

```
|- (!p. ~t) = ~(?p. t)
```

Failure

Fails if applied to a term not of the form `!p. ~t`.

See also

Conv.FORALL_NOT_CONV, PairRules.PEXISTS_NOT_CONV, PairRules.NOT_PEXISTS_-CONV, PairRules.NOT_PFORALL_CONV

PFORALL_OR_CONV (PairRules)

```
PairRules.PFORALL_OR_CONV : conv
```

Moves a paired universal quantification inwards through a disjunction.

When applied to a term of the form `!p. t \/ u`, where no variable in `p` is free in both `t` and `u`, PFORALL_OR_CONV returns a theorem of one of three forms, depending on occurrences of the variables from `p` in `t` and `u`. If variables from `p` are free in `t` but not in `u`, then the theorem:

```
|- (!p. t \/ u) = (!p. t) \/ u
```

is returned. If variables from p are free in u but none are free in t, then the result is:

```
|- (!p. t \/ u) = t \/ (!t. u)
```

And if no variable from p is free in either t nor u, then the result is:

```
|- (!p. t \/ u) = (!p. t) \/ (!p. u)
```

Failure

PFORALL_OR_CONV fails if it is applied to a term not of the form `!p. t \/ u`, or if it is applied to a term `!p. t \/ u` in which variables from p are free in both t and u.

See also

Conv.FORALL_OR_CONV, PairRules.OR_PFORALL_CONV, PairRules.LEFT_OR_PFORALL_CONV, PairRules.RIGHT_OR_PFORALL_CONV

PGEN (PairRules)

```
PairRules.PGEN : (term -> thm -> thm)
```

Generalizes the conclusion of a theorem.

When applied to a paired structure of variables p and a theorem `A |- t`, the inference rule PGEN returns the theorem `A |- !p. t`, provided that no variable in p occurs free in the assumptions A. There is no compulsion that the variables of p should be free in t.

```
    A |- t
------------  PGEN "p"              [where p does not occur free in A]
 A |- !p. t
```

Failure

Fails if p is not a paired structure of variables, of if any variable in p is free in the assumptions.

See also

`Thm.GEN`, `PairRules.PGENL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`

PGEN_TAC (PairRules)

```
PairRules.PGEN_TAC : tactic
```

Strips the outermost paired universal quantifier from the conclusion of a goal.

When applied to a goal `A ?- !p. t`, the tactic `PGEN_TAC` reduces it to `A ?- t[p'/p]` where `p'` is a variant of the paired variable structure `p` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `p'` is just `p`.

```
  A ?- !p. t
==============  PGEN_TAC
 A ?- t[p'/p]
```

Failure

Fails unless the goal's conclusion is a paired universally quantification.

See also

`Tactic.GEN_TAC`, `PairRules.FILTER_PGEN_TAC`, `PairRules.PGEN`, `PairRules.PGENL`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`, `PairRules.PSTRIP_TAC`, `PairRules.P_PGEN_TAC`

PGENL (PairRules)

```
PairRules.PGENL : (term list -> thm -> thm)
```

Generalizes zero or more pairs in the conclusion of a theorem.

When applied to a list of paired variable structures `[p1;...;pn]` and a theorem `A |- t`, the inference rule `PGENL` returns the theorem `A |- !p1...pn. t`, provided none of the constituent variables from any of the pairs `pi` occur free in the assumptions.

```
       A |- t
------------------  PGENL "[p1;...;pn]"       [where no pi is free in A]
 A |- !p1...pn. t
```

Failure

Fails unless all the terms in the list are paired structures of variables, none of the variables from which are free in the assumption list.

See also

`Thm.GENL`, `PairRules.PGEN`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`

|> (Lib)

```
op Lib.|> : 'a -> ('a -> 'b) -> 'b
```

Infix operator for writing function application.

The expression `x |> f` is equal to `f x`. This way of writing application has two advantages, both apparent when multiple functions are being applied. Without using `|>`, one might write `f (g (h x))`. With it, one writes `x |> h |> g |> f`. The latter form needs fewer parentheses, and also makes the order in which functions will operate correspond to a left-to-right reading.

Failure

Never fails.

pluck (Lib)

```
Lib.pluck : ('a -> bool) -> 'a list -> 'a * 'a list
```

Pull an element out of a list.

An invocation `pluck P [x1,...,xk,...,xn]` returns a pair `(xk,[x1,...,xk-1,xk+1,...xn])`, where `xk` has been lifted out of the list without disturbing the relative positions of the other elements. For this to happen, `P xk` must hold, and `P xi` must not have held for `x1,...,xk-1`.

Failure

If the input list is empty. Also fails if `P` holds of no member of the list. Also fails if an application of `P` fails.

Example

```
> val (x,rst) = pluck (fn x => x mod 2 = 0) [1,2,3];
val rst = [1, 3]: int list
val x = 2: int
```

See also

`Lib.first`, `Lib.filter`, `Lib.mapfilter`, `Lib.assoc1`, `Lib.assoc2`, `Lib.assoc`, `Lib.rev_assoc`

++ (bossLib)

```
op bossLib.++ : simpset * ssfrag -> simpset
```

Infix operator for augmenting simpsets with `ssfrag` values.

The ++ function combines its two arguments and creates a new simpset. This is a way of creating simpsets that are tailored to the particular simplification task at hand.

Failure

Never fails.

Example

Here we add the `UNWIND_ss` `ssfrag` value to the `pure_ss` simpset to exploit the former's point-wise elimination conversions.

```
> SIMP_CONV (pureSimps.pure_ss ++ boolSimps.UNWIND_ss) []
           (Term`!x. x ==> (?y. P(x,y) /\ (y = 5))`);
val it = ⊢ (∀x. x ⇒ ∃y. P (x,y) ∧ y = 5) ⇔ P (T,5): thm
```

See also

`simpLib.mk_simpset`, `simpLib.rewrites`, `simpLib.SIMP_CONV`, `pureSimps.pure_ss`

PMATCH_MP (PairRules)

```
PairRules.PMATCH_MP : (thm -> thm -> thm)
```

Modus Ponens inference rule with automatic matching.

When applied to theorems `A1 |- !p1...pn. t1 ==> t2` and `A2 |- t1'`, the inference rule `PMATCH_MP` matches `t1` to `t1'` by instantiating free or paired universally quantified variables in the first theorem (only), and returns a theorem `A1 u A2 |- !pa..pk. t2'`, where `t2'` is a correspondingly instantiated version of `t2`. Polymorphic types are also instantiated if necessary.

Variables free in the consequent but not the antecedent of the first argument theorem will be replaced by variants if this is necessary to maintain the full generality of the theorem, and any pairs which were universally quantified over in the first argument theorem will be universally quantified over in the result, and in the same order.

```
   A1 |- !p1..pn. t1 ==> t2   A2 |- t1'
  ------------------------------------  MATCH_MP
         A1 u A2 |- !pa..pk. t2'
```

Failure

Fails unless the first theorem is a (possibly repeatedly paired universally quantified) implication whose antecedent can be instantiated to match the conclusion of the second theorem, without instantiating any variables which are free in `A1`, the first theorem's assumption list.

See also

`Drule.MATCH_MP`

PMATCH_MP_TAC (PairRules)

`PairRules.PMATCH_MP_TAC : thm_tactic`

Reduces the goal using a supplied implication, with matching.

When applied to a theorem of the form

```
A' |- !p1...pn. s ==> !q1...qm. t
```

`PMATCH_MP_TAC` produces a tactic that reduces a goal whose conclusion `t'` is a substitution and/or type instance of `t` to the corresponding instance of `s`. Any variables free in `s` but not in `t` will be existentially quantified in the resulting subgoal:

```
    A ?- !u1...ui. t'
======================  PMATCH_MP_TAC (A' |- !p1...pn. s ==> !q1...qm. t)
    A ?- ?w1...wp. s'
```

where `w1`, ..., `wp` are (type instances of) those pairs among `p1`, ..., `pn` having variables that do not occur free in `t`. Note that this is not a valid tactic unless `A'` is a subset of `A`.

Failure

Fails unless the theorem is an (optionally paired universally quantified) implication whose consequent can be instantiated to match the goal. The generalized pairs `u1`, ..., `ui` must occur in `s'` in order for the conclusion `t` of the supplied theorem to match `t'`.

See also

`Tactic.MATCH_MP_TAC`

polymorphic (Type)

```
Type.polymorphic : hol_type -> bool
```

Checks if there is a type variable in a type.

An invocation `polymorphic ty` checks to see if `ty` has an occurrence of any type variable. It is equivalent in functionality to `not o null o type_vars`, but may be more efficient in some situations, since it can stop processing once it finds one type variable.

Failure

Never fails.

Example

```
> polymorphic (bool --> alpha --> ind);
val it = true: bool
```

Comments

`polymorphic` is also equivalent to `exists_tyvar (K true)`, and no faster.

See also

`Type.type_vars`, `Type.type_var_in`, `Type.exists_tyvar`

POP_ASSUM (Tactical)

```
Tactical.POP_ASSUM : thm_tactic -> tactic
```

Applies tactic generated from the first element of a goal's assumption list.

When applied to a theorem-tactic and a goal, POP_ASSUM applies the theorem-tactic to the ASSUMEd first element of the assumption list, and applies the resulting tactic to the goal without the first assumption in its assumption list:

```
POP_ASSUM f ({A1,...,An} ?- t) = f (A1 |- A1) ({A2,...,An} ?- t)
```

Failure

Fails if the assumption list of the goal is empty, or the theorem-tactic fails when applied to the popped assumption, or if the resulting tactic fails when applied to the goal (with depleted assumption list).

Comments

It is possible simply to use the theorem ASSUME A1 as required rather than use POP_ASSUM; this will also maintain A1 in the assumption list, which is generally useful. In addition, this approach can equally well be applied to assumptions other than the first.

There are admittedly times when POP_ASSUM is convenient, but it is most unwise to use it if there is more than one assumption in the assumption list, since this introduces a dependency on the ordering, which is vulnerable to changes in the HOL system.

Another point to consider is that if the relevant assumption has been obtained by DISCH_TAC, it is often cleaner to use DISCH_THEN with a theorem-tactic. For example, instead of:

```
DISCH_TAC THEN POP_ASSUM (SUBST1_TAC o SYM)
```

one might use

```
DISCH_THEN (SUBST1_TAC o SYM)
```

The tactical POP_ASSUM is also available under the lower-case version of the name, pop_assum.

Example

The goal:

```
{4 = SUC x} ?- x = 3
```

can be solved by:

```
POP_ASSUM
 (fn th => REWRITE_TAC[REWRITE_RULE[num_CONV “4”, INV_SUC_EQ] th])
```

Making more delicate use of an assumption than rewriting or resolution using it.

See also

Tactical.ASSUM_LIST, Tactical.EVERY_ASSUM, Tactic.IMP_RES_TAC, Tactical.POP_ASSUM_LIST, Tactical.POP_LAST_ASSUM, Rewrite.REWRITE_TAC

POP_ASSUM_LIST (Tactical)

```
Tactical.POP_ASSUM_LIST : (thm list -> tactic) -> tactic
```

Generates a tactic from the assumptions, discards the assumptions and applies the tactic.

When applied to a function and a goal, POP_ASSUM_LIST applies the function to a list of theorems corresponding to the ASSUMEd assumptions of the goal, then applies the resulting tactic to the goal with an empty assumption list.

```
POP_ASSUM_LIST f ({A1,...,An} ?- t) = f [A1 |- A1, ..., An |- An] (?- t)
```

Failure

Fails if the function fails when applied to the list of ASSUMEd assumptions, or if the resulting tactic fails when applied to the goal with no assumptions.

Comments

There is nothing magical about POP_ASSUM_LIST: the same effect can be achieved by using ASSUME a explicitly wherever the assumption a is used. If POP_ASSUM_LIST is used, it is unwise to select elements by number from the ASSUMEd-assumption list, since this introduces a dependency on ordering.

Example

Suppose we have a goal of the following form:

```
{a /\ b, c, (d /\ e) /\ f} ?- t
```

Then we can split the conjunctions in the assumption list apart by applying the tactic:

```
POP_ASSUM_LIST (MAP_EVERY STRIP_ASSUME_TAC)
```

which results in the new goal:

```
{a, b, c, d, e, f} ?- t
```

Making more delicate use of the assumption list than simply rewriting or using resolution.

See also

Tactical.ASSUM_LIST, Tactical.EVERY_ASSUM, Tactic.IMP_RES_TAC, Tactical.POP_ASSUM, Rewrite.REWRITE_TAC

POP_LAST_ASSUM (Tactical)

```
Tactical.POP_LAST_ASSUM : thm_tactic -> tactic
```

Applies tactic generated from the last element of a goal's assumption list.

When applied to a theorem-tactic and a goal, POP_LAST_ASSUM applies the theorem-tactic to the ASSUMEd last element of the assumption list, and applies the resulting tactic to the goal without that assumption in its assumption list:

```
POP_LAST_ASSUM f ({A1,...,Am,An} ?- t) = f (An |- An) ({A1,...,Am} ?- t)
```

Failure

Fails if the assumption list of the goal is empty, or the theorem-tactic fails when applied to the popped assumption, or if the resulting tactic fails when applied to the goal (with depleted assumption list).

Comments

The tactical `POP_LAST_ASSUM` is also available under the lower-case version of the name, `pop_last_assum`.

See also

`Tactical.ASSUM_LIST`, `Tactical.EVERY_ASSUM`, `Tactic.IMP_RES_TAC`, `Tactical.POP_ASSUM`, `Rewrite.REWRITE_TAC`

pp (Drule)

```
Drule.pp : thmpos_dtype.match_position -> thm -> thm
```

Promotes the designated premise to the "top" of an implicational theorem.

A call to `pp pos th` finds the premise denoted by `pos` in `th` and promotes it so that it occurs as the outermost antecedent of the theorem. The theorem argument `th` is first normalised by a call to `MP_CANON`.

Any theorem whose top level operator (after universal quantifiers are stripped away) is an implication can be viewed as being of the form

```
∀v1 .. vn. p1 /\ p2 /\ .. pn ==> c
```

where the variables `v1` to `vn` may be free in some of the antecedents and/or conclusion `c`. To promote a premise `pi` transforms the above into

```
∀va .. vk. pi ==> ∀vx .. vz. pa /\ pb ... /\ pj ==> c
```

The four constructors of the `match_position` type can be used to designate different premises. The `Pos f` form applies the function `f` to the list of premises, and is expected to return a member of the given list. The `Pat q` form finds the first premise that matches the given quotation pattern. In this context, `Any` is viewed as a synonym for `Pos hd`. Finally, the `Concl` form selects the conclusion of the theorem, and "promotes" it by taking the contrapositive of the theorem.

After promotion some cleanup is performed. If a contrapositive was taken, double negations in the promoted premise are removed, and in all cases, universal quantifiers of variables not present in the promoted premise are pushed down to govern other premises.

Failure

Fails if the provided match position does not denote a premise present in the given theorem.

Example

```
> sortingTheory.ALL_DISTINCT_PERM;
val it = ⊢ ∀l1 l2. PERM l1 l2 ⇒ (ALL_DISTINCT l1 ⇔ ALL_DISTINCT l2): thm

> it |> iffLR |> pp (Pos last);
val it = ⊢ ∀l1. ALL_DISTINCT l1 ⇒ ∀l2. PERM l1 l2 ⇒ ALL_DISTINCT l2: thm
```

See also

`Drule.MP_CANON`, `Tactic.mp_then`

pp_tag (Tag)

```
Tag.pp_tag : tag Parse.pprinter
```

Prettyprinter for tags.

An invocation `pp_tag t` will produce a pretty representation for tag `t`. Such a pretty-printer can be used to produce outputs, or return strings, or to combine with other pretty representations to create compound values.

Failure

Never fails.

Example

```
> show_tags := true;
val it = (): unit

> Portable.pprint Tag.pp_tag (Tag.read "fooble");
[oracles: fooble] [axioms: ]
val it = (): unit
```

pp_term_without_overloads_on (Parse)

```
Parse.pp_term_without_overloads_on : string list -> term PP.pprinter
```

Printing function for terms without using overload mappings of certain tokens.

The call `pp_term_without_overloads_on ls` returns a printing function to print terms without using any overload mappings of the tokens in `ls`, using the system's standard pretty-printing stream type.

Example

```
> val termpp = pp_term_without_overloads_on ["+"];
val termpp = fn: term Parse.pprinter
> val _ = Portable.pprint termpp ``x + y`` ;
arithmetic$+ x y
val it = (): unit
```

Failure

Should never fail.

See also

`Parse.pp_term_without_overloads`, `Parse.print_term_without_overloads_on`, `Parse.term_without_overloads_on_to_string`, `Parse.print_from_grammars`

pprint (Portable)

```
Portable.pprint : 'a PP.pprinter -> 'a -> unit
```

Pretty-prints a value to output

A call to `pprint ppf x` will call the pretty-printing function `ppf` on value `x`, with the pretty-printing output printed. string that is eventually returned to the user. The linewidth used for determining when to wrap with newline characters is 72.

Failure

Fails if the pretty-printing function fails on the particular input value.

Example

```
> Portable.pprint PP.add_string "hello";
hello
val it = (): unit
```

See also

`Lib.ppstring`

ppstring (Lib)

```
Lib.ppstring : 'a PP.pprinter -> 'a -> string
```

Pretty-prints a value into a string.

A call to `ppstring ppf x` will call the pretty-printing function `ppf` on value `x`, with the pretty-printing output stored in the string that is eventually returned to the user. The linewidth used for determining when to wrap with newline characters is given by the reference `Globals.linewidth` (typically 72).

Failure

Fails if the pretty-printing function fails on the particular input value.

Example

```
> ppstring PP.add_string "hello"
val it = "hello": string
```

Comments

The returned string may contain unwanted terminal-specific escape codes, see `rawterm_pp`.

See also

`Portable.pprint`, `Parse.term_to_string`, `Parse.rawterm_pp`

PRE_CONV (reduceLib)

```
reduceLib.PRE_CONV : conv
```

Calculates by inference the predecessor of a numeral.

If `n` is a numeral (e.g. `0`, `1`, `2`, `3`,...), then `PRE_CONV "PRE n"` returns the theorem:

```
|- PRE n = s
```

where `s` is the numeral that denotes the predecessor of the natural number denoted by `n`.

Failure

`PRE_CONV tm` fails unless `tm` is of the form ``` ``PRE n`` ```, where `n` is a numeral.

Example

```
> PRE_CONV ``PRE 0``;
val it = |- PRE 0 = 0 : thm

> PRE_CONV ``PRE 1``;
val it = |- PRE 1 = 0 : thm

> PRE_CONV ``PRE 22``;
val it = |- PRE 22 = 21 : thm
```

PRED_ASSUM (Tactical)

```
Tactical.PRED_ASSUM : (term -> bool) -> thm_tactic -> tactic
```

Discharges a selected assumption and passes it to a theorem-tactic.

`PRED_ASSUM` finds the first assumption satisfying the prediate given, removes it from the assumption list, `ASSUME`s it, passes it to the theorem-tactic and then applies the consequent tactic. Thus, where `t` is the first assumption satisfying `p`,

```
PRED_ASSUM p f ([a1,... ai, t, aj, ... an], goal) =
  f (ASSUME t) ([a1,... ai, aj,... an], goal)
```

For example (again, where `t` is the first assumption in `A u {t}` satisfying `p`), if

```
 A ?- c
========  f (ASSUME t)
 B ?- v
```

then

```
 A u {t} ?- c
===============  PRED_ASSUM p f
    B ?- v
```

Failure

`PRED_ASSUM p` will fail on goals where no assumption safisfies `p`.

See also

`Thm_cont.UNDISCH_THEN`, `Tactical.PAT_ASSUM`, `Tactical.POP_ASSUM`, `Tactic.UNDISCH_TAC`

prefer_form_with_tok (Parse)

```
Parse.prefer_form_with_tok : {term_name : string, tok : string} -> unit
```

Sets a grammar rule's preferred flag, causing it to be preferentially printed.

A call to `prefer_form_with_tok` causes the parsing/pretty-printing rule specified by the `term_name-tok` combination to be the preferred rule for pretty-printing purposes. This change affects the global grammar.

Failure

Never fails.

Example

Imagine that one wants to use an infix "U" to stand for the "UNION" term. This could be done as follows:

```
> set_mapped_fixity {term_name = "UNION", fixity = Infixl 500,
                     tok = "U"};
val it = () : unit

> ``s U t``;
val it = ``s U t`` : term

> dest_term it;
val it = COMB(``$UNION s``, ``t``) : lambda
```

Having made this change, one might prefer to see the form with UNION printed:

```
> prefer_form_with_tok {term_name = "UNION", tok = "UNION"};
val it = () : unit

> ``s U t``;
val it = ``s UNION t`` : term
```

Comments

As the example above demonstrates, using this function does not affect the parser at all.

There is a companion temp_prefer_form_with_tok function, which has the same effect on the global grammar, but which does not cause this effect to persist when the current theory is exported.

prefer_int (intLib)

```
intLib.prefer_int : unit -> unit
```

Makes the parser favour integer possibilities in ambiguous terms.

Calling `prefer_int()` causes the global grammar to be altered so that the standard arithmetic operator symbols (+, ∗, etc.), as well as numerals, are given integral types if possible. This effect is brought about through the application of multiple calls to `temp_overload_on`, so that the "arithmetic symbols" need not have been previously mapping to integral possibilities at all (as would be the situation after a call to `deprecate_int`).

Failure

Never fails.

See also

`intLib.deprecate_int`, `Parse.overload_on`

PRENEX_CONV (Arith)

`Arith.PRENEX_CONV : conv`

Puts a formula into prenex normal form.

This function puts a formula into prenex normal form, and in the process splits any Boolean equalities (if-and-only-if) into two implications. If there is a Boolean-valued subterm present as the condition of a conditional, the subterm will be put in prenex normal form, but quantifiers will not be moved out of the condition. Some renaming of variables may take place.

Failure

Never fails.

Example

```
> Arith.PRENEX_CONV ``!m n. (m <= n) ==> !p. (m < SUC(n + p))``;
val it =
   ⊢ (∀m n. m ≤ n ⇒ ∀p. m < SUC (n + p)) ⇔ ∀m n p. m ≤ n ⇒ m < SUC (n + p):
```

```
   thm

> Arith.PRENEX_CONV ``!p. (!m. m >= p) = (p = 0)``;
val it =
   ⊢ (∀p. (∀m. m ≥ p) ⇔ p = 0) ⇔
     ∀p m. ∃m'. (m' ≥ p ⇒ p = 0) ∧ (p = 0 ⇒ m ≥ p): thm
```

Useful as a preprocessor to decision procedures which require their argument formula to be in prenex normal form.

See also

`Arith.is_prenex`

prim_irule (Tactic)

```
Tactic.prim_irule : thm_tactic
```

For a goal which is an instance of the conclusion of the supplied theorem, replace the goal by the instantiated hypotheses of the supplied theorem.

When given a theorem `th = A' |- t` and a goal `A ?- t'` where `t` can be matched to `t'` by instantiating free variables, including appropriate type instantiation, `prim_irule` replaces the goal by new subgoals which are the hypotheses `A'`, instantiated

The order of the new subgoals corresponds to the order in which `hyp th` returns the hypotheses `A'`

Failure

Fails unless the theorem has a conclusion which is instantiable to match that of the goal.

Comments

`irule` also pre-processes the supplied theorem, which will normally be useful

`prim_irule` differs from `MATCH_ACCEPT_TAC` in that hypotheses of the supplied theorem may also be substituted, and will appear as new subgoals

See also

Tactic.irule, Tactic.MATCH_ACCEPT_TAC, Tactic.ACCEPT_TAC

prim_mk_const (Term)

```
Term.prim_mk_const : {Thy:string, Name:string} -> term
```

Build a constant.

If Name is the name of a previously declared constant in theory Thy, then prim_mk_const {Thy,Name} will return the specified constant.

Failure

If Name is not the name of a constant declared in theory Thy.

Example

```
> prim_mk_const {Thy="min", Name="="};
val it = “$=”: term

> type_of it;
val it = “:α -> α -> bool”: hol_type
```

Comments

The difference between mk_thy_const (and mk_const) and prim_mk_const is that mk_thy_const and mk_const will create type instances of polymorphic constants, while prim_mk_const merely returns the originally declared constant.

See also

Term.mk_thy_const

prim_variant (Term)

```
Term.prim_variant : term list -> term -> term
```

Rename a variable to be different from any in a list.

The function `prim_variant` is exactly the same as `variant`, except that it doesn't rename away from constants.

Failure

`prim_variant l t` fails if any term in the list `l` is not a variable or if `t` is not a variable.

Example

```
> variant [] (mk_var("T",bool));
val it = “T'”: term

> prim_variant [] (mk_var("T",bool));
val it = “T”: term
```

Comments

The extra amount of renaming that `variant` does is useful when generating new constant names (even though it returns a variable) inside high-level definition mechanisms. Otherwise, `prim_variant` seems preferable.

See also

`Term.variant`, `Term.mk_var`, `Term.genvar`, `Term.mk_primed_var`

prime (Lib)

```
Lib.prime : string -> string
```

Attach a prime mark to a string.

A call `prime s` is equal to `s ^ "'"`.

Failure

Never fails.

See also

`Term.variant`

print_datatypes (EmitTeX)

```
EmitTeX.print_datatypes : string -> unit
```

Prints datatype declarations for the named theory to the screeen (standard out).

An invocation of `print_datatypes thy`, where `thy` is the name of a currently loaded theory segment, will print the datatype declarations made in that theory.

Failure

Never fails. If `thy` is not the name of a currently loaded theory segment then no output will be produced.

Example

```
> new_theory "example";
<<HOL message: Restarting theory "example">>
<<HOL warning: Theory.callhooks: Hook Parse.clear_consts_from_grammar failed on eve
Exception raised at Term.dest_thy_const: not a const
>>
val it = (): unit
> val _ = Hol_datatype `example = First | Second`;
<<HOL message: Defined type: "example">>
```

```
> EmitTeX.print_datatypes "example";
example = First | Second
val it = (): unit
```

See also

`EmitTeX.datatype_thm_to_string`, `bossLib.Hol_datatype`

print_from_grammars (Parse)

```
Parse.print_from_grammars :
  (type_grammar.grammar * term_grammar.grammar) ->
  (hol_type Parse.pprinter * term Parse.pprinter)
```

Returns printing functions based on the supplied grammars.

When given a pair consisting of a type and term grammar (such a pair is exported with every theory, under the name `<thy>_grammars`), this function returns printing functions that use those grammars to render terms and types using the system's standard pretty-printing stream type.

Failure

Never fails.

Example

With `arithmeticTheory` loaded, arithmetic expressions and numerals print pleasingly:

```
- load "arithmeticTheory";
> val it = () : unit

- ``3 + x * 4``;
> val it = ``3 + x * 4`` : term
```

The printing of these terms is controlled by the global grammar, which is augmented when the theory of arithmetic is loaded. Printing functions based on the grammar of the base theory `bool` can be defined:

```
> val (typepp, termpp) = print_from_grammars bool_grammars;
val termpp = fn : term Parse.pprinter
val typepp = fn : hol_type Parse.pprinter
```

These functions can then be used to print arithmetic terms (note that neither the global parser nor printer are disturbed by this activity), using the `Portable.pprint` function (or `Lib.ppstring`, which returns a string):

```
> Portable.pprint termpp ``3 + x * 4``;
arithmetic$+
  (arithmetic$NUMERAL
     (arithmetic$BIT1 (arithmetic$BIT1 arithmetic$ZERO)))
  (arithmetic$* x
     (arithmetic$NUMERAL
        (arithmetic$BIT2 (arithmetic$BIT1 arithmetic$ZERO))))
> val it = () : unit
```

Not only have the fixities of + and ∗ been ignored, but the constants in the term, belonging to `arithmeticTheory`, are all printed in "long identifier" form because the grammars from `boolTheory` don't know about them.

Printing terms with early grammars such as `bool_grammars` can remove layers of potentially confusing pretty-printing, including complicated concrete syntax and overloading, and even the underlying representation of numerals.

See also

`Parse.parse_from_grammars`, `Parse.print_term_by_grammar`, `Parse.Term`, `Portable.pprint`, `Lib.ppstring`

print_tac (goalStack)

```
goalStack.print_tac : string -> tactic
```

Prints the goal; does not change it.

The tactic `print_tac s` prints the string `s` followed by (the pretty-printed string of) the goal to which it is applied (using the standard SML `print` function). The effect on

the goal is as if the tactic `ALL_TAC` had been applied (i.e., the state of the goal is not changed).

Failure

Never fails.

Comments

This is useful for debugging tactic applications in contexts where the usual interactive goal-stack is not available.

See also

`goalStack.note_tac`, `Tactical.ALL_TAC`.

print_term (Parse)

```
Parse.print_term : term -> unit
```

Prints a term to the screen (standard out).

The function `print_term` prints a term to the screen. It first converts the term into a string, and then outputs that string to the standard output stream.

The conversion to the string is done by `term_to_string`. The term is printed using the pretty-printing information contained in the global grammar.

Failure

Should never fail.

See also

`Parse.term_to_string`

print_term_as_tex (EmitTeX)

```
EmitTeX.print_term_as_tex : term -> unit
```

Prints a term as LaTeX.

An invocation of `print_term_as_tex tm` will print the term `tm`, replacing various character patterns (e.g. /\ and \/) with LaTeX commands. The translation is controlled by the string to string function `EmitTeX.hol_to_tex`.

Failure

Should never fail.

Example

```
- EmitTeX.print_term_as_tex ``\l h. {x | l <= x /\ x <= h}`` before print "\n";
\HOLTokenLambda{}l h. \HOLTokenLeftbrace{}x | l \HOLTokenLeq{} x \HOLTokenConj{} x
> val it = () : unit
```

Comments

The LaTeX style file `holtexbasic.sty` (or `holtex.sty`) should be used and the output should be pasted into a Verbatim environment.

See also

`EmitTeX.print_type_as_tex`, `EmitTeX.print_theorem_as_tex`, `EmitTeX.print_theory_as_tex`, `EmitTeX.print_theories_as_tex_doc`, `EmitTeX.tex_theory`

print_term_by_grammar (Parse)

```
Parse.print_term_by_grammar :
  (type_grammar.grammar * term_grammar.grammar) -> term -> unit
```

Prints a term to standard out, using grammars to specify how.

Where `print_term` uses the (implicit) global grammars to control the printing of its term argument, the `print_term_by_grammar` uses user-supplied grammars. These can control the printing of concrete syntax (operator fixities and precedency) and the degree of constant overloading.

Failure

Never fails.

See also

`Parse.print_from_grammars`

print_theorem_as_tex (EmitTeX)

```
EmitTeX.print_theorem_as_tex : thm -> unit
```

Prints a theorem as LaTeX.

An invocation of `print_theorem_as_tex thm` will print the term `thm`, replacing various character patterns (e.g. /\ and \/) with LaTeX commands. The translation is controlled by the string to string function `EmitTeX.hol_to_tex`. If the theorem is for a datatype then the function `datatype_thm_to_string` is used to produce the orginal declaration.

Failure

Should never fail.

Example

```
- EmitTeX.print_theorem_as_tex listTheory.CONS before print "\n";
\HOLTokenTurnstile{} \HOLTokenForall{}l. \HOLTokenNeg{}NULL l \HOLTokenImp{} (HD l::TL
> val it = () : unit
- EmitTeX.print_theorem_as_tex listTheory.datatype_list before print "\n";
list = [] | CONS of \HOLTokenQuote{}a \HOLTokenImp{} \HOLTokenQuote{}a list
> val it = () : unit
```

Comments

The LaTeX style file `holtexbasic.sty` (or `holtex.sty`) should be used and the output should be pasted into a Verbatim environment.

See also

`EmitTeX.print_term_as_tex`, `EmitTeX.print_type_as_tex`, `EmitTeX.print_theory_as_tex`, `EmitTeX.print_theories_as_tex_doc`, `EmitTeX.tex_theory`

print_theories_as_tex_doc (EmitTeX)

```
EmitTeX.print_theories_as_tex_doc : string list -> string -> unit
```

Emits theories as LaTeX commands and creates a document template.

An invocation of `print_theories_as_tex_doc thys name` will export the named theories `thys` as a collection of LaTeX commands and it will also generate a document, whose file name is given by `name`, that presents all of the theories. The theories are exported with `print_theory_as_tex`.

Failure

Will fail if there is a system error when trying to write the files. It will not overwite the file `name`, however, the theories may be overwritten.

Example

The invocation

```
- EmitTeX.print_theories_as_tex_doc ["arithmetic", "list", "words"] "report";
> val it = () : unit
```

will generate four files “HOLarithmetic.tex”, “HOLlist.tex”, “HOLwords.tex” and “report.tex”.

The document can be built as follows:

```
$ cp $HOLHOME/src/emit/holtex.sty .
$ pdflatex report
$ makeindex report
$ pdflatex report
```

See also

`EmitTeX.print_term_as_tex`, `EmitTeX.print_type_as_tex`, `EmitTeX.print_theorem_as_tex`, `EmitTeX.print_theory_as_tex`, `EmitTeX.tex_theory`

print_theory (Hol_pp)

```
Hol_pp.print_theory : string -> unit
```

Print a theory on the standard output.

An invocation `print_theory s` will display the contents of the theory segment `s` on the standard output. The string `"-"` may be used to denote the current theory segment.

Failure

If `s` is not the name of a loaded theory.

Example

```
> print_theory "combin";
Theory: combin

Parents:
    marker

Term constants:
    :>              :β -> (β -> α) -> α
    ASSOC           :(α -> α -> α) -> bool
    C               :(α -> β -> γ) -> β -> α -> γ
    COMM            :(α -> α -> β) -> bool
```

```
    EXTENSIONAL   :(α -> bool) -> (α -> β) -> bool
    FAIL          :α -> β -> α
    FCOMM         :(α -> β -> α) -> (γ -> α -> α) -> bool
    I             :α -> α
    K             :α -> β -> α
    LEFT_ID       :(α -> β -> β) -> α -> bool
    MONOID        :(α -> α -> α) -> α -> bool
    RESTRICTION   :(α -> bool) -> (α -> β) -> α -> β
    RIGHT_ID      :(α -> β -> α) -> β -> bool
    S             :(α -> β -> γ) -> (α -> β) -> α -> γ
    UPDATE        :α -> β -> (α -> β) -> α -> β
    W             :(α -> α -> β) -> α -> β
    o             :(γ -> β) -> (α -> γ) -> α -> β

Definitions:
    APP_DEF
      ⊢ ∀x f. (x :> f) = f x
    ASSOC_DEF
      ⊢ ∀f. ASSOC f ⇔ ∀x y z. f x (f y z) = f (f x y) z
    COMM_DEF
      ⊢ ∀f. COMM f ⇔ ∀x y. f x y = f y x
    C_DEF
      ⊢ flip = (λf x y. f y x)
    EXTENSIONAL_def
      ⊢ ∀s f. EXTENSIONAL s f ⇔ ∀x. x ∉ s ⇒ f x = ARB
    FAIL_DEF
      ⊢ FAIL = (λx y. x)
    FCOMM_DEF
      ⊢ ∀f g. FCOMM f g ⇔ ∀x y z. g x (f y z) = f (g x y) z
    I_DEF
      ⊢ I = S K K
    K_DEF
      ⊢ K = (λx y. x)
    LEFT_ID_DEF
      ⊢ ∀f e. LEFT_ID f e ⇔ ∀x. f e x = x
    MONOID_DEF
      ⊢ ∀f e. MONOID f e ⇔ ASSOC f ∧ RIGHT_ID f e ∧ LEFT_ID f e
    RESTRICTION
      ⊢ ∀s f x. RESTRICTION s f x = if x ∈ s then f x else ARB
    RIGHT_ID_DEF
```

```
[...Output elided...]
```

See also

`DB.dest_theory`, `DB.thy`

print_theory_as_tex (EmitTeX)

```
EmitTeX.print_theory_as_tex : string -> unit
```

Emits a theory as LaTeX commands.

An invocation of `print_theory_as_tex thy` will export the named theory as a collection of LaTeX commands. The output file is named "HOLthy.tex", where `thy` is the named theory. The prefix "HOL" can be changed by setting `holPrefix`. The file is stored in the directory `emitTeXDir`. By default the current working directory is used.

The LaTeX file will contain commands for displaying the theory's datatypes, definitions and theorems.

Failure

Will fail if there is a system error when trying to write the file. If the theory is not loaded then a message will be printed and an empty file will be created.

Example

The list theory is exported with:

```
- EmitTeX.print_theory_as_tex "list";
> val it = () : unit
```

The resulting file can be included in a LaTeX document with

```
\input{HOLlist}
```

Some examples of the available LaTeX commands are listed below:

```
\HOLlistDatatypeslist
\HOLlistDefinitionsALLXXDISTINCT
\HOLlistTheoremsALLXXDISTINCTXXFILTER
```

Underscores in HOL names are replaced by "XX"; quotes become "YY" and numerals are expanded out e.g. "1" becomes "One".

Complete listings of the datatypes, definitions and theorems are displayed with:

```
\HOLlistDatatypes
\HOLlistDefinitions
\HOLlistTheorems
```

The date the theory was build can be displayed with:

```
\HOLlistDate
```

The generated LaTeX will reflect the output of `Parse.thm_to_string`, which is under the control of the user. For example, the line width can be changed by setting `Globals.linewidth`.

The Verbatim display environment is used, however, "boxed" versions can be constructed. For example,

```
\BUseVerbatim{HOLlistDatatypeslist}
```

can be used inside tables and figures.

Comments

The LaTeX style file `holtexbasic.sty` (or `holtex.sty`) should be used. These style files can be modified by the user. For example, the font can be changed to Helvetica with

```
\fvset{fontfamily=helvetica}
```

However, note that this will adversely effect the alignment of the output.

See also

`EmitTeX.print_term_as_tex`, `EmitTeX.print_type_as_tex`, `EmitTeX.print_theorem_as_tex`, `EmitTeX.print_theories_as_tex_doc`, `EmitTeX.tex_theory`

print_type_as_tex (EmitTeX)

```
EmitTeX.print_type_as_tex : hol_type -> unit
```

Prints a type as LaTeX.

An invocation of `print_type_as_tex ty` will print the type `ty`, replacing various character patterns (e.g. # and ->) with LaTeX commands. The translation is controlled by the string to string function `EmitTeX.hol_to_tex`.

Failure

Should never fail.

Example

```
- EmitTeX.print_type_as_tex ``:bool # bool -> num`` before print "\n";
:bool \HOLTokenProd{} bool \HOLTokenMap{} num
> val it = () : unit
```

Comments

The LaTeX style file `holtexbasic.sty` (or `holtex.sty`) should be used and the output should be pasted into a Verbatim environment.

See also

`EmitTeX.print_term_as_tex`, `EmitTeX.print_theorem_as_tex`, `EmitTeX.print_theory_as_tex`, `EmitTeX.print_theories_as_tex_doc`, `EmitTeX.tex_theory`

print_without_macros (Parse)

```
Parse.print_without_macros : term -> unit
```

Prints a term to standard output, using the current grammars but without non-trivial overloading

Where `print_term` uses the (implicit) global grammars to control the printing of its term argument, the `print_without_macros` uses these grammars, modified to remove non-trivial overloading. (Each constant is overloaded with itself, which avoids the printing of the theory name for every constant).

Failure

Never fails.

Sometimes one wants to see how a term is built up, where the pretty-printing simplifies it to the point where this is not clear.

For example:

```
``MEM`` ;
val it = ``\x l. MEM x l`` ;
print_without_macros ``MEM`` ;
 \x l. x IN LIST_TO_SET l

concl ratTheory.RATND_RAT_OF_NUM ;
val it = (RATN (&n) = &n) /\ (RATD (&n) = 1): term
Parse.print_without_macros (concl ratTheory.RATND_RAT_OF_NUM) ;
(RATN (rat_of_num n) = int_of_num n) /\ (RATD (rat_of_num n) = 1n)
```

Comments

To change the (implicit) global grammars to remove overloading, see `clear_overloads`

See also

`Parse.print_term_by_grammar`, `term_grammar.clear_overloads`

PROVE (BasicProvers)

```
BasicProvers.PROVE : thm list -> term -> thm
```

Re-exported from `bossLib.PROVE`. See that entry for full documentation.

prove (Tactical)

```
Tactical.prove : term * tactic -> thm
```

Attempts to prove a boolean term using the supplied tactic.

When applied to a term-tactic pair `(tm,tac)`, the function `prove` attempts to prove the goal `?- tm`, that is, the term `tm` with no assumptions, using the tactic `tac`. If `prove` succeeds, it returns the corresponding theorem `A |- tm`, where the assumption list `A` may not be empty if the tactic is invalid; `prove` has no inbuilt validity-checking.

By default this function calls `TAC_PROOF` (*q.v.*) with a wrapper that augments that underlying function's error-reporting, but this call is made via a user-updatable reference. This means there are contexts in which this function may, for example, automatically assert all terms it is passed as theorems using `mk_thm`.

Comments

To emulate interactive use and its variations, `prove` is appropriate; to get fixed behaviour, use `TAC_PROOF`.

Failure

Fails if the term is not of type `bool` (and so cannot possibly be the conclusion of a theorem), or if the tactic does not solve the goal.

See also

`Tactical.prove_goal`, `Tactical.set_prover`, `Tactical.TAC_PROOF`.

PROVE (bossLib)

```
bossLib.PROVE : thm list -> term -> thm
```

Also exported as `BasicProvers.PROVE`.

Prove a theorem with use of supplied lemmas.

An invocation `PROVE thl M` attempts to prove `M` using an automated reasoner supplied with the lemmas in `thl`. The automated reasoner performs a first order proof search. It currently provides some support for polymorphism and higher-order values (lambda terms).

Failure

If the proof search fails, or if `M` does not have type `bool`.

Example

```
> PROVE []  (concl SKOLEM_THM);
Meson search level: ........
val it = ⊢ ∀P. (∀x. ∃y. P x y) ⇔ ∃f. ∀x. P x (f x): thm

> let open arithmeticTheory
  in
   PROVE [ADD_ASSOC, ADD_SYM, ADD_CLAUSES]
     (Term `x + 0 + y + z = y + (z + x)`)
  end;
Meson search level: ............
val it = ⊢ x + 0 + y + z = y + (z + x): thm
```

Comments

Some output (a row of dots) is currently generated as `PROVE` works. If the frequency of dot emission becomes slow, that is a sign that the term is not likely to be proved with the current lemmas.

Unlike `MESON_TAC`, `PROVE` can handle terms with conditionals.

See also

`bossLib.PROVE_TAC`, `mesonLib.MESON_TAC`, `mesonLib.ASM_MESON_TAC`

prove_abs_fn_one_one (Drule)

```
Drule.prove_abs_fn_one_one : thm -> thm
```

Proves that a type abstraction function is one-to-one (injective).

If `th` is a theorem of the form returned by the function `define_new_type_bijections`:

```
|- (!a. abs(rep a) = a) /\ (!r. P r = (rep(abs r) = r))
```

then `prove_abs_fn_one_one th` proves from this theorem that the function `abs` is one-to-one for values that satisfy `P`, returning the theorem:

```
|- !r r'. P r ==> P r' ==> ((abs r = abs r') = (r = r'))
```

Failure

Fails if applied to a theorem not of the form shown above.

See also

`Definition.new_type_definition`, `Drule.define_new_type_bijections`, `Drule.prove_abs_fn_onto`, `Drule.prove_rep_fn_one_one`, `Drule.prove_rep_fn_onto`

prove_abs_fn_onto (Drule)

```
Drule.prove_abs_fn_onto : thm -> thm
```

Proves that a type abstraction function is onto (surjective).

If `th` is a theorem of the form returned by the function `define_new_type_bijections`:

```
|- (!a. abs(rep a) = a) /\ (!r. P r = (rep(abs r) = r))
```

then `prove_abs_fn_onto th` proves from this theorem that the function `abs` is onto, returning the theorem:

```
|- !a. ?r. (a = abs r) /\ P r
```

Failure

Fails if applied to a theorem not of the form shown above.

See also

`Definition.new_type_definition`, `Drule.define_new_type_bijections`, `Drule.prove_abs_fn_one_one`, `Drule.prove_rep_fn_one_one`, `Drule.prove_rep_fn_onto`

prove_case_elim_thm (Prim_rec)

```
Prim_rec.prove_case_elim_thm : {case_def : thm, nchotomy : thm} -> thm
```

Proves a theorem that eliminates applications of case constants with boolean type.

If `case_def` is the definition of a data type's case constant, where each clause is of the form

```
!a1 .. ai f1 .. fm. type_CASE (ctor_i a1 .. ai) f1 .. fm = f_i a1 .. ai
```

and if `nchotomy` is a theorem describing how a data type's values are classified by constructor, of the form

```
!v. (?a1 .. ai. v = ctor_1 a1 .. ai) \/
    (?b1 .. bj. v = ctor_2 b1 .. bj) \/
    ...
```

then a call to `prove_case_elim_thm{case_def = case_def, nchotomy = nchotomy}` will return a theorem of the form

```
type_CASE v f1 .. fm <=>
  (?a1 .. ai. v = ctor_1 a1 .. ai /\ f1 a1 .. ai) \/
  (?b1 .. bj. v = ctor_2 b1 .. bj /\ f2 b1 .. bj) \/
  ...
```

Used as a rewrite theorem, this result will "eliminate" occurrences of the case-constant from a term, when the case-constant term has boolean type.

Failure

Will fail if the provided theorems are not of the required form. The theorems stored in the `TypeBase` are of the correct form. The theorem returned by `Prim_rec.prove_cases_thm` is of the correct form for the `nchotomy` argument to this function.

Example

```
> prove_case_elim_thm {case_def = TypeBase.case_def_of ``:num``,
                       nchotomy = TypeBase.nchotomy_of ``:num``};
val it = ⊢ num_CASE x v f ⇔ x = 0 ∧ v ∨ ∃n. x = SUC n ∧ f n: thm

> prove_case_elim_thm {case_def = TypeBase.case_def_of ``:bool``,
                       nchotomy = TypeBase.nchotomy_of ``:bool``};
val it = ⊢ (if x then t1 else t2) ⇔ (x ⇔ T) ∧ t1 ∨ (x ⇔ F) ∧ t2: thm
```

See also

`Prim_rec.prove_cases_thm`, `Prim_rec.prove_case_rand_thm`

prove_case_eq_thm (Prim_rec)

```
Prim_rec.prove_case_eq_thm : {case_def : thm, nchotomy : thm} -> thm
```

Proves a rewrite for eliminating certain forms of case expression.

If `case_def` is the definition of a data type's case constant, where each clause is of the form

```
!a1 .. ai f1 .. fm. type_CASE (ctor_i a1 .. ai) f1 .. fm = f_i a1 .. ai
```

and if `nchotomy` is a theorem describing how a data type's values are classified by constructor, of the form

```
!v. (?a1 .. ai. v = ctor_1 a1 .. ai) \/
    (?b1 .. bj. v = ctor_2 b1 .. bj) \/
    ...
```

then a call to `prove_case_elim_thm{case_def = case_def, nchotomy = nchotomy}` will return a theorem of the form

```
(type_CASE u f1 .. fm = v) <=>
  (?a1 .. ai. u = ctor_1 a1 .. ai /\ f1 a1 .. ai = v) \/
  (?b1 .. bj. u = ctor_2 b1 .. bj /\ f2 b1 .. bj = v) \/
  ...
```

Failure

Will fail if the provided theorems are not of the required form. The theorems stored in the `TypeBase` are of the correct form. The theorem returned by `Prim_rec.prove_cases_thm` is of the correct form for the `nchotomy` argument to this function.

See also

`Prim_rec.prove_case_elim_thm`, `Prim_rec.prove_case_rand_thm`

prove_case_rand_thm (Prim_rec)

```
Prim_rec.prove_case_rand_thm : {case_def : thm, nchotomy : thm} -> thm
```

Proves a theorem that "lifts" applied case constants up within a term.

If `case_def` is the definition of a data type's case constant, where each clause is of the form

```
!a1 .. ai f1 .. fm. type_CASE (ctor_i a1 .. ai) f1 .. fm = f_i a1 .. ai
```

and if `nchotomy` is a theorem describing how a data type's values are classified by constructor, of the form

```
!v. (?a1 .. ai. v = ctor_1 a1 .. ai) \/
    (?b1 .. bj. v = ctor_2 b1 .. bj) \/
    ...
```

then a call to `prove_case_rand_thm{case_def = case_def, nchotomy = nchotomy}` will return a theorem of the form

```
f (type_CASE x f1 .. fm) =
  type_CASE x (\a1 .. ai. f (f1 a1 .. ai))
              (\b1 .. bj. f (f2 b1 .. bj))
              ...
```

Given the typical pretty-printing provided for case-terms, the right-hand side of the above theorem will actually print as

```
case x of
   ctor_1 a1 .. ai => f (f1 a1 .. ai)
 | ctor_2 b1 .. bj => f (f2 b1 .. bj)
 | ...
```

Failure

Will fail if the provided theorems are not of the required form. The theorems stored in the `TypeBase` are of the correct form. The theorem returned by `Prim_rec.prove_cases_thm` is of the correct form for the `nchotomy` argument to this function.

Example

```
> prove_case_rand_thm {case_def = TypeBase.case_def_of ``:num``,
                       nchotomy = TypeBase.nchotomy_of ``:num``};
val it = ⊢ f' (num_CASE x v f) = case x of 0 => f' v | SUC n => f' (f n): thm
```

See also

`Prim_rec.prove_cases_thm`

prove_cases_thm (Prim_rec)

```
Prim_rec.prove_cases_thm : (thm -> thm)
```

Proves a structural cases theorem for an automatically-defined concrete type.

`prove_cases_thm` takes as its argument a structural induction theorem, in the form returned by `prove_induction_thm` for an automatically-defined concrete type. When applied to such a theorem, `prove_cases_thm` automatically proves and returns a theorem which states that every value the concrete type in question is denoted by the value returned by some constructor of the type.

Failure

Fails if the argument is not a theorem of the form returned by `prove_induction_thm`

Example

Given the following structural induction theorem for labelled binary trees:

```
|- !P. (!x. P(LEAF x)) /\ (!b1 b2. P b1 /\ P b2 ==> P(NODE b1 b2)) ==>
       (!b. P b)
```

`prove_cases_thm` proves and returns the theorem:

```
|- !b. (?x. b = LEAF x) \/ (?b1 b2. b = NODE b1 b2)
```

This states that every labelled binary tree `b` is either a leaf node with a label `x` or a tree with two subtrees `b1` and `b2`.

See also

`Prim_rec.INDUCT_THEN`, `Prim_rec.new_recursive_definition`, `Prim_rec.prove_constructors_distinct`, `Prim_rec.prove_constructors_one_one`, `Prim_rec.prove_induction_thm`, `Prim_rec.prove_rec_fn_exists`

prove_constructors_distinct (Prim_rec)

```
Prim_rec.prove_constructors_distinct : (thm -> thm)
```

Proves that the constructors of an automatically-defined concrete type yield distinct values.

`prove_constructors_distinct` takes as its argument a primitive recursion theorem, in the form returned by `define_type` for an automatically-defined concrete type. When applied to such a theorem, `prove_constructors_distinct` automatically proves and returns a theorem which states that distinct constructors of the concrete type in question yield distinct values of this type.

Failure

Fails if the argument is not a theorem of the form returned by `define_type`, or if the concrete type in question has only one constructor.

Example

Given the following primitive recursion theorem for labelled binary trees:

```
|- !f0 f1.
     ?! fn.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

`prove_constructors_distinct` proves and returns the theorem:

```
|- !x b1 b2. ~(LEAF x = NODE b1 b2)
```

This states that leaf nodes are different from internal nodes. When the concrete type in question has more than two constructors, the resulting theorem is just conjunction of inequalities of this kind.

See also

`Prim_rec.INDUCT_THEN`, `Prim_rec.new_recursive_definition`, `Prim_rec.prove_cases_thm`, `Prim_rec.prove_constructors_one_one`, `Prim_rec.prove_induction_thm`, `Prim_rec.prove_rec_fn_exists`

prove_constructors_one_one (Prim_rec)

```
Prim_rec.prove_constructors_one_one : (thm -> thm)
```

Proves that the constructors of an automatically-defined concrete type are injective.

`prove_constructors_one_one` takes as its argument a primitive recursion theorem, in the form returned by `define_type` for an automatically-defined concrete type. When applied to such a theorem, `prove_constructors_one_one` automatically proves and returns a theorem which states that the constructors of the concrete type in question are injective (one-to-one). The resulting theorem covers only those constructors that take arguments (i.e. that are not just constant values).

Failure

Fails if the argument is not a theorem of the form returned by `define_type`, or if all the constructors of the concrete type in question are simply constants of that type.

Example

Given the following primitive recursion theorem for labelled binary trees:

```
|- !f0 f1.
     ?! fn.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

`prove_constructors_one_one` proves and returns the theorem:

```
|- (!x x'. (LEAF x = LEAF x') = (x = x')) /\
   (!b1 b2 b1' b2'.
     (NODE b1 b2 = NODE b1' b2') = (b1 = b1') /\ (b2 = b2'))
```

This states that the constructors `LEAF` and `NODE` are both injective.

See also

`Prim_rec.INDUCT_THEN`, `Prim_rec.new_recursive_definition`, `Prim_rec.prove_cases_thm`, `Prim_rec.prove_constructors_distinct`, `Prim_rec.prove_induction_thm`, `Prim_rec.prove_rec_fn_exists`

prove_goal (Tactical)

```
Tactical.prove_goal : goal * tactic -> thm
```

Proves a goal with a tactic.

By default (using the function `TAC_PROOF` (*q.v.*)), a call to `prove_goal((asl,g), tac)` attempts to prove the specified goal `(asl,g)` by applying the tactic `tac` to it, generating a result `(sgs,vf)`. This result has `sgs` a list of subgoals and `vf` a validation function. If `sgs` is empty, then the value of `vf []` is returned.

If the provided tactic is valid, the resulting theorem will have hypotheses that are a subset of the assumptions `asl`, and will have a conclusion that is alpha-equivalent to `g`.

The default behaviour can be changed with the function `Tactical.set_prover`.

Failure

Fails if the goal `(asl,g)` is not well-typed (all members of the list `asl` must be of type `:bool` as must be the conclusion `g`), or if the supplied tactic does not prove the goal.

Comments

`Tactical.prove_goal` is to `TAC_PROOF` as `Tactical.prove` is to `Tactical.default_prover`.

See also

`Tactical.prove`, `Tactical.set_prover`, `Tactical.TAC_PROOF`.

PROVE_HYP (Drule)

```
Drule.PROVE_HYP : thm -> thm -> thm
```

Eliminates a provable assumption from a theorem.

When applied to two theorems, PROVE_HYP returns a theorem having the conclusion of the second. The new hypotheses are the union of the two hypothesis sets (first deleting, however, the conclusion of the first theorem from the hypotheses of the second).

```
  A1 |- t1     A2 |- t2
------------------------  PROVE_HYP
 A1 u (A2 - {t1}) |- t2
```

Failure

Never fails.

Comments

This is the Cut rule. It is not necessary for the conclusion of the first theorem to be the same as an assumption of the second, but PROVE_HYP is otherwise of doubtful value.

See also

Thm.DISCH, Thm.MP, Drule.UNDISCH

prove_induction_thm (Prim_rec)

```
Prim_rec.prove_induction_thm : (thm -> thm)
```

Derives structural induction for an automatically-defined concrete type.

`prove_induction_thm` takes as its argument a primitive recursion theorem, in the form returned by `define_type` for an automatically-defined concrete type. When applied to such a theorem, `prove_induction_thm` automatically proves and returns a theorem that states a structural induction principle for the concrete type described by the argument theorem. The theorem returned by `prove_induction_thm` is in a form suitable for use with the general structural induction tactic `INDUCT_THEN`.

Failure

Fails if the argument is not a theorem of the form returned by `define_type`.

Example

Given the following primitive recursion theorem for labelled binary trees:

```
|- !f0 f1.
     ?! fn.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

`prove_induction_thm` proves and returns the theorem:

```
|- !P. (!x. P(LEAF x)) /\ (!b1 b2. P b1 /\ P b2 ==> P(NODE b1 b2)) ==>
       (!b. P b)
```

This theorem states the principle of structural induction on labelled binary trees: if a predicate `P` is true of all leaf nodes, and if whenever it is true of two subtrees `b1` and `b2` it is also true of the tree `NODE b1 b2`, then `P` is true of all labelled binary trees.

See also

`Prim_rec.INDUCT_THEN`, `Prim_rec.new_recursive_definition`, `Prim_rec.prove_cases_thm`, `Prim_rec.prove_constructors_distinct`, `Prim_rec.prove_constructors_one_one`, `Prim_rec.prove_rec_fn_exists`

prove_rec_fn_exists (Prim_rec)

```
Prim_rec.prove_rec_fn_exists : thm -> term -> thm
```

Proves the existence of a primitive recursive function over a concrete recursive type.

`prove_rec_fn_exists` is a version of `new_recursive_definition` which proves only that the required function exists; it does not make a constant specification. The first argument is a primitive recursion theorem of the form generated by `Hol_datatype`, and the second is a user-supplied primitive recursive function definition. The theorem which is returned asserts the existence of the recursively-defined function in question (if it is primitive recursive over the type characterized by the theorem given as the first argument). See the entry for `new_recursive_definition` for details.

Failure

As for `new_recursive_definition`.

Example

Given the following primitive recursion theorem for labelled binary trees:

```
|- !f0 f1.
     ?fn.
       (!a. fn (LEAF a) = f0 a) /\
       !a0 a1. fn (NODE a0 a1) = f1 a0 a1 (fn a0) (fn a1) : thm
```

`prove_rec_fn_exists` can be used to prove the existence of primitive recursive functions over binary trees. Suppose the value of `th` is this theorem. Then the existence of a recursive function `Leaves`, which computes the number of leaves in a binary tree, can be proved as shown below:

```
- prove_rec_fn_exists th
   ``(Leaves (LEAF (x:'a)) = 1) /\
     (Leaves (NODE t1 t2) = (Leaves t1) + (Leaves t2))``;
> val it =
    |- ?Leaves.
         (!x. Leaves (LEAF x) = 1) /\
         !t1 t2. Leaves (NODE t1 t2) = Leaves t1 + Leaves t2 : thm
```

The result should be compared with the example given under `new_recursive_definition`.

See also

`bossLib.Hol_datatype`, `Prim_rec.new_recursive_definition`

prove_rep_fn_one_one (Drule)

```
Drule.prove_rep_fn_one_one : thm -> thm
```

Proves that a type representation function is one-to-one (injective).

If `th` is a theorem of the form returned by the function `define_new_type_bijections`:

```
|- (!a. abs(rep a) = a) /\ (!r. P r = (rep(abs r) = r))
```

then `prove_rep_fn_one_one th` proves from this theorem that the function `rep` is one-to-one, returning the theorem:

```
|- !a a'. (rep a = rep a') = (a = a')
```

Failure

Fails if applied to a theorem not of the form shown above.

See also

`Definition.new_type_definition`, `Drule.define_new_type_bijections`, `Drule.prove_abs_fn_one_one`, `Drule.prove_abs_fn_onto`, `Drule.prove_rep_fn_onto`

prove_rep_fn_onto (Drule)

```
Drule.prove_rep_fn_onto : thm -> thm
```

Proves that a type representation function is onto (surjective).

If `th` is a theorem of the form returned by the function `define_new_type_bijections`:

```
|- (!a. abs(rep a) = a) /\ (!r. P r = (rep(abs r) = r))
```

then `prove_rep_fn_onto th` proves from this theorem that the function `rep` is onto the set of values that satisfy `P`, returning the theorem:

```
|- !r. P r = (?a. r = rep a)
```

Failure

Fails if applied to a theorem not of the form shown above.

See also

`Definition.new_type_definition`, `Drule.define_new_type_bijections`, `Drule.prove_abs_fn_one_one`, `Drule.prove_abs_fn_onto`, `Drule.prove_rep_fn_one_one`

PROVE_TAC (BasicProvers)

```
BasicProvers.PROVE_TAC : thm list -> tactic
```

Re-exported from `bossLib.PROVE_TAC`. See that entry for full documentation.

PROVE_TAC (bossLib)

```
bossLib.PROVE_TAC : thm list -> tactic
```

Also exported as `BasicProvers.PROVE_TAC`.

Solve a goal with use of hypotheses and supplied lemmas.

An invocation `PROVE_TAC thl` attempts to solve the goal it is applied to by executing a proof procedure that is semi-complete for pure first order logic. The assumptions of the goal and the theorems in `thl` are used. The procedure makes special provision for handling polymorphic and higher-order values (lambda terms). It also handles conditional expressions.

Failure

PROVE_TAC fails if it searches to a depth equal to the contents of the reference variable mesonLib.max_depth (set to 30 by default, but changeable by the user) without finding a proof.

Comments

PROVE_TAC can only progress the goal to a successful proof of the goal or not at all. In this respect it differs from tactics such as simplification and rewriting. Its ability to solve existential goals and to make effective use of transitivity theorems make it a particularly powerful tactic.

See also

bossLib.PROVE, mesonLib.MESON_TAC, mesonLib.ASM_MESON_TAC, mesonLib.GEN_-MESON_TAC

PROVEHYP_THEN (Thm_cont)

```
Thm_cont.PROVEHYP_THEN : (thm -> thm -> tactic) -> thm -> tactic
```

Makes antecedent of theorem as subgoal, continues with both parts as theorems.

An application of the tactic PROVEHYP_THEN th2tac th to a goal g requires that th be an (universally quantified) implication (or a negation, in which case ~p is treated as p ==> F). Given an implication |- !x1..xn. l ==> r x1..xn, the result is a new subgoal requiring the user to prove l, and the application of th2tac to the theorems with conclusion l and !x1..xn. r x1..xn.

Diagrammatically, one might see this as

```
        A ?- g
   ============================================  PROVEHYP_THEN th2tac th
   A ?- l  ...  th2tac (A |- l) (A |- r) (A ?- g)
```

Failure

Fails if the theorem argument is not an implication or negation.

Example

```
> FIRST_X_ASSUM (PROVEHYP_THEN (K MP_TAC)) ([“p”, “p ==> q”], “r”)
val it = ([([“p”], “p”), ([“p”], “q ==> r”)], fn):
         goal list * validation
```

The use of `FIRST_X_ASSUM` pulls out the first implicational theorem, and gives the user the requirement to prove `p` as a subgoal. In the other subgoal, `q` has become a new antecedent in the goal (thanks to the action of `MP_TAC`).

Comments

This function is also available under the name `provehyp_then`.

See also

`Tactic.impl_keep_tac`, `Tactic.impl_tac`

PRUNE_CONV (unwindLib)

`unwindLib.PRUNE_CONV : conv`

Prunes all hidden variables.

`PRUNE_CONV "?l1 ... lr. t1 /\ ... /\ eqn1 /\ ... /\ eqnr /\ ... /\ tp"` returns a theorem of the form:

```
|- (?l1 ... lr. t1 /\ ... /\ eqn1 /\ ... /\ eqnr /\ ... /\ tp) =
   (t1 /\ ... /\ tp)
```

where each `eqni` has the form `"!y1 ... ym. li x1 ... xn = b"` and `li` does not appear free in any of the other conjuncts or in `b`. The conversion works if one or more of

the eqni's are not present, that is if li is not free in any of the conjuncts, but does not work if li appears free in more than one of the conjuncts. p may be zero, that is, all the conjuncts may be eqni's. In this case the result will be simply T (true). Also, for each eqni, m and n may be zero.

Failure

Fails if the argument term is not of the specified form or if any of the li's are free in more than one of the conjuncts or if the equation for any li is recursive.

Example

```
#PRUNE_CONV
# "?l2 l1.
#   (!(x:num). l1 x = F) /\ (!x. l2 x = ~(out x)) /\ (!(x:num). out x = T)";;
|- (?l2 l1. (!x. l1 x = F) /\ (!x. l2 x = ~out x) /\ (!x. out x = T)) =
   (!x. out x = T)
```

See also

unwindLib.PRUNE_ONCE_CONV, unwindLib.PRUNE_ONE_CONV, unwindLib.PRUNE_SOME_CONV, unwindLib.PRUNE_SOME_RIGHT_RULE, unwindLib.PRUNE_RIGHT_RULE

PRUNE_ONCE_CONV (unwindLib)

```
unwindLib.PRUNE_ONCE_CONV : conv
```

Prunes one hidden variable.

PRUNE_ONCE_CONV "?l. t1 /\ ... /\ ti /\ eq /\ t(i+1) /\ ... /\ tp" returns a theorem of the form:

```
|- (?l. t1 /\ ... /\ ti /\ eq /\ t(i+1) /\ ... /\ tp) =
   (t1 /\ ... /\ ti /\ t(i+1) /\ ... /\ tp)
```

where eq has the form "!y1 ... ym. l x1 ... xn = b" and l does not appear free in the ti's or in b. The conversion works if eq is not present, that is if l is not free in any

of the conjuncts, but does not work if `l` appears free in more than one of the conjuncts. Each of `m`, `n` and `p` may be zero.

Failure

Fails if the argument term is not of the specified form or if `l` is free in more than one of the conjuncts or if the equation for `l` is recursive.

Example

```
#PRUNE_ONCE_CONV "?l2. (!(x:num). l1 x = F) /\ (!x. l2 x = ~(l1 x))";;
|- (?l2. (!x. l1 x = F) /\ (!x. l2 x = ~l1 x)) = (!x. l1 x = F)
```

See also

`unwindLib.PRUNE_ONE_CONV`, `unwindLib.PRUNE_SOME_CONV`, `unwindLib.PRUNE_CONV`, `unwindLib.PRUNE_SOME_RIGHT_RULE`, `unwindLib.PRUNE_RIGHT_RULE`

PRUNE_ONE_CONV (unwindLib)

```
unwindLib.PRUNE_ONE_CONV : (string -> conv)
```

Prunes a specified hidden variable.

`PRUNE_ONE_CONV` `` `lj` `` when applied to the term:

```
"?l1 ... lj ... lr. t1 /\ ... /\ ti /\ eq /\ t(i+1) /\ ... /\ tp"
```

returns a theorem of the form:

```
|- (?l1 ... lj ... lr. t1 /\ ... /\ ti /\ eq /\ t(i+1) /\ ... /\ tp) =
   (?l1 ... l(j-1) l(j+1) ... lr. t1 /\ ... /\ ti /\ t(i+1) /\ ... /\ tp)
```

where `eq` has the form `"!y1 ... ym. lj x1 ... xn = b"` and `lj` does not appear free in the `ti`'s or in `b`. The conversion works if `eq` is not present, that is if `lj` is not free

in any of the conjuncts, but does not work if `lj` appears free in more than one of the conjuncts. Each of `m`, `n` and `p` may be zero.

If there is more than one line with the specified name (but with different types), the one that appears outermost in the existential quantifications is pruned.

Failure

Fails if the argument term is not of the specified form or if `lj` is free in more than one of the conjuncts or if the equation for `lj` is recursive. The function also fails if the specified line is not one of the existentially quantified lines.

Example

```
#PRUNE_ONE_CONV `l2` "?l2 l1. (!(x:num). l1 x = F) /\ (!x. l2 x = ~(l1 x))";;
|- (?l2 l1. (!x. l1 x = F) /\ (!x. l2 x = ~l1 x)) = (?l1. !x. l1 x = F)

#PRUNE_ONE_CONV `l1` "?l2 l1. (!(x:num). l1 x = F) /\ (!x. l2 x = ~(l1 x))";;
evaluation failed     PRUNE_ONE_CONV
```

See also

`unwindLib.PRUNE_ONCE_CONV`, `unwindLib.PRUNE_SOME_CONV`, `unwindLib.PRUNE_CONV`, `unwindLib.PRUNE_SOME_RIGHT_RULE`, `unwindLib.PRUNE_RIGHT_RULE`

PRUNE_RIGHT_RULE (unwindLib)

```
unwindLib.PRUNE_RIGHT_RULE : (thm -> thm)
```

Prunes all hidden variables.

`PRUNE_RIGHT_RULE` behaves as follows:

```
 A |- !z1 ... zr.
       t = ?l1 ... lr. t1 /\ ... /\ eqn1 /\ ... /\ eqnr /\ ... /\ tp
-------------------------------------------------------------------
                A |- !z1 ... zr. t = t1 /\ ... /\ tp
```

where each eqni has the form "!y1 ... ym. li x1 ... xn = b" and li does not appear free in any of the other conjuncts or in b. The rule works if one or more of the eqni's are not present, that is if li is not free in any of the conjuncts, but does not work if li appears free in more than one of the conjuncts. p may be zero, that is, all the conjuncts may be eqni's. In this case the result will be simply T (true). Also, for each eqni, m and n may be zero.

Failure

Fails if the argument theorem is not of the specified form or if any of the li's are free in more than one of the conjuncts or if the equation for any li is recursive.

Example

```
#PRUNE_RIGHT_RULE
# (ASSUME
#   "!(in:num->bool) (out:num->bool).
#     DEV (in,out) =
#      ?(l1:num->bool) l2.
#       (!x. l1 x = F) /\ (!x. l2 x = ~(in x)) /\ (!x. out x = ~(in x))");;
. |- !in out. DEV(in,out) = (!x. out x = ~in x)
```

See also

unwindLib.PRUNE_SOME_RIGHT_RULE, unwindLib.PRUNE_ONCE_CONV, unwindLib.PRUNE_ONE_CONV, unwindLib.PRUNE_SOME_CONV, unwindLib.PRUNE_CONV

PRUNE_SOME_CONV (unwindLib)

```
unwindLib.PRUNE_SOME_CONV : (string list -> conv)
```

Prunes several hidden variables.

PRUNE_SOME_CONV [`li1`;...;`lik`] when applied to the term:

```
"?l1 ... lr. t1 /\ ... /\ eqni1 /\ ... /\ eqnik /\ ... /\ tp"
```

returns a theorem of the form:

```
|- (?l1 ... lr. t1 /\ ... /\ eqni1 /\ ... /\ eqnik /\ ... /\ tp) =
   (?li(k+1) ... lir. t1 /\ ... /\ tp)
```

where for `1 <= j <= k`, each `eqnij` has the form:

```
"!y1 ... ym. lij x1 ... xn = b"
```

and `lij` does not appear free in any of the other conjuncts or in `b`. The `li`'s are related by the equation:

```
{{li1,...,lik}} u {{li(k+1),...,lir}} = {{l1,...,lr}}
```

The conversion works if one or more of the `eqnij`'s are not present, that is if `lij` is not free in any of the conjuncts, but does not work if `lij` appears free in more than one of the conjuncts. `p` may be zero, that is, all the conjuncts may be `eqnij`'s. In this case the body of the result will be `T` (true). Also, for each `eqnij`, `m` and `n` may be zero.

If there is more than one line with a specified name (but with different types), the one that appears outermost in the existential quantifications is pruned. If such a line name is mentioned twice in the list, the two outermost occurrences of lines with that name will be pruned, and so on.

Failure

Fails if the argument term is not of the specified form or if any of the `lij`'s are free in more than one of the conjuncts or if the equation for any `lij` is recursive. The function also fails if any of the specified lines are not one of the existentially quantified lines.

Example

```
#PRUNE_SOME_CONV [`l1`;`l2`]
# "?l3 l2 l1.
#   (!(x:num). l1 x = F) /\ (!x. l2 x = ~(l3 x)) /\ (!(x:num). l3 x = T)";;
|- (?l3 l2 l1. (!x. l1 x = F) /\ (!x. l2 x = ~l3 x) /\ (!x. l3 x = T)) =
   (?l3. !x. l3 x = T)
```

See also

unwindLib.PRUNE_ONCE_CONV, unwindLib.PRUNE_ONE_CONV, unwindLib.PRUNE_CONV, unwindLib.PRUNE_SOME_RIGHT_RULE, unwindLib.PRUNE_RIGHT_RULE

PRUNE_SOME_RIGHT_RULE (unwindLib)

```
unwindLib.PRUNE_SOME_RIGHT_RULE : (string list -> thm -> thm)
```

Prunes several hidden variables.

PRUNE_SOME_RIGHT_RULE [`li1`;...;`lik`] behaves as follows:

```
    A |- !z1 ... zr.
          t = ?l1 ... lr. t1 /\ ... /\ eqni1 /\ ... /\ eqnik /\ ... /\ tp
   ------------------------------------------------------------------------
           A |- !z1 ... zr. t = ?li(k+1) ... lir. t1 /\ ... /\ tp
```

where for 1 <= j <= k, each eqnij has the form:

```
"!y1 ... ym. lij x1 ... xn = b"
```

and lij does not appear free in any of the other conjuncts or in b. The li's are related by the equation:

```
{{li1,...,lik}} u {{li(k+1),...,lir}} = {{l1,...,lr}}
```

The rule works if one or more of the eqnij's are not present, that is if lij is not free in any of the conjuncts, but does not work if lij appears free in more than one of the conjuncts. p may be zero, that is, all the conjuncts may be eqnij's. In this case the conjunction will be transformed to T (true). Also, for each eqnij, m and n may be zero.

If there is more than one line with a specified name (but with different types), the one that appears outermost in the existential quantifications is pruned. If such a line name is mentioned twice in the list, the two outermost occurrences of lines with that name will be pruned, and so on.

Failure

Fails if the argument theorem is not of the specified form or if any of the `lij`'s are free in more than one of the conjuncts or if the equation for any `lij` is recursive. The function also fails if any of the specified lines are not one of the existentially quantified lines.

Example

```
#PRUNE_SOME_RIGHT_RULE [`l1`;`l2`]
# (ASSUME
#   "!(in:num->bool) (out:num->bool).
#     DEV (in,out) =
#      ?(l1:num->bool) l2.
#       (!x. l1 x = F) /\ (!x. l2 x = ~(in x)) /\ (!x. out x = ~(in x))");;
. |- !in out. DEV(in,out) = (!x. out x = ~in x)
```

See also

`unwindLib.PRUNE_RIGHT_RULE`, `unwindLib.PRUNE_ONCE_CONV`, `unwindLib.PRUNE_ONE_CONV`, `unwindLib.PRUNE_SOME_CONV`, `unwindLib.PRUNE_CONV`

PSELECT_CONV (PairRules)

`PairRules.PSELECT_CONV : conv`

Eliminates a paired epsilon term by introducing an existential quantifier.

The conversion `PSELECT_CONV` expects a boolean term of the form `"t[@p.t[p]/p]"`, which asserts that the epsilon term `@p.t[p]` denotes a pair, `p` say, for which `t[p]` holds. This assertion is equivalent to saying that there exists such a pair, and `PSELECT_CONV` applied to a term of this form returns the theorem `|- t[@p.t[p]/p] = ?p. t[p]`.

Failure

Fails if applied to a term that is not of the form `"p[@p.t[p]/p]"`.

See also

Conv.SELECT_CONV, PairRules.PSELECT_ELIM, PairRules.PSELECT_INTRO, PairRules.PSELECT_-RULE

PSELECT_ELIM (PairRules)

```
PairRules.PSELECT_ELIM : thm -> term * thm -> thm
```

Eliminates a paired epsilon term, using deduction from a particular instance.

PSELECT_ELIM expects two arguments, a theorem th1, and a pair (p,th2): term * thm. The conclusion of th1 must have the form P($@ P), which asserts that the epsilon term $@ P denotes some value at which P holds. The paired variable structure p appears only in the assumption P p of the theorem th2. The conclusion of the resulting theorem matches that of th2, and the hypotheses include the union of all hypotheses of the premises excepting P p.

```
 A1 |- P($@ P)     A2 u {P p} |- t
-----------------------------------  PSELECT_ELIM th1 (p ,th2)
          A1 u A2 |- t
```

where p is not free in A2. If p appears in the conclusion of th2, the epsilon term will NOT be eliminated, and the conclusion will be t[$@ P/p].

Failure

Fails if the first theorem is not of the form A1 |- P($@ P), or if any of the variables from the variable structure p occur free in any other assumption of th2.

See also

Drule.SELECT_ELIM, PairRules.PCHOOSE, PairRules.PSELECT_CONV, PairRules.PSELECT_-INTRO, PairRules.PSELECT_RULE

PSELECT_EQ (PairRules)

```
PairRules.PSELECT_EQ : (term -> thm -> thm)
```

Applies epsilon abstraction to both terms of an equation.

When applied to a paired structure of variables `p` and a theorem whose conclusion is equational:

```
A |- t1 = t2
```

the inference rule `PSELECT_EQ` returns the theorem:

```
A |- (@p. t1) = (@p. t2)
```

provided no variable in `p` is free in the assumptions.

```
      A |- t1 = t2
--------------------------  SELECT_EQ "p"      [where p is not free in A]
 A |- (@p. t1) = (@p. t2)
```

Failure

Fails if the conclusion of the theorem is not an equation, of if `p` is not a paired structure of variables, or if any variable in `p` is free in `A`.

See also

`Drule.SELECT_EQ`, `PairRules.PFORALL_EQ`, `PairRules.PEXISTS_EQ`

PSELECT_INTRO (PairRules)

```
PairRules.PSELECT_INTRO : (thm -> thm)
```

Introduces an epsilon term.

PSELECT_INTRO takes a theorem with an applicative conclusion, say P x, and returns a theorem with the epsilon term $@ P in place of the original operand x.

```
    A |- P x
--------------  PSELECT_INTRO
 A |- P($@ P)
```

The returned theorem asserts that $@ P denotes some value at which P holds.

Failure

Fails if the conclusion of the theorem is not an application.

Comments

This function is exactly the same as SELECT_INTRO, it is duplicated in the pair library for completeness.

See also

Drule.SELECT_INTRO, PairRules.PEXISTS, PairRules.PSELECT_CONV, PairRules.PSELECT_ELIM, PairRules.PSELECT_RULE

PSELECT_RULE (PairRules)

```
PairRules.PSELECT_RULE : (thm -> thm)
```

Introduces a paired epsilon term in place of a paired existential quantifier.

The inference rule PSELECT_RULE expects a theorem asserting the existence of a pair p such that t holds. The equivalent assertion that the epsilon term @p.t denotes a pair p for which t holds is returned as a theorem.

```
    A |- ?p. t
-----------------  PSELECT_RULE
 A |- t[(@p.t)/p]
```

Failure

Fails if applied to a theorem the conclusion of which is not a paired existential quantifier.

See also

`Drule.SELECT_RULE`, `PairRules.PCHOOSE`, `PairRules.PSELECT_CONV`, `PairRules.PEXISTS_CONV`, `PairRules.PSELECT_ELIM`, `PairRules.PSELECT_INTRO`

PSKOLEM_CONV (PairRules)

```
PairRules.PSKOLEM_CONV : conv
```

Proves the existence of a pair of Skolem functions.

When applied to an argument of the form `!p1...pn. ?q. tm`, the conversion `PSKOLEM_CONV` returns the theorem:

```
|- (!p1...pn. ?q. tm) = (?q'. !p1...pn. tm[q' p1 ... pn/yq)
```

where `q'` is a primed variant of the pair `q` not free in the input term.

Failure

`PSKOLEM_CONV tm` fails if `tm` is not a term of the form `!p1...pn. ?q. tm`.

Example

Both `q` and any `pi` may be a paired structure of variables:

```
- PSKOLEM_CONV
   (Term `!(x11:'a,x12:'a) (x21:'a,x22:'a).
          ?(y1:'a,y2:'a). tm x11 x12 x21 x21 y1 y2`);

> val it =
 |- (!(x11,x12) (x21,x22). ?(y1,y2). tm x11 x12 x21 x21 y1 y2) =
```

```
    ?(y1,y2).
      !(x11,x12) (x21,x22).
        tm x11 x12 x21 x21 (y1 (x11,x12) (x21,x22)) (y2 (x11,x12) (x21,x22))
  : thm
```

See also

Conv.SKOLEM_CONV, PairRules.P_PSKOLEM_CONV

PSPEC (PairRules)

```
PairRules.PSPEC : (term -> thm -> thm)
```

Specializes the conclusion of a theorem.

When applied to a term q and a theorem `A |- !p. t`, then PSPEC returns the theorem `A |- t[q/p]`. If necessary, variables will be renamed prior to the specialization to ensure that q is free for p in t, that is, no variables free in q become bound after substitution.

```
   A |- !p. t
--------------  PSPEC "q"
 A |- t[q/p]
```

Failure

Fails if the theorem's conclusion is not a paired universal quantification, or if p and q have different types.

Example

PSPEC specialised paired quantifications.

```
- PSPEC (Term `(1,2)`) (ASSUME (Term`!(x,y). (x + y) = (y + x)`));
> val it =  [.] |- 1 + 2 = 2 + 1 : thm
```

PSPEC treats paired structures of variables as variables and preserves structure accordingly.

```
- PSPEC (Term `x:'a#'a`) (ASSUME (Term `!(x:'a,y:'a). (x,y) = (x,y)`));
> val it =  [.] |- x = x : thm
```

See also

Thm.SPEC, PairRules.IPSPEC, PairRules.PSPECL, PairRules.PSPEC_ALL, PairRules.PGEN, PairRules.PGENL

PSPEC_ALL (PairRules)

```
PairRules.PSPEC_ALL : (thm -> thm)
```

Specializes the conclusion of a theorem with its own quantified pairs.

When applied to a theorem `A |- !p1...pn. t`, the inference rule PSPEC_ALL returns the theorem `A |- t[p1'/p1]...[pn'/pn]` where the `pi'` are distinct variants of the corresponding `pi`, chosen to avoid clashes with any variables free in the assumption list and with the names of constants. Normally `pi'` is just `pi`, in which case PSPEC_ALL simply removes all universal quantifiers.

```
    A |- !p1...pn. t
--------------------------  PSPEC_ALL
 A |- t[p1'/x1]...[pn'/xn]
```

Failure

Never fails.

See also

Drule.SPEC_ALL, PairRules.PGEN, PairRules.PGENL, PairRules.PGEN_TAC, PairRules.PSPEC, PairRules.PSPECL, PairRules.PSPEC_TAC

PSPEC_PAIR (PairRules)

```
PairRules.PSPEC_PAIR : thm -> term * thm
```

Specializes the conclusion of a theorem, returning the chosen variant.

When applied to a theorem `A |- !p. t`, the inference rule `PSPEC_PAIR` returns the term `q'` and the theorem `A |- t[q'/p]`, where `q'` is a variant of `p` chosen to avoid free variable capture.

```
  A |- !p. t
-------------  PSPEC_PAIR
 A |- t[q'/q]
```

Failure

Fails unless the theorem's conclusion is a paired universal quantification.

Comments

This rule is very similar to plain `PSPEC`, except that it returns the variant chosen, which may be useful information under some circumstances.

See also

`Drule.SPEC_VAR`, `PairRules.PGEN`, `PairRules.PGENL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`

PSPEC_TAC (PairRules)

```
PairRules.PSPEC_TAC : term * term -> tactic
```

Generalizes a goal.

When applied to a pair of terms `(q,p)`, where `p` is a paired structure of variables and a goal `A ?- t`, the tactic `PSPEC_TAC` generalizes the goal to `A ?- !p. t[p/q]`, that is, all components of `q` are turned into the corresponding components of `p`.

```
     A ?- t
================  PSPEC_TAC (q,p)
 A ?- !x. t[p/q]
```

Failure

Fails unless `p` is a paired structure of variables with the same type as `q`.

Example

```
> g `1 + 2 = 2 + 1`;
val it =
   Proof manager status: 2 proofs.
   2. Incomplete goalstack:
        Initial goal:
        ∃R. WF R ∧ (∀rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)) ∧
            ∀rst x ord. R (ord,FILTER ($¬ ∘ ord x) rst) (ord,x::rst)

   1. Incomplete goalstack:
        Initial goal:
        1 + 2 = 2 + 1

> e (PairRules.PSPEC_TAC (Term`(1,2)`, Term`(x:num,y:num)`));
OK..
1 subgoal:
val it =

   ∀(x,y). x + y = y + x
```

Removing unnecessary speciality in a goal, particularly as a prelude to an inductive proof.

See also

`PairRules.PGEN`, `PairRules.PGENL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSTRIP_TAC`

PSPECL (PairRules)

```
PairRules.PSPECL : (term list -> thm -> thm)
```

Specializes zero or more pairs in the conclusion of a theorem.

When applied to a term list `[q1;...;qn]` and a theorem `A |- !p1...pn. t`, the inference rule `SPECL` returns the theorem `A |- t[q1/p1]...[qn/pn]`, where the substitutions are made sequentially left-to-right in the same way as for `PSPEC`.

```
     A |- !p1...pn. t
--------------------------  SPECL "[q1;...;qn]"
  A |- t[q1/p1]...[qn/pn]
```

It is permissible for the term-list to be empty, in which case the application of `PSPECL` has no effect.

Failure

Fails unless each of the terms is of the same type as that of the appropriate quantified variable in the original theorem. Fails if the list of terms is longer than the number of quantified pairs in the theorem.

See also

`Drule.SPECL`, `PairRules.PGEN`, `PairRules.PGENL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`

PSTRIP_ASSUME_TAC (PairRules)

```
PairRules.PSTRIP_ASSUME_TAC : thm_tactic
```

Splits a theorem into a list of theorems and then adds them to the assumptions.

Given a theorem `th` and a goal `(A,t)`, `PSTRIP_ASSUME_TAC th` splits `th` into a list of theorems. This is done by recursively breaking conjunctions into separate conjuncts, cases-splitting disjunctions, and eliminating paired existential quantifiers by choosing arbitrary variables. Schematically, the following rules are applied:

```
         A ?- t
======================  PSTRIP_ASSUME_TAC (A' |- v1 /\ ... /\ vn)
```

```
    A u {v1,...,vn} ?- t

                A ?- t
   ================================  PSTRIP_ASSUME_TAC (A' |- v1 \/ ... \/ vn)
    A u {v1} ?- t ... A u {vn} ?- t

          A ?- t
   ====================  PSTRIP_ASSUME_TAC (A' |- ?p. v)
    A u {v[p'/p]} ?- t
```

where `p'` is a variant of the pair `p`.

If the conclusion of `th` is not a conjunction, a disjunction or a paired existentially quantified term, the whole theorem `th` is added to the assumptions.

As assumptions are generated, they are examined to see if they solve the goal (either by being alpha-equivalent to the conclusion of the goal or by deriving a contradiction).

The assumptions of the theorem being split are not added to the assumptions of the goal(s), but they are recorded in the proof. This means that if `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), `PSTRIP_ASSUME_TAC (A'|-v)` results in an invalid tactic.

Failure

Never fails.

`PSTRIP_ASSUME_TAC` is used when applying a previously proved theorem to solve a goal, or when enriching its assumptions so that resolution, rewriting with assumptions and other operations involving assumptions have more to work with.

See also

`PairRules.PSTRIP_THM_THEN`, `PairRules.PSTRIP_ASSUME_TAC`, `PairRules.PSTRIP_GOAL_THEN`, `PairRules.PSTRIP_TAC`

PSTRIP_GOAL_THEN (PairRules)

```
PairRules.PSTRIP_GOAL_THEN : (thm_tactic -> tactic)
```

Splits a goal by eliminating one outermost connective, applying the given theorem-tactic to the antecedents of implications.

Given a theorem-tactic `ttac` and a goal `(A,t)`, `PSTRIP_GOAL_THEN` removes one outermost occurrence of one of the connectives `!`, `==>`, `~` or `/\` from the conclusion of the goal `t`. If `t` is a universally quantified term, then `STRIP_GOAL_THEN` strips off the quantifier. Note that `PSTRIP_GOAL_THEN` will strip off paired universal quantifications.

```
    A ?- !p. u
==============  PSTRIP_GOAL_THEN ttac
 A ?- u[p'/p]
```

where `p'` is a primed variant that contains no variables that appear free in the assumptions `A`. If `t` is a conjunction, then `PSTRIP_GOAL_THEN` simply splits the conjunction into two subgoals:

```
    A ?- v /\ w
=================  PSTRIP_GOAL_THEN ttac
 A ?- v   A ?- w
```

If `t` is an implication `"u ==> v"` and if:

```
    A ?- v
===============  ttac (u |- u)
   A' ?- v'
```

then:

```
    A ?- u ==> v
====================  PSTRIP_GOAL_THEN ttac
      A' ?- v'
```

Finally, a negation `~t` is treated as the implication `t ==> F`.

Failure

`PSTRIP_GOAL_THEN ttac (A,t)` fails if `t` is not a paired universally quantified term, an implication, a negation or a conjunction. Failure also occurs if the application of `ttac` fails, after stripping the goal.

`PSTRIP_GOAL_THEN` is used when manipulating intermediate results (obtained by stripping outer connectives from a goal) directly, rather than as assumptions.

See also

PairRules.PGEN_TAC, Tactic.STRIP_GOAL_THEN, PairRules.FILTER_PSTRIP_THEN, PairRules.PSTRIP_TAC, PairRules.FILTER_PSTRIP_TAC

PSTRIP_TAC (PairRules)

PairRules.PSTRIP_TAC : tactic

Splits a goal by eliminating one outermost connective.

Given a goal (A,t), PSTRIP_TAC removes one outermost occurrence of one of the connectives !, ==>, ~ or /\ from the conclusion of the goal t. If t is a universally quantified term, then STRIP_TAC strips off the quantifier. Note that PSTRIP_TAC will strip off paired quantifications.

```
  A ?- !p. u
============  PSTRIP_TAC
 A ?- u[p'/p]
```

where p' is a primed variant of the pair p that does not contain any variables that appear free in the assumptions A. If t is a conjunction, then PSTRIP_TAC simply splits the conjunction into two subgoals:

```
   A ?- v /\ w
================  PSTRIP_TAC
 A ?- v   A ?- w
```

If t is an implication, PSTRIP_TAC moves the antecedent into the assumptions, stripping conjunctions, disjunctions and existential quantifiers according to the following rules:

```
 A ?- v1 /\ ... /\ vn ==> v
============================
    A u {v1,...,vn} ?- v
```

```
   A ?- v1 \/ ... \/ vn ==> v
================================
 A u {v1} ?- v ... A u {vn} ?- v
```

```
  A ?- (?p. w) ==> v
====================
 A u {w[p'/p]} ?- v
```

where p' is a primed variant of the pair p that does not appear free in A. Finally, a negation ~t is treated as the implication t ==> F.

Failure

PSTRIP_TAC (A,t) fails if t is not a paired universally quantified term, an implication, a negation or a conjunction.

When trying to solve a goal, often the best thing to do first is REPEAT PSTRIP_TAC to split the goal up into manageable pieces.

See also

PairRules.PGEN_TAC, PairRules.PSTRIP_GOAL_THEN, PairRules.FILTER_PSTRIP_THEN, Tactic.STRIP_TAC, PairRules.FILTER_PSTRIP_TAC

PSTRIP_THM_THEN (PairRules)

PairRules.PSTRIP_THM_THEN : thm_tactical

PSTRIP_THM_THEN applies the given theorem-tactic using the result of stripping off one outer connective from the given theorem.

Given a theorem-tactic ttac, a theorem th whose conclusion is a conjunction, a disjunction or a paired existentially quantified term, and a goal (A,t), STRIP_THM_THEN ttac th first strips apart the conclusion of th, next applies ttac to the theorem(s) resulting from the stripping and then applies the resulting tactic to the goal.

In particular, when stripping a conjunctive theorem A'|- u /\ v, the tactic

```
ttac(u|-u) THEN ttac(v|-v)
```

resulting from applying ttac to the conjuncts, is applied to the goal. When stripping a disjunctive theorem A'|- u \/ v, the tactics resulting from applying ttac to the disjuncts, are applied to split the goal into two cases. That is, if

```
 A ?- t                          A ?- t
=========  ttac (u|-u)    and   =========  ttac (v|-v)
 A ?- t1                         A ?- t2
```

then:

```
       A ?- t
================== PSTRIP_THM_THEN ttac (A'|- u \/ v)
 A ?- t1  A ?- t2
```

When stripping a paired existentially quantified theorem `A'|- ?p. u`, the tactic resulting from applying `ttac` to the body of the paired existential quantification, `ttac(u|-u)`, is applied to the goal. That is, if:

```
 A ?- t
=========  ttac (u|-u)
 A ?- t1
```

then:

```
   A ?- t
=============  PSTRIP_THM_THEN ttac (A'|- ?p. u)
   A ?- t1
```

The assumptions of the theorem being split are not added to the assumptions of the goal(s) but are recorded in the proof. If `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), `PSTRIP_THM_THEN ttac th` results in an invalid tactic.

Failure

`PSTRIP_THM_THEN ttac th` fails if the conclusion of `th` is not a conjunction, a disjunction or a paired existentially quantification. Failure also occurs if the application of `ttac` fails, after stripping the outer connective from the conclusion of `th`.

`PSTRIP_THM_THEN` is used enrich the assumptions of a goal with a stripped version of a previously-proved theorem.

See also

`Thm_cont.STRIP_THM_THEN`, `PairRules.PSTRIP_ASSUME_TAC`, `PairRules.PSTRIP_GOAL_THEN`, `PairRules.PSTRIP_TAC`

PSTRUCT_CASES_TAC (PairRules)

```
PairRules.PSTRUCT_CASES_TAC : thm_tactic
```

Performs very general structural case analysis.

When it is applied to a theorem of the form:

```
th = A' |- ?p11...p1m. (x=t1) /\ (B11 /\ ... /\ B1k) \/ ... \/
           ?pn1...pnp. (x=tn) /\ (Bn1 /\ ... /\ Bnp)
```

in which there may be no paired existential quantifiers where a 'vector' of them is shown above, `PSTRUCT_CASES_TAC th` splits a goal `A ?- s` into `n` subgoals as follows:

```
                          A ?- s
==========================================================
 A u {B11,...,B1k} ?- s[t1/x] ... A u {Bn1,...,Bnp} ?- s[tn/x]
```

that is, performs a case split over the possible constructions (the `ti`) of a term, providing as assumptions the given constraints, having split conjoined constraints into separate assumptions. Note that unless `A'` is a subset of `A`, this is an invalid tactic.

Failure

Fails unless the theorem has the above form, namely a conjunction of (possibly multiply paired existentially quantified) terms which assert the equality of the same variable `x` and the given terms.

Generating a case split from the axioms specifying a structure.

See also

`Tactic.STRUCT_CASES_TAC`

PSUB_CONV (PairRules)

```
PairRules.PSUB_CONV : (conv -> conv)
```

Applies a conversion to the top-level subterms of a term.

For any conversion `c`, the function returned by `PSUB_CONV c` is a conversion that applies `c` to all the top-level subterms of a term. If the conversion `c` maps `t` to `|- t = t'`, then `SUB_CONV c` maps a paired abstraction `"\p.t"` to the theorem:

```
|- (\p.t) = (\p.t')
```

That is, `PSUB_CONV c "\p.t"` applies `c` to the body of the paired abstraction `"\p.t"`. If `c` is a conversion that maps `"t1"` to the theorem `|- t1 = t1'` and `"t2"` to the theorem `|- t2 = t2'`, then the conversion `PSUB_CONV c` maps an application `"t1 t2"` to the theorem:

```
|- (t1 t2) = (t1' t2')
```

That is, `PSUB_CONV c "t1 t2"` applies `c` to the both the operator `t1` and the operand `t2` of the application `"t1 t2"`. Finally, for any conversion `c`, the function returned by `PSUB_CONV c` acts as the identity conversion on variables and constants. That is, if `"t"` is a variable or constant, then `PSUB_CONV c "t"` returns `|- t = t`.

Failure

`PSUB_CONV c tm` fails if `tm` is a paired abstraction `"\p.t"` and the conversion `c` fails when applied to `t`, or if `tm` is an application `"t1 t2"` and the conversion `c` fails when applied to either `t1` or `t2`. The function returned by `PSUB_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

See also

`Conv.SUB_CONV`, `PairRules.PABS_CONV`, `Conv.RAND_CONV`, `Conv.RATOR_CONV`

Psyntax (structure)

```
Psyntax : Psyntax_sig
```

A structure that provides a tuple-style environment for term manipulation.

Each function in the `Psyntax` structure has a corresponding "record version" in the `Rsyntax` structure, and vice versa. One can flip-flop between the two structures by opening one and then the other. One can also use long identifiers in order to use both syntaxes at once.

Failure

Never fails.

Example

The following shows how to open the `Psyntax` structure and the functions that subsequently become available in the top level environment. Documentation for each of these functions is available online.

```
- open Psyntax;
```

This command results in the following functions entering the top-level name-space. Term creation functions:

```
val mk_var = fn : string * hol_type -> term
val mk_const = fn : string * hol_type -> term
val mk_comb = fn : term * term -> term
val mk_abs = fn : term * term -> term
val mk_primed_var = fn : string * hol_type -> term
val mk_eq = fn : term * term -> term
val mk_imp = fn : term * term -> term
val mk_select = fn : term * term -> term
val mk_forall = fn : term * term -> term
val mk_exists = fn : term * term -> term
val mk_conj = fn : term * term -> term
val mk_disj = fn : term * term -> term
val mk_cond = fn : term * term * term -> term
val mk_let = fn : term * term -> term
```

Term "destructor" functions (i.e., those functions that pull a term apart, and reveal some of its internal structure):

```
val dest_var = fn : term -> string * hol_type
val dest_const = fn : term -> string * hol_type
val dest_comb = fn : term -> term * term
val dest_abs = fn : term -> term * term
val dest_eq = fn : term -> term * term
val dest_imp = fn : term -> term * term
val dest_select = fn : term -> term * term
val dest_forall = fn : term -> term * term
val dest_exists = fn : term -> term * term
val dest_conj = fn : term -> term * term
val dest_disj = fn : term -> term * term
val dest_cond = fn : term -> term * term * term
val dest_let = fn : term -> term * term
```

The `lambda` datatype for taking terms apart, which is the range of the `dest_term` function.

```
datatype lambda =
   VAR of string * hol_type
 | CONST of {Name : string, Thy : string, Ty : hol_type}
 | COMB of term * term
 | LAMB of term * term
val dest_term : term -> lambda
```

See also

`Rsyntax`

PTAUT_CONV (tautLib)

`tautLib.PTAUT_CONV : conv`

Tautology checker. Proves closed propositional formulae true or false.

Given a term of the form `"!x1 ... xn. t"` where `t` contains only Boolean constants, Boolean-valued variables, Boolean equalities, implications, conjunctions, disjunctions, negations and Boolean-valued conditionals, and all the variables in `t` appear in `x1 ... xn`, the conversion `PTAUT_CONV` proves the term to be either true or false, that is, one of the following theorems is returned:

```
|- (!x1 ... xn. t) = T
|- (!x1 ... xn. t) = F
```

This conversion also accepts propositional terms that are not fully universally quantified. However, for such a term, the conversion will only succeed if the term is valid.

Failure

Fails if the term is not of the form `"!x1 ... xn. f[x1,...,xn]"` where `f[x1,...,xn]` is a propositional formula (except that the variables do not have to be universally quantified if the term is valid).

Example

```
#PTAUT_CONV ``!x y z w. (((x \/ ~y) ==> z) /\ (z ==> ~w) /\ w) ==> y``;
|- (!x y z w. (x \/ ~y ==> z) /\ (z ==> ~w) /\ w ==> y) = T

#PTAUT_CONV ``(((x \/ ~y) ==> z) /\ (z ==> ~w) /\ w) ==> y``;
|- (x \/ ~y ==> z) /\ (z ==> ~w) /\ w ==> y = T

#PTAUT_CONV ``!x. x = T``;
|- (!x. x = T) = F

#PTAUT_CONV ``x = T``;
Uncaught exception:
HOL_ERR
```

See also

`tautLib.PTAUT_PROVE`, `tautLib.PTAUT_TAC`, `tautLib.TAUT_CONV`

PTAUT_PROVE (tautLib)

```
tautLib.PTAUT_PROVE : term -> thm
```

Tautology checker. Proves propositional formulae.

Given a term that contains only Boolean constants, Boolean-valued variables, Boolean equalities, implications, conjunctions, disjunctions, negations and Boolean-valued conditionals, PTAUT_PROVE returns the term as a theorem if it is valid. The variables in the term may be universally quantified.

Failure

Fails if the term is not a valid propositional formula.

Example

```
#PTAUT_PROVE ``!x y z w. (((x \/ ~y) ==> z) /\ (z ==> ~w) /\ w) ==> y``;
|- !x y z w. (x \/ ~y ==> z) /\ (z ==> ~w) /\ w ==> y

#PTAUT_PROVE ``(((x \/ ~y) ==> z) /\ (z ==> ~w) /\ w) ==> y``;
|- (x \/ ~y ==> z) /\ (z ==> ~w) /\ w ==> y

#PTAUT_PROVE ``!x. x = T``;
Uncaught exception:
HOL_ERR

#PTAUT_PROVE ``x = T``;
Uncaught exception:
HOL_ERR
```

See also

tautLib.PTAUT_CONV, tautLib.PTAUT_TAC, tautLib.TAUT_PROVE

PTAUT_TAC (tautLib)

```
tautLib.PTAUT_TAC : tactic
```

Tautology checker. Proves propositional goals.

Given a goal with a conclusion that contains only Boolean constants, Boolean-valued variables, Boolean equalities, implications, conjunctions, disjunctions, negations and Boolean-valued conditionals, this tactic will prove the goal if it is valid. If all the variables in the conclusion are universally quantified, this tactic will also reduce an invalid goal to false.

Failure

Fails if the conclusion of the goal is not of the form `!x1 ... xn. f[x1,...,xn]` where `f[x1,...,xn]` is a propositional formula (except that the variables do not have to be universally quantified if the goal is valid).

See also

`tautLib.PTAUT_CONV`, `tautLib.PTAUT_PROVE`, `tautLib.TAUT_TAC`

PTREE_ADD_CONV (patriciaLib)

`patriciaLib.PTREE_ADD_CONV : conv`

Conversion for evaluating applications of `patricia$ADD` and `patricia$ADD_LIST`.

The conversion `PTREE_ADD_CONV` evaluates terms of the form `t |+ (m,n)` or `t |++ l` where `t` is a well-formed Patricia tree (correctly constructed using `patricia$Empty`, `patricia$Leaf` and `patricia$Branch`) and `m` is a natural number literal.

Failure

The conversion will fail if the supplied term is not a suitable application of `patricia$ADD` or `patricia$ADD_LIST`.

Example

```
> patriciaLib.PTREE_ADD_CONV ``Empty |+ (3, x:num)``;
val it = ⊢ <{}> |+ (3,x) = Leaf 3 x: thm
```

```
> DEPTH_CONV patriciaLib.PTREE_ADD_CONV ``Empty |+ (3, 2) |+ (2,1)``;
val it = ⊢ <{}> |+ (3,2) |+ (2,1) = Branch 0 0 (Leaf 3 2) (Leaf 2 1): thm
```

See also

`patriciaLib.PTREE_CONV`

PTREE_CONV (patriciaLib)

`patriciaLib.PTREE_CONV : conv`

Conversion for evaluating Patricia tree operations.

The conversion `PTREE_CONV` evaluates Patricia tree operations such as `ADD`, `ADD_LIST`, `REMOVE`, `SIZE`, `PEEK` and `FIND`. These evaluations work for constants that are defined using `Define_mk_ptree`. When adding to, or removing from, a Patricia tree a new contant will be defined after `patriciaLib.ptree_new_defn_depth` operations. By default `ptree_new_defn_depth` is `~1`, which means that new constants are never defined.

Example

Consider the following Patricia tree:

```
val ptree = Define_mk_ptree "ptree" (int_ptree_of_list [(1,``1``), (2, ``2``)]);
<<HOL message: Saved IS_PTREE theorem for new constant "ptree">>
val ptree = |- ptree = Branch 0 0 (Leaf 1 1) (Leaf 2 2): thm
```

Adding a list of updates expands into applications of `ADD`:

```
> real_time patriciaLib.PTREE_CONV ``ptree |++ [(3,3); (4,4); (5,5); (6,6); (7,7)]``;
realtime: 0.000s
val it =
   ⊢ ptree |++ [(3,3); (4,4); (5,5); (6,6); (7,7)] =
     ptree |+ (3,3) |+ (4,4) |+ (5,5) |+ (6,6) |+ (7,7): thm
```

However, setting ptree_new_defn_depth will cause new definitions to be made:

```
> patriciaLib.ptree_new_defn_depth := 2;
val it = (): unit
> real_time patriciaLib.PTREE_CONV ``ptree |++ [(3,3); (4,4); (5,5); (6,6); (7,7)]`
realtime: 0.000s
val it =
   ⊢ ptree |++ [(3,3); (4,4); (5,5); (6,6); (7,7)] =
     ptree |+ (3,3) |+ (4,4) |+ (5,5) |+ (6,6) |+ (7,7): thm
```

New definitions will also be made when removing elements:

```
> real_time patriciaLib.PTREE_CONV ``ptree2 \\ 6 \\ 5``;
realtime: 0.000s
val it = ⊢ ptree2 \\ 6 \\ 5 = ptree2 \\ 6 \\ 5: thm
```

Here, the conversion is not smart enough to work out that ptree3 is the same as ptree1.

```
> (DEPTH_CONV patriciaLib.PTREE_DEFN_CONV THENC EVAL) ``ptree1 = ptree3``;
val it = ⊢ ptree1 = ptree3 ⇔ ptree1 = ptree3: thm
```

Look-up behaves as expected:

```
> real_time patriciaLib.PTREE_CONV ``ptree1 ' 2``;
realtime: 0.000s
val it = ⊢ ptree1 ' 2 = ptree1 ' 2: thm
> real_time patriciaLib.PTREE_CONV ``ptree1 ' 5``;
realtime: 0.000s
val it = ⊢ ptree1 ' 5 = ptree1 ' 5: thm
```

Comments

The conversion PTREE_CONV is automatically added to the standard compset. Thus, EVAL will have the same behaviour when patriciaLib is loaded.

Run-times should be respectable when working with large Patricia trees. However, this is predicated on the assumption that relatively small numbers of updates are made following an initial application of `Define_mk_ptree`. In this sense, the Patricia tree development is best suited to situations where users require fast "read-only" look-up; where the work of building the look-up tree can be performed outside of the logic (i.e. in ML).

See also

`patriciaLib.Define_mk_ptree`, `patriciaLib.PTREE_DEFN_CONV`

PTREE_DEFN_CONV (patriciaLib)

```
patriciaLib.PTREE_DEFN_CONV : conv
```

Conversion for evaluating applications of `ADD` and `REMOVE` to Patricia tree constants.

Given a constant `c` defined using `Define_mk_ptree`, the conversion `PTREE_DEFN_CONV` will evaluate term of the form `c |+ (k,x)`, `c \\ k` and `c` where `k` is a natural number literal.

Example

```
> val ptree = patriciaLib.Define_mk_ptree "ptree" (patriciaLib.int_ptree_of_list [(1,``1
val ptree = ⊢ ptree = Branch 0 0 (Leaf 1 1) (Leaf 2 2): thm

> patriciaLib.PTREE_DEFN_CONV ``ptree \\ 1``;
val it = ⊢ ptree \\ 1 = Leaf 2 2: thm

> patriciaLib.PTREE_DEFN_CONV ``ptree |+ (3,3)``;
val it =
   ⊢ ptree |+ (3,3) =
     Branch 0 0 (Branch 1 1 (Leaf 3 3) (Leaf 1 1)) (Leaf 2 2): thm
```

Comments

The conversion PTREE_DEFN_CONV has limited uses and is mostly used internally by the conversion PTREE_CONV.

See also

patriciaLib.Define_mk_ptree, patriciaLib.PTREE_CONV

PTREE_DEPTH_CONV (patriciaLib)

```
patriciaLib.PTREE_DEPTH_CONV : conv
```

Conversion for evaluating applications of patricia$DEPTH.

The conversion PTREE_DEPTH_CONV evaluates terms of the form DEPTH t where t is a well-formed Patricia tree (constructed by patricia$Empty, patricia$Leaf and patricia$Branch).

Failure

The conversion will fail if the supplied term is not a suitable application of patricia$DEPTH.

Example

```
> patriciaLib.PTREE_DEPTH_CONV ``DEPTH Empty``;
val it = ⊢ DEPTH <{}> = 0: thm

> patriciaLib.PTREE_DEPTH_CONV ``DEPTH (Branch 0 0 (Leaf 3 2) (Leaf 2 1))``;
val it = ⊢ DEPTH (Branch 0 0 (Leaf 3 2) (Leaf 2 1)) = 2: thm
```

See also

patriciaLib.PTREE_CONV

PTREE_EVERY_LEAF_CONV (patriciaLib)

`patriciaLib.PTREE_EVERY_LEAF_CONV : conv`

Conversion for evaluating applications of `patricia$EVERY_LEAF`.

The conversion `PTREE_EVERY_LEAF_CONV` evaluates terms of the form `EVERY_LEAF P t` where `t` is a well-formed Patricia tree (constructed by `patricia$Empty`, `patricia$Leaf` and `patricia$Branch`) and `P` is predicate.

Failure

The conversion will fail if the supplied term is not a suitable application of `patricia$EVERY_LEAF`.

Example

```
> patriciaLib.PTREE_EVERY_LEAF_CONV ``EVERY_LEAF (=) Empty``;
Exception- Type error in function application.
   Function: patriciaLib.PTREE_EVERY_LEAF_CONV : conv -> conv
   Argument: (Parse.Term [QUOTE " (*#loc 1 37*)EVERY_LEAF (=) Empty"])
      : term
   Reason: Can't unify term to term -> thm (Incompatible types)
Fail "Static Errors" raised

> patriciaLib.PTREE_EVERY_LEAF_CONV ``EVERY_LEAF (\x y. (x < 3) ==> (y = 1)) (Branch 0 0
Exception- Type error in function application.
   Function: patriciaLib.PTREE_EVERY_LEAF_CONV : conv -> conv
   Argument:
      (
      Parse.Term
      [
         QUOTE
         " (*#loc 1 37*)EVERY_LEAF (\\x y. (x < 3) ==> (y = 1)) (Branch 0 0 (Leaf 3 2) (
         ]) : term
   Reason: Can't unify term to term -> thm (Incompatible types)
```

```
Fail "Static Errors" raised

> patriciaLib.PTREE_EVERY_LEAF_CONV ``EVERY_LEAF (\x y. x < 2) (Branch 0 0 (Leaf 3
Exception- Type error in function application.
   Function: patriciaLib.PTREE_EVERY_LEAF_CONV : conv -> conv
   Argument:
      (
      Parse.Term
      [
         QUOTE
         " (*#loc 1 37*)EVERY_LEAF (\\x y. x < 2) (Branch 0 0 (Leaf 3 2) (Leaf 2 1)
         ]) : term
   Reason: Can't unify term to term -> thm (Incompatible types)
Fail "Static Errors" raised
```

See also

patriciaLib.PTREE_CONV

PTREE_EXISTS_LEAF_CONV (patriciaLib)

```
patriciaLib.PTREE_EXISTS_LEAF_CONV : conv
```

Conversion for evaluating applications of patricia$EXISTS_LEAF.

The conversion PTREE_EXISTS_LEAF_CONV evaluates terms of the form EXISTS_LEAF P t where t is a well-formed Patricia tree (constructed by patricia$Empty, patricia$Leaf and patricia$Branch) and P is predicate.

Failure

The conversion will fail if the supplied term is not a suitable application of patricia$EXISTS_LEAF.

Example

```
> patriciaLib.PTREE_EXISTS_LEAF_CONV ``EXISTS_LEAF (=) Empty``;
Exception- Type error in function application.
   Function: patriciaLib.PTREE_EXISTS_LEAF_CONV : conv -> conv
   Argument:
      (Parse.Term [QUOTE " (*#loc 1 38*)EXISTS_LEAF (=) Empty"]) :
      term
   Reason: Can't unify term to term -> thm (Incompatible types)
Fail "Static Errors" raised

> patriciaLib.PTREE_EXISTS_LEAF_CONV ``EXISTS_LEAF (\x y. y = 2) (Branch 0 0 (Leaf 3 2)
Exception- Type error in function application.
   Function: patriciaLib.PTREE_EXISTS_LEAF_CONV : conv -> conv
   Argument:
      (
      Parse.Term
      [
         QUOTE
         " (*#loc 1 38*)EXISTS_LEAF (\\x y. y = 2) (Branch 0 0 (Leaf 3 2) (Leaf 2 1))"
         ]) : term
   Reason: Can't unify term to term -> thm (Incompatible types)
Fail "Static Errors" raised

> patriciaLib.PTREE_EXISTS_LEAF_CONV ``EXISTS_LEAF (\x y. y = 3) (Branch 0 0 (Leaf 3 2)
Exception- Type error in function application.
   Function: patriciaLib.PTREE_EXISTS_LEAF_CONV : conv -> conv
   Argument:
      (
      Parse.Term
      [
         QUOTE
         " (*#loc 1 38*)EXISTS_LEAF (\\x y. y = 3) (Branch 0 0 (Leaf 3 2) (Leaf 2 1))"
         ]) : term
   Reason: Can't unify term to term -> thm (Incompatible types)
Fail "Static Errors" raised
```

See also

patriciaLib.PTREE_CONV

PTREE_IN_PTREE_CONV (patriciaLib)

```
patriciaLib.PTREE_IN_PTREE_CONV : conv
```

Conversion for evaluating applications of `patricia$IN_PTREE`.

The conversion `PTREE_IN_PTREE_CONV` evaluates terms of the form `n IN_PTREE t` where `t` is a well-formed unit Patricia tree (constructed by `patricia$Empty`, `patricia$Leaf` and `patricia$Branch`) and `n` is a natural number literal.

Failure

The conversion will fail if the supplied term is not a suitable application of `patricia$IN_PTREE`.

Example

```
> patriciaLib.PTREE_IN_PTREE_CONV ``1 IN_PTREE Empty``;
Exception- HOL_ERR
  (at Conv.RAND_CONV:
     at Conv.REWR_CONV: lhs of thm doesn't match term) raised

> patriciaLib.PTREE_IN_PTREE_CONV ``3 IN_PTREE (Branch 0 0 (Leaf 3 ()) (Leaf 2 ()))
Exception- HOL_ERR
  (at Conv.RAND_CONV:
     at Conv.REWR_CONV: lhs of thm doesn't match term) raised
```

See also

`patriciaLib.PTREE_CONV`

PTREE_INSERT_PTREE_CONV (patriciaLib)

```
patriciaLib.PTREE_INSERT_PTREE_CONV : conv
```

Conversion for evaluating applications of `patricia$INSERT_PTREE`.

The conversion `PTREE_INSERT_PTREE_CONV` evaluates terms of the form `m INSERT_PTREE_PTREE t` where `t` is a well-formed unit Patricia tree (correctly constructed using `patricia$Empty`, `patricia$Leaf` and `patricia$Branch`) and `m` is a natural number literal.

Failure

The conversion will fail if the supplied term is not a suitable application of `patricia$INSERT_PTREE`.

Example

```
> patriciaLib.PTREE_INSERT_PTREE_CONV ``2 INSERT_PTREE Empty``;
val it = ⊢ <{2}> = Leaf 2 (): thm

> DEPTH_CONV patriciaLib.PTREE_INSERT_PTREE_CONV ``3 INSERT_PTREE 2 INSERT_PTREE Empty``
val it = ⊢ <{3; 2}> = Branch 0 0 (Leaf 3 ()) (Leaf 2 ()): thm
```

See also

`patriciaLib.PTREE_CONV`

PTREE_IS_PTREE_CONV (patriciaLib)

```
patriciaLib.PTREE_IS_PTREE_CONV : conv
```

Conversion for evaluating applications of `patricia$IS_PTREE`.

The conversion `PTREE_IS_PTREE_CONV` evaluates terms of the form `IS_PTREE t` where `t` is any tree constructed by `patricia$Empty`, `patricia$Leaf` and `patricia$Branch`. Well-formed trees correspond with those that can be constructed by `patricia$ADD`.

Failure

The conversion will fail if the supplied term is not a suitable application of patricia$IS_-PTREE.

Example

```
> patriciaLib.PTREE_IS_PTREE_CONV ``IS_PTREE Empty``;
val it = ⊢ IS_PTREE <{}> ⇔ T: thm

> patriciaLib.PTREE_IS_PTREE_CONV ``IS_PTREE (Branch 0 0 (Leaf 3 2) (Leaf 2 1))``;
val it = ⊢ IS_PTREE (Branch 0 0 (Leaf 3 2) (Leaf 2 1)) ⇔ T: thm

> patriciaLib.PTREE_IS_PTREE_CONV ``IS_PTREE (Branch 0 0 (Leaf 3 2) (Leaf 1 1))``;
val it = ⊢ IS_PTREE (Branch 0 0 (Leaf 3 2) (Leaf 1 1)) ⇔ F: thm
```

See also

patriciaLib.PTREE_CONV

PTREE_PEEK_CONV (patriciaLib)

```
patriciaLib.PTREE_PEEK_CONV : conv
```

Conversion for evaluating applications of patricia$PEEK.

The conversion PTREE_PEEK_CONV evaluates terms of the form t ' m where t is a well-formed Patricia tree (constructed by patricia$Empty, patricia$Leaf and patricia$Branch) and m is a natural number literal.

Failure

The conversion will fail if the supplied term is not a suitable application of patricia$PEEK.

Example

```
> patriciaLib.PTREE_PEEK_CONV ``Empty ' 3``;
Exception- HOL_ERR
  (at Conv.RAND_CONV:
     at Conv.REWR_CONV: lhs of thm doesn't match term) raised

> patriciaLib.PTREE_PEEK_CONV ``Branch 0 0 (Leaf 3 2) (Leaf 2 1) ' 3``;
Exception- HOL_ERR
  (at Conv.RAND_CONV:
     at Conv.REWR_CONV: lhs of thm doesn't match term) raised
```

See also

`patriciaLib.PTREE_CONV`

PTREE_REMOVE_CONV (patriciaLib)

```
patriciaLib.PTREE_REMOVE_CONV : conv
```

Conversion for evaluating applications of `patricia$REMOVE`.

The conversion `PTREE_REMOVE_CONV` evaluates terms of the form `t \\ m` where `t` is a well-formed Patricia tree (constructed by `patricia$Empty`, `patricia$Leaf` and `patricia$Branch`) and `m` is a natural number literal.

Failure

The conversion will fail if the supplied term is not a suitable application of `patricia$REMOVE`.

Example

```
> patriciaLib.PTREE_REMOVE_CONV ``Empty \\ 3``;
val it = ⊢ <{}> \\ 3 = <{}>: thm
```

```
> patriciaLib.PTREE_REMOVE_CONV ``Branch 0 0 (Leaf 3 2) (Leaf 2 1) \\ 3``;
val it = ⊢ Branch 0 0 (Leaf 3 2) (Leaf 2 1) \\ 3 = Leaf 2 1: thm
```

See also

patriciaLib.PTREE_CONV

PTREE_SIZE_CONV (patriciaLib)

```
patriciaLib.PTREE_SIZE_CONV : conv
```

Conversion for evaluating applications of patricia$SIZE.

The conversion PTREE_SIZE_CONV evaluates terms of the form SIZE t where t is a well-formed Patricia tree (constructed by patricia$Empty, patricia$Leaf and patricia$Branch).

Failure

The conversion will fail if the supplied term is not a suitable application of patricia$SIZE.

Example

```
> patriciaLib.PTREE_SIZE_CONV ``SIZE Empty``;
val it = ⊢ SIZE <{}> = 0: thm

> patriciaLib.PTREE_SIZE_CONV ``SIZE (Branch 0 0 (Leaf 3 2) (Leaf 2 1))``;
val it = ⊢ SIZE (Branch 0 0 (Leaf 3 2) (Leaf 2 1)) = 2: thm
```

See also

patriciaLib.PTREE_CONV

PTREE_TRANSFORM_CONV (patriciaLib)

`patriciaLib.PTREE_TRANSFORM_CONV : conv`

Conversion for evaluating applications of `patricia$TRANSFORM`.

The conversion `PTREE_TRANSFORM_CONV` evaluates terms of the form `TRANSFORM f t` where `t` is a well-formed Patricia tree (constructed by `patricia$Empty`, `patricia$Leaf` and `patricia$Branch`) and `f` is map.

Failure

The conversion will fail if the supplied term is not a suitable application of `patricia$TRANSFORM`.

Example

```
> patriciaLib.PTREE_TRANSFORM_CONV ``TRANSFORM ODD Empty``;
Exception- Type error in function application.
   Function: patriciaLib.PTREE_TRANSFORM_CONV : conv -> conv
   Argument: (Parse.Term [QUOTE " (*#loc 1 36*)TRANSFORM ODD Empty"])
      : term
   Reason: Can't unify term to term -> thm (Incompatible types)
Fail "Static Errors" raised

> patriciaLib.PTREE_TRANSFORM_CONV ``TRANSFORM ODD (Branch 0 0 (Leaf 3 2) (Leaf 2 1))``;
Exception- Type error in function application.
   Function: patriciaLib.PTREE_TRANSFORM_CONV : conv -> conv
   Argument:
      (
      Parse.Term
      [
         QUOTE
         " (*#loc 1 36*)TRANSFORM ODD (Branch 0 0 (Leaf 3 2) (Leaf 2 1))"
         ]) : term
   Reason: Can't unify term to term -> thm (Incompatible types)
Fail "Static Errors" raised
```

See also

patriciaLib.PTREE_CONV

PURE_ASM_REWRITE_RULE (Rewrite)

```
Rewrite.PURE_ASM_REWRITE_RULE : (thm list -> thm -> thm)
```

Rewrites a theorem including the theorem's assumptions as rewrites.

The list of theorems supplied by the user and the assumptions of the object theorem are used to generate a set of rewrites, without adding implicitly the basic tautologies stored under implicit_rewrites. The rule searches for matching subterms in a top-down recursive fashion, stopping only when no more rewrites apply. For a general description of rewriting strategies see GEN_REWRITE_RULE.

Failure

Rewriting with PURE_ASM_REWRITE_RULE does not result in failure. It may diverge, in which case PURE_ONCE_ASM_REWRITE_RULE may be used.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.PURE_ONCE_ASM_REWRITE_RULE

PURE_ASM_REWRITE_TAC (Rewrite)

```
Rewrite.PURE_ASM_REWRITE_TAC : (thm list -> tactic)
```

Rewrites a goal including the goal's assumptions as rewrites.

PURE_ASM_REWRITE_TAC generates a set of rewrites from the supplied theorems and the assumptions of the goal, and applies these in a top-down recursive manner until no match is found. See GEN_REWRITE_TAC for more information on the group of rewriting tactics.

Failure

PURE_ASM_REWRITE_TAC does not fail, but it can diverge in certain situations. For limited depth rewriting, see PURE_ONCE_ASM_REWRITE_TAC. It can also result in an invalid tactic.

To advance or solve a goal when the current assumptions are expected to be useful in reducing the goal.

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.FILTER_ASM_REWRITE_TAC, Rewrite.FILTER_ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_REWRITE_TAC, Rewrite.PURE_ONCE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST_TAC

PURE_CASE_TAC (BasicProvers)

```
BasicProvers.PURE_CASE_TAC : tactic
```

Case splits on a term t that features in the goal as `case t of ...`.

BasicProvers.PURE_CASE_TAC searches the goal for an instance of `case t of ...`, and performs a `BasicProvers.Cases_on `t``.

Failure

BasicProvers.PURE_CASE_TAC fails if there is no instance of `case t of ...` in the goal, where the case term is a case constant in the typebase and all the free variables of t are free in the goal.

See also

BasicProvers.CASE_TAC

PURE_ONCE_ASM_REWRITE_RULE (Rewrite)

```
Rewrite.PURE_ONCE_ASM_REWRITE_RULE : (thm list -> thm -> thm)
```

Rewrites a theorem once, including the theorem's assumptions as rewrites.

PURE_ONCE_ASM_REWRITE_RULE excludes the basic tautologies in implicit_rewrites from the theorems used for rewriting. It searches for matching subterms once only, without recursing over already rewritten subterms. For a general introduction to rewriting tools see GEN_REWRITE_RULE.

Failure

PURE_ONCE_ASM_REWRITE_RULE does not fail and does not diverge.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_ASM_REWRITE_RULE, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_ASM_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.REWRITE_RULE

PURE_ONCE_ASM_REWRITE_TAC (Rewrite)

```
Rewrite.PURE_ONCE_ASM_REWRITE_TAC : (thm list -> tactic)
```

Rewrites a goal once, including the goal's assumptions as rewrites.

A set of rewrites generated from the assumptions of the goal and the supplied theorems is used to rewrite the term part of the goal, making only one pass over the goal. The basic tautologies are not included as rewrite theorems. The order in which the given theorems are applied is an implementation matter and the user should not depend on any ordering. See GEN_REWRITE_TAC for more information on rewriting tactics in general.

Failure

PURE_ONCE_ASM_REWRITE_TAC does not fail and does not diverge.

Manipulation of the goal by rewriting with its assumptions, in instances where rewriting with tautologies and recursive rewriting is undesirable.

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.FILTER_ASM_REWRITE_-TAC, Rewrite.FILTER_ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_-REWRITE_TAC, Rewrite.PURE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST_TAC

PURE_ONCE_REWRITE_CONV (Rewrite)

```
Rewrite.PURE_ONCE_REWRITE_CONV : (thm list -> conv)
```

Rewrites a term once with only the given list of rewrites.

PURE_ONCE_REWRITE_CONV generates rewrites from the list of theorems supplied by the user, without including the tautologies given in implicit_rewrites. The applicable rewrites are employeded once, without entailing in a recursive search for matches over the term. See GEN_REWRITE_CONV for more details about rewriting strategies in HOL.

Failure

This rule does not fail, and it does not diverge.

See also

Rewrite.GEN_REWRITE_CONV, Conv.ONCE_DEPTH_CONV, Rewrite.ONCE_REWRITE_CONV, Rewrite.PURE_REWRITE_CONV, Rewrite.REWRITE_CONV

PURE_ONCE_REWRITE_RULE (Rewrite)

```
Rewrite.PURE_ONCE_REWRITE_RULE : (thm list -> thm -> thm)
```

Rewrites a theorem once with only the given list of rewrites.

PURE_ONCE_REWRITE_RULE generates rewrites from the list of theorems supplied by the user, without including the tautologies given in implicit_rewrites. The applicable rewrites are employeded once, without entailing in a recursive search for matches over the theorem. See GEN_REWRITE_RULE for more details about rewriting strategies in HOL.

Failure

This rule does not fail, and it does not diverge.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.GEN_REWRITE_RULE, Conv.ONCE_DEPTH_CONV, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.REWRITE_RULE

PURE_ONCE_REWRITE_TAC (Rewrite)

```
Rewrite.PURE_ONCE_REWRITE_TAC : (thm list -> tactic)
```

Rewrites a goal using a supplied list of theorems, making one rewriting pass over the goal.

PURE_ONCE_REWRITE_TAC generates a set of rewrites from the given list of theorems, and applies them at every match found through searching once over the term part of the goal, without recursing. It does not include the basic tautologies as rewrite theorems. The order in which the rewrites are applied is unspecified. For more information on rewriting tactics see GEN_REWRITE_TAC.

Failure

PURE_ONCE_REWRITE_TAC does not fail and does not diverge.

This tactic is useful when the built-in tautologies are not required as rewrite equations and recursive rewriting is not desired.

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.FILTER_ASM_REWRITE_TAC, Rewrite.FILTER_ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_

REWRITE_TAC, Rewrite.PURE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_ASM_REWRITE_-TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST_TAC

PURE_REWRITE_CONV (Rewrite)

```
Rewrite.PURE_REWRITE_CONV : (thm list -> conv)
```

Rewrites a term with only the given list of rewrites.

This conversion provides a method for rewriting a term with the theorems given, and excluding simplification with tautologies in `implicit_rewrites`. Matching subterms are found recursively, until no more matches are found. For more details on rewriting see GEN_REWRITE_CONV.

PURE_REWRITE_CONV is useful when the simplifications that arise by rewriting a theorem with `implicit_rewrites` are not wanted.

Failure

Does not fail. May result in divergence, in which case PURE_ONCE_REWRITE_CONV can be used.

See also

Rewrite.GEN_REWRITE_CONV, Rewrite.ONCE_REWRITE_CONV, Rewrite.PURE_ONCE_REWRITE_-CONV, Rewrite.REWRITE_CONV

PURE_REWRITE_RULE (Rewrite)

```
Rewrite.PURE_REWRITE_RULE : (thm list -> thm -> thm)
```

Rewrites a theorem with only the given list of rewrites.

This rule provides a method for rewriting a theorem with the theorems given, and excluding simplification with tautologies in `implicit_rewrites`. Matching subterms are

found recursively starting from the term in the conclusion part of the theorem, until no more matches are found. For more details on rewriting see GEN_REWRITE_RULE.

PURE_REWRITE_RULE is useful when the simplifications that arise by rewriting a theorem with implicit_rewrites are not wanted.

Failure

Does not fail. May result in divergence, in which case PURE_ONCE_REWRITE_RULE can be used.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_ASM_REWRITE_RULE, Rewrite.PURE_ONCE_ASM_REWRITE_RULE, Rewrite.PURE_ONCE_REWRITE_RULE, Rewrite.REWRITE_RULE

PURE_REWRITE_TAC (Rewrite)

```
Rewrite.PURE_REWRITE_TAC : (thm list -> tactic)
```

Rewrites a goal with only the given list of rewrites.

PURE_REWRITE_TAC behaves in the same way as REWRITE_TAC, but without the effects of the built-in tautologies. The order in which the given theorems are applied is an implementation matter and the user should not depend on any ordering. For more information on rewriting strategies see GEN_REWRITE_TAC.

Failure

PURE_REWRITE_TAC does not fail, but it can diverge in certain situations; in such cases PURE_ONCE_REWRITE_TAC may be used.

This tactic is useful when the built-in tautologies are not required as rewrite equations. It is sometimes useful in making more time-efficient replacements according to equations for which it is clear that no extra reduction via tautology will be needed. (The difference in efficiency is only apparent, however, in quite large examples.)

PURE_REWRITE_TAC advances goals but solves them less frequently than REWRITE_TAC; to be precise, PURE_REWRITE_TAC only solves goals which are rewritten to "T" (i.e. TRUTH) without recourse to any other tautologies.

Example

It might be necessary, say for subsequent application of an induction hypothesis, to resist reducing a term b = T to b.

```
- PURE_REWRITE_TAC [] ([], Term `b = T`);
> val it = ([([], `b = T`)], fn)
    : (term list * term) list * (thm list -> thm)

- REWRITE_TAC [] ([], Term `b = T`);
> val it = ([([], `b`)], fn)
    : (term list * term) list * (thm list -> thm)
```

See also

Rewrite.ASM_REWRITE_TAC, Rewrite.FILTER_ASM_REWRITE_TAC, Rewrite.FILTER_ONCE_ASM_REWRITE_TAC, Rewrite.GEN_REWRITE_TAC, Rewrite.ONCE_ASM_REWRITE_TAC, Rewrite.ONCE_REWRITE_TAC, Rewrite.PURE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_ASM_REWRITE_TAC, Rewrite.PURE_ONCE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST_TAC

pure_ss (pureSimps)

pureSimps.pure_ss : simpset

A simpset containing only the conditional rewrite generator and no additional rewrites.

This simpset sits at the root of the simpset hierarchy. It contains no rewrites, congruences, conversions or decision procedures. Instead it contains just the code which converts theorems passed to it as context into (possibly conditional) rewrites.

Simplification with pure_ss is analogous to rewriting with PURE_REWRITE_TAC and others. The only difference is that the theorems passed to SIMP_TAC pure_ss are interpreted as

conditional rewrite rules. Though the pure_ss can't take advantage of extra contextual information garnered through congruences, it can still discharge side conditions. (This is demonstrated in the examples below.)

Failure

Can't fail, as it is not a functional value.

Example

The theorem ADD_EQ_SUB from arithmeticTheory states that

```
|- !m n p. n <= p ==> ((m + n = p) = m = p - n)
```

We can use this result to make progress with the following goal in conjunction with pure_ss in a way that no form of REWRITE_TAC could:

```
- ASM_SIMP_TAC pure_ss [ADD_EQ_SUB] ([“x <= y”], “z + x = y”);
> val it = ([([`x <= y`], `z = y - x`)], fn) : tactic_result
```

This example illustrates the way in which the simplifier can do conditional rewriting. However, the lack of the congruence for implications means that using pure_ss will not be able to discharge the side condition in the goal below:

```
- SIMP_TAC pure_ss [ADD_EQ_SUB] ([], “x <= y ==> (z + x = y)”);
> val it = ([([], `x <= y ==> (z + x = y)`)], fn) : tactic_result
```

As bool_ss has the relevant congruence included, it does make progress in the same situation:

```
- SIMP_TAC bool_ss [ADD_EQ_SUB] ([], “x <= y ==> (z + x = y)”);
> val it = ([([], `x <= y ==> (z = y - x)`)], fn) : tactic_result
```

The pure_ss simpset might be used in the most delicate simplification situations, or, mimicking the way it is used within the distribution itself, as a basis for the construction of other simpsets.

Comments

There is also a pureSimps.PURE_ss ssfrag value. Its usefulness is questionable.

See also

boolSimps.bool_ss, Rewrite.PURE_REWRITE_TAC, simpLib.SIMP_CONV, simpLib.SIMP_TAC

pvariant (PairRules)

```
PairRules.pvariant : (term list -> term -> term)
```

Modifies variable and constant names in a paired structure to avoid clashes.

When applied to a list of (possibly paired structures of) variables to avoid clashing with, and a pair to modify, `pvariant` returns a variant of the pair. That is, it changes the names of variables and constants in the pair as intuitively as possible to make them distinct from any variables in the list, or any (non-hidden) constants. This is normally done by adding primes to the names.

The exact form of the altered names should not be relied on, except that the original variables will be unmodified unless they are in the list to avoid clashing with. Also note that if the same variable occurs more that one in the pair, then each instance of the variable will be modified in the same way.

Failure

`pvariant l p` fails if any term in the list `l` is not a paired structure of variables, or if `p` is not a paired structure of variables and constants.

Example

The following shows a case that exhibits most possible behaviours:

```
- pvariant [Term `b:'a`, Term `(c:'a,c':'a)`]
           (Term `((a:'a,b:'a),(c:'a,b':'a,T,b:'a))`);
val it = `(a,b''),c'',b',T',b''` : term
```

The function `pvariant` is extremely useful for complicated derived rules which need to rename pairs variable to avoid free variable capture while still making the role of the pair obvious to the user.

See also

Term.variant, Term.genvar

Q_TAC (Tactical)

```
Tactical.Q_TAC : (term -> tactic) -> term quotation -> tactic
```

A tactical that parses in the context of a goal, a la the Q library.

When applied to a term tactic T and a quotation q, the tactic Q_TAC T q first parses the quotation q in the context of the goal to yield the term tm, and then applies the tactic T tm to the goal.

Failure

The application of Q_TAC to a term tactic T and a quotation q never fails. The resulting composite tactic Q_TAC T q fails when applied to a goal if either q cannot be parsed, or T tm fails when applied to the goal.

Comments

Useful for avoiding decorating terms with type abbreviations.

See also

Tactical.EVERY, Tactical.FIRST, Tactical.ORELSE, Tactical.THEN, Tactical.THEN1, Tactical.THENL

QCHANGED_CONSEQ_CONV (ConseqConv)

```
ConseqConv.QCHANGED_CONSEQ_CONV : conseq_conv -> conseq_conv
```

Makes a consequence conversion fail if applying it raises the UNCHANGED exception.

See also

Conv.QCHANGED_CONV, ConseqConv.CHANGED_CONSEQ_CONV

QCHANGED_CONV (Conv)

```
Conv.QCHANGED_CONV : conv -> conv
```

Makes a conversion fail if applying it raises the UNCHANGED exception.

If c is a conversion that maps a term t to a theorem |- t = t', then so too is QCHANGED_CONV c. If c applied to t raises the special UNCHANGED exception used by conversions to indicate that they haven't changed an input, then QCHANGED_CONV c will fail, raising a different exception HOL_ERR ... when applied to t.

The purpose of this is that some enclosing functions handle the UNCHANGED exception as though c had succeeded by returning the theorem |- t = t.

This behaviour is similar to that of CHANGED_CONV, except that that conversion also fails if the conversion c returns a theorem when applied to t, and if that theorem has alpha-convertible left and right hand sides.

Failure

QCHANGED_CONV c t fails (other than by raising UNCHANGED) if c applied t raises the UNCHANGED exception, or if c fails otherwise when applied to t.

QCHANGED_CONV can be used in places where CHANGED_CONV is appropriate, and where one knows that the conversion argument will not return an instance of reflexivity, or if one does not mind this occurring and not being trapped. Because it is no more than an exception handler, QCHANGED_CONV is very efficient.

See also

Conv.UNCHANGED, Conv.CHANGED_CONV

QCONV (Conv)

```
Conv.QCONV : conv -> conv
```

Stops a conversion raising the `UNCHANGED` exception.

If conversion `c` applied to term `t` raises the `UNCHANGED` exception, then `QCONV c t` instead returns the theorem `|- t = t`.

Failure

`QCONV c t` fails if the application of `c` to `t` fails.

See also

`Conv.UNCHANGED`, `Conv.CHANGED_CONV`, `Conv.QCHANGED_CONV`

QI_ss (bossLib)

```
bossLib.QI_ss : ssfrag
```

Simpset-fragment for instantiating quantifiers with some default heuristics.

`QI_ss` is short for `QUANT_INST_ss [std_qp]`.

See also

`bossLib.QI_TAC`, `quantHeuristicsLib.QUANT_INST_ss`

QI_TAC (bossLib)

```
bossLib.QI_TAC : tactic
```

Try to instantiate quantifiers with some default heuristics.

`QI_TAC` is short for `QUANT_INSTANTIATE_TAC [std_qp]`.

See also

`bossLib.ASM_QI_TAC`, `quantHeuristicsLib.QUANT_INSTANTIATE_TAC`, `quantHeuristicsLib.ASM_QUANT_INSTANTIATE_TAC`

quadruple (Lib)

```
Lib.quadruple : 'a -> 'b -> 'c -> 'd -> 'a * 'b * 'c * 'd
```

Makes four values into a quadruple.

`quadruple x1 x2 x3 x4` returns `(x1, x2, x3, x4)`.

Failure

Never fails.

See also

`Lib.quadruple_of_list`, `Lib.pair`, `Lib.triple`

quadruple_of_list (Lib)

```
Lib.quadruple_of_list : 'a list -> 'a * 'a * 'a * 'a
```

Turns a four-element list into a quadruple.

`quadruple_of_list [x1, x2, x3, x4]` returns `(x1, x2, x3, x4)`.

Failure

Fails if applied to a list that is not of length 4.

See also

`Lib.singleton_of_list`, `Lib.pair_of_list`, `Lib.triple_of_list`

QUANT_CONSEQ_CONV (ConseqConv)

```
ConseqConv.QUANT_CONSEQ_CONV : (conseq_conv -> conseq_conv)
```

Applies a consequence conversion to the body of an existentially or universally quantified term.

See also

`Conv.QUANT_CONV`, `ConseqConv.FORALL_CONSEQ_CONV`, `ConseqConv.EXISTS_CONSEQ_CONV`

QUANT_CONV (Conv)

```
Conv.QUANT_CONV : conv -> conv
```

Applies a conversion underneath a quantifier.

If `conv N` returns `A |- N = P`, then `QUANT_CONV conv (M (\v.N))` returns `A |- M (\v.N) = M (\v.P)`.

Failure

If `conv N` fails, or if `v` is free in `A`.

Example

```
> QUANT_CONV SYM_CONV (Term `!x. x + 0 = x`);
val it = ⊢ (∀x. x + 0 = x) ⇔ ∀x. x = x + 0: thm
```

Comments

For deeply nested quantifiers, `STRIP_QUANT_CONV` and `STRIP_BINDER_CONV` are more efficient than iterated application of `QUANT_CONV`, `BINDER_CONV`, or `ABS_CONV`.

See also

`Conv.BINDER_CONV`, `Conv.STRIP_QUANT_CONV`, `Conv.STRIP_BINDER_CONV`, `Conv.ABS_CONV`

QUANT_INST_ss (quantHeuristicsLib)

```
quantHeuristicsLib.QUANT_INST_ss : quant_param list -> simpLib.ssfrag
```

A simpset fragement corresponding to `QUANT_INSTANTIATE_CONV`.

See also

`quantHeuristicsLib.QUANT_INSTANTIATE_CONV`

QUANT_INSTANTIATE_CONV(quantHeuristicsLib)

```
quantHeuristicsLib.QUANT_INSTANTIATE_CONV : quant_param list -> conv
```

Instantiate quantifiers in a term using a given list of quantifier heuristic parameters.

This conversion tries to instantiate quantifiers. Therefore, it uses the given list of quantifier heuristic parameters. If the list is empty, it knows about the usual Boolean Connectives, quantifiers and equations. The parameter `quantHeuristicsArgsLib.std_qp` adds

knowledge about option-types, pairs, lists, records and natural numbers. The stateful parameter quantHeuristicsArgsLib.Type_Base_qp can be used to extract information about user defined datatypes.

Example

```
> quantHeuristicsLib.QUANT_INSTANTIATE_CONV [] ``?x. ((x=7) \/ (7 = x)) /\ P x``;
val it = ⊢ (∃x. (x = 7 ∨ 7 = x) ∧ P x) ⇔ P 7: thm

> quantHeuristicsLib.QUANT_INSTANTIATE_CONV [] ``?x. !y. (x=7) /\ P x y``;
val it = ⊢ (∃x. ∀y. x = 7 ∧ P x y) ⇔ ∀y. P 7 y: thm

> quantHeuristicsLib.QUANT_INSTANTIATE_CONV [] ``?x. (f(8 + 2) = f(x + 2)) /\ P(f (
val it = ⊢ (∃x. f (8 + 2) = f (x + 2) ∧ P (f (x + 2))) ⇔ P (f (8 + 2)): thm

> quantHeuristicsLib.QUANT_INSTANTIATE_CONV [quantHeuristicsLibParameters.std_qp] `
val it = ⊢ (∀x. IS_SOME x ⇒ P x) ⇔ ∀x_x'. P (SOME x_x'): thm

> quantHeuristicsLib.QUANT_INSTANTIATE_CONV [quantHeuristicsLibParameters.std_qp] `
val it = ⊢ (∀l. l ≠ [] ⇒ LENGTH l > 0) ⇔ ∀l_h l_t. LENGTH (l_h::l_t) > 0: thm
```

See also

quantHeuristicsLib.QUANT_INST_ss, quantHeuristicsLib.QUANT_INSTANTIATE_TAC, quantHeuristicsLib.ASM_QUANT_INSTANTIATE_TAC, quantHeuristicsLib.FAST_QUANT_INSTANTIATE_CONV, quantHeuristicsLib.FAST_QUANT_INST_ss, quantHeuristicsLib.FAST_QUANT_INSTANTIATE_TAC

QUANT_INSTANTIATE_TAC (quantHeuristicsLib)

```
quantHeuristicsLib.QUANT_INSTANTIATE_TAC : quant_param list -> tactic
```

A tactic to instantiate quantifiers in a term using a given list of quantifier heuristic parameters.

This tactic is based on `quantHeuristicsLib.QUANT_INSTANTIATE_CONV`. It tries to instantiate quantifiers. Free variables of the goal are seen as universally quantified by this tactic. Therefore, it tries to instantiate these free variables. In contrast to `quantHeuristicsLib.ASM_QUANT_INSTANTIATE_TAC` this tactic does not take the assumptions of the goal into account.

See also

`quantHeuristicsLib.QUANT_INSTANTIATE_CONV`, `quantHeuristicsLib.ASM_QUANT_INSTANTIATE_TAC`

QUANT_TAC (quantHeuristicsLib)

```
quantHeuristicsLib.QUANT_TAC : (string * Parse.term Lib.frag list * Parse.term Parse.fra
```

A tactic to instantiate quantifiers in a term using an explitly given list of (partial) instantiations.

This tactic can be seen as a generalisation of `Q.EXISTS_TAC`. When applied to a term fragment `u` and a goal `?x. t`, the tactic `EXISTS_TAC` reduces the goal to `t[u/x]`. `QUANT_TAC` allows to perform similar instantiations of quantifiers at subpositions, provided the subposition occurs in a formula composed of standard operators that the tactic can handle. It can - depending on negation level - instantiate both existential and universal quantifiers. Moreover, it allows partial instantiations and instantiating multiple variables at the same time.

`QUANT_TAC` gets a list of triples `(var_name, instantiation, free_vars)` as an argument. `var_name` is the name of the variable to be instantiated; `instantiation` is the term this variable should be instantiated with. Finally, `free_vars` is a list of free variables in `instantiation` that should remain quantified.

As this tactic adresses variables by their name, resulting proofs might not be robust. Therefore, this tactic should be used carefully.

Example

Given the goal

```
!x. (!z. P x z) ==> ?a b.    Q a        b z
```

where z and a are natural numbers, the call `QUANT_TAC [("z", `0`, []), ("a", `SUC a'`, [`a'`])]` instantiates z with 0 and a with `SUC a'`, where `a'` is free. The variable z is universally quantified, but in the antecedent of the implication. Therefore, it can be safely instantiated. a is existentially quantified. In this example we just want to say that a is not 0, therefore `a'` is considered as a free variable and thus remains existentially quantified. The call results in the goal

```
!x. (    P x 0) ==> ?  b a'. Q (SUC a') b z
```

See also

`Tactic.EXISTS_TAC`

quote (Lib)

```
Lib.quote : string -> string
```

Put quotation marks around a string.

An application `quote s` is equal to `"\"" ^ s ^ "\""`. This is often useful when printing messages.

Failure

Never fails

Example

```
> "foo";
val it = "foo": string

> quote "foo";
val it = "val it = foo ; \n": string
```

See also

`Lib.mlquote`

r (proofManagerLib)

```
proofManagerLib.r : int -> unit
```

Reorders the subgoals on top of the subgoal package goal stack.

The function `r` is part of the subgoal package. The name `rotate` may also be used to access the same function. For a general description of the subgoal package, see `set_goal`.

The `r` function's basic step of operation is to take the first element of the current list of sub-goals and move it to the end of the same list. The numeric argument passed to `r` specifies how many times this operation is to be performed.

Failure

Raises the `NO_PROOFS` exception if there is no current proof manipulated by the subgoal package. Raises a `HOL_ERR` if the current goal state only has one sub-goal, or if the argument passed to `r` is negative.

Interactively attacking subgoals in a different order to that generated by the subgoal package.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

Raise (Feedback)

```
Feedback.Raise : exn -> 'a
```

Print an exception before re-raising it.

The `Raise` function prints out information about its argument exception before re-raising it. It uses the value of `ERR_to_string` to format the message, and prints the information on the `outstream` held in `ERR_outstream`.

Failure

Never fails, since it always succeeds in raising the supplied exception.

Example

```
> Raise (mk_HOL_ERR "Foo" "bar" "incomprehensible input");
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.

Exception raised at Foo.bar: incomprehensible input
HOL_ERR (at Foo.bar: incomprehensible input) raised
```

See also

`Feedback`, `Feedback.ERR_to_string`, `Feedback.ERR_outstream`, `Lib.try`, `Lib.trye`

rand (Term)

```
Term.rand : term -> term
```

Returns the operand from a combination (function application).

If `M` is a combination, i.e., has the form `(t1 t2)`, then `rand M` returns `t2`.

Failure

Fails if `M` is not a combination.

See also

`Term.rator`, `Term.dest_comb`

RAND_CONV (Conv)

```
Conv.RAND_CONV : (conv -> conv)
```

Applies a conversion to the operand of an application.

If `c` is a conversion that maps a term `"t2"` to the theorem `|- t2 = t2'`, then the conversion `RAND_CONV c` maps applications of the form `"t1 t2"` to theorems of the form:

```
|- (t1 t2) = (t1 t2')
```

That is, `RAND_CONV c "t1 t2"` applies `c` to the operand of the application `"t1 t2"`.

Failure

`RAND_CONV c tm` fails if `tm` is not an application or if `tm` has the form `"t1 t2"` but the conversion `c` fails when applied to the term `t2`. The function returned by `RAND_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

Example

```
> RAND_CONV numLib.num_CONV (Term `SUC 2`);
val it = ⊢ SUC 2 = SUC (SUC 1): thm
```

See also

`Conv.ABS_CONV`, `Conv.BINOP_CONV`, `Conv.LAND_CONV`, `Conv.RATOR_CONV`, `Conv.SUB_CONV`, `Conv.RHS_CONV`

random_tnn (mlTreeNeuralNetwork)

```
mlTreeNeuralNetwork.random_tnn : (term * int list) list -> tnn
```

Creates a random tree neural network (TNN) with the precised dimensions for each neural network operators.

To create an initial TNN, the user first needs to gather all operators (constants or variables) appearing in the examples. Then, given an embedding dimension `d`, for each operator `f` with arity `a` the list of dimensions of is to be defined as `[a x d,u1,...,uk,d]`. The natural numbers `u1,...,uk` are sizes of the intermediate layers that can be freely chosen by the user. In the case of a head operator `h`, the input dimension is to be `d` and the output dimension is to be the length of the objective `l`.

Failure

Fails if the list of dimensions is empty.

Example

```
> val tnn =
   mlTreeNeuralNetwork.random_tnn [(``h: bool -> bool``,[4,10,1]),(``$~``,[4,8,4]),
val tnn = <Redblackmap(3)>: mlTreeNeuralNetwork.tnn
```

Comments

Precising a list of dimensions of length 1 results in an unusable empty neural network for this operator.

See also

`mlTreeNeuralNetwork.train_tnn`

rator (Term)

```
Term.rator : term -> term
```

Returns the operator from a combination (function application).

If `M` is a combination, i.e., has the form `(t1 t2)`, then `rator M` returns `t1`.

Failure

Fails if `M` is not a combination.

See also

`Term.rand`, `Term.dest_comb`

RATOR_CONV (Conv)

```
Conv.RATOR_CONV : (conv -> conv)
```

Applies a conversion to the operator of an application.

If `c` is a conversion that maps a term `"t1"` to the theorem `|- t1 = t1'`, then the conversion `RATOR_CONV c` maps applications of the form `"t1 t2"` to theorems of the form:

```
|- (t1 t2) = (t1' t2)
```

That is, `RATOR_CONV c "t1 t2"` applies `c` to the operand of the application `"t1 t2"`.

Failure

`RATOR_CONV c tm` fails if `tm` is not an application or if `tm` has the form `"t1 t2"` but the conversion `c` fails when applied to the term `t1`. The function returned by `RATOR_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

Example

```
> RATOR_CONV BETA_CONV (Term `(\x y. x + y) 1 2`);
val it = ⊢ (λx y. x + y) 1 2 = (λy. 1 + y) 2: thm
```

See also

Conv.ABS_CONV, Conv.RAND_CONV, Conv.SUB_CONV

raw_match (Term)

```
Term.raw_match :
  hol_type list -> term set ->
  term -> term ->
  (term,term) subst * (hol_type,hol_type) subst ->
    ((term,term) subst * term set) *
    ((hol_type,hol_type) subst * hol_type list)
```

Primitive term matcher.

The most primitive matching algorithm for HOL terms is raw_match. An invocation raw_match avoid_tys avoid_tms pat ob (tmS,tyS), if it succeeds, returns a substitution pair ((TmS,TmID),(TyS,TyID)) such that

```
aconv (subst TmS' (inst TyS pat)) ob.
```

where TmS' is TmS instantiated by TyS. The arguments avoid_tys and avoid_tms specify type and term variables in pat that are not allowed to become redexes in S and T.

The pair (tmS,tyS) is an accumulator argument. This allows raw_match to be folded through lists of terms to be matched. (TmS,TyS) must agree with (tmS,tyS). This means that if there is a {redex,residue} in TmS and also a {redex,residue} in tmS so that both redex fields are equal, then the residue fields must be alpha-convertible. Similarly for types: if there is a {redex,residue} in TyS and also a {redex,residue} in tyS so that both redex fields are equal, then the residue fields must also be equal. If these conditions hold, then the result-pair (TmS,TyS) includes (tmS,tyS).

Finally, note that the result also includes a set (resp. a list) of term and type variables, accompanying the substitutions. These represent identity bindings that have occurred in

the process of doing the match. If `raw_match` is to be folded across multiple problems, these output values will need to be merged with `avoid_tms` and `avoid_tys` respectively on the next call so that they cannot be instantiated a second time. Because they are identity bindings, they do not need to be referred to in validating the central identity above.

Failure

`raw_match` will fail if no `TmS` and `TyS` meeting the above requirements can be found. If a match `(TmS,TyS)` between `pat` and `ob` can be found, but elements of `avoid_tys` would appear as redexes in `TyS` or elements of `avoid_tms` would appear as redexes in `TmS`, then `raw_match` will also fail.

Example

We first perform a match that requires type instantitations, and also alpha-convertibility.

```
> val ((tmS,_),(tyS,_)) =
   raw_match [] empty_varset
             “\x:'a. x = f (y:'b)”
             “\a.    a = ~p” ([],[]);
val tmS = [{redex = “y”, residue = “p”}, {redex = “f”, residue = “$¬”}]:
   (term, term) Term.subst
val tyS =
   [{redex = “:β”, residue = “:bool”}, {redex = “:α”, residue = “:bool”}]:
   (hol_type, hol_type) Term.subst
```

One of the main differences between `raw_match` and more refined derivatives of it, is that the returned substitutions are un-normalized by `raw_match`. If one naively applied `(tmS,tyS)` to `\x:'a. x = f (y:'b)`, type instantiation with `tyS` would be applied first, yielding `\x:bool. x = f (y:bool)`. Then substitution with `tmS` would be applied, unsuccessfully, since both `f` and `y` in the pattern term have been type instantiated, but the corresponding elements of the substitution haven't. Thus, higher level operations building on `raw_match` typically instantiate `tmS` by `tyS` to get `tmS'` before applying `(tmS',tyT)` to the pattern term. This can be achieved by using `norm_subst`. However, `raw_match` exposes this level of detail to the programmer.

Comments

Higher level matchers are generally preferable, but `raw_match` is occasionally useful when programming inference rules.

See also

`Term.match_term`, `Term.match_terml`, `Term.norm_subst`, `Term.subst`, `Term.inst`, `Type.raw_match_type`, `Type.match_type`, `Type.match_typel`, `Type.type_subst`

raw_match_type (Type)

```
Type.raw_match_type
  : hol_type -> hol_type ->
    (hol_type,hol_type) subst * hol_type list ->
    (hol_type,hol_type) subst * hol_type list
```

Primitive type matching algorithm.

An invocation `raw_match_type pat ty (S,Id)` performs matching, just as `match_type`, except that it takes an extra accumulating parameter `(S,Id)`, which represents a 'raw' substitution that the match `(theta,id)` of `pat` and `ty` must be compatible with. If matching is successful, `(theta,id)` is merged with `(S,Id)` to yield the result.

Failure

A call to `raw_match_type pat ty (S,Id)` will fail when `match_type pat ty` would. It will also fail when a `{redex,residue}` calculated in the course of matching `pat` and `ty` is such that there is a `{redex_i,residue_i}` in `S` and `redex` equals `redex_i` but `residue` does not equal `residue_i`.

Example

```
> val res1 = raw_match_type alpha ":'a -> bool" ([],[]);
val res1 = ([{redex = ":α", residue = ":α -> bool"}], []):
   (hol_type, hol_type) Lib.subst * hol_type list
```

```
> raw_match_type ":'a -> 'b -> 'c"
                 ":('a -> bool) -> 'b -> ind" res1;
val it =
   ([{redex = ":γ", residue = ":ind"},
     {redex = ":α", residue = ":α -> bool"}], [":β"]):
   (hol_type, hol_type) Lib.subst * hol_type list
```

Comments

Probably exposes too much internal state of the matching algorithm.

See also

`Type.match_type`

rawterm_pp (Parse)

```
Parse.rawterm_pp : ('a -> 'b) -> 'a -> 'b
```

Causes a function to use the raw terminal backend when pretty-printing.

Functions that pretty-print HOL types, terms and theorems do so through an abstraction called a "backend". Using these backends allows output to be customised to the facilities provided by different display devices. For example, on terminals supporting DEC's vt100 colour coding, free variables are displayed in blue. There is also a "raw terminal" backend, that doesn't change the output in any way.

When an interactive session begins, HOL links all of the pretty-printing functions to a backend value stored in a reference, `Parse.current_backend`. Of course, this reference can be changed as a user desires. A call to `rawterm_pp f` function wraps a call to `Lib.with_flag`, setting the current backend to be the raw terminal value for the duration of the `f`'s application to its (first) argument.

Failure

A call to `rawterm_pp f` never fails. A call to `rawterm_pp f x` should only fail if `f x` would fail, but this ultimately depends on `f`'s implementation.

Example

In a vt100-compatible terminal, capturing the output of pp_term reveals a stream of horrible-looking escape codes:

```
> ppstring pp_term ``p /\ q``;
val it = "\^[[0;1;34mp\^[[0m /\\ \^[[0;1;34mq\^[[0m": string
```

If this string is to be print-ed to the vt100, it will colour the p and q a pleasant blue colour. If, on the other hand, the string is to be output to a file, the colouring is probably not desirable. Then one can use rawterm_pp to get the unadorned characters of the output:

```
> rawterm_pp (ppstring pp_term) ``p /\ q``;
val it = "p /\\ q": string
```

This last usage is so common that it is already available in the library as term_to_string.

Comments

If a function f is curried with multiple arguments, say f x y, then care will probably be needed with modifying it with rawterm_pp. In particular, rawterm_pp f x y is likely not to work, while rawterm_pp (f x) y probably will.

See also

Lib.ppstring, Parse.term_to_string, Lib.with_flag

rd (proofManagerLib)

```
proofManagerLib.rd : unit -> proof
```

Restores the proof state, redoing the effects of a previous expansion.

The function rd is part of the subgoal package. It is an abbreviation for the function redo. For a description of the subgoal package, see set_goal.

Failure

As for `redo`.

Back tracking in a goal-directed proof to undo errors or try different tactics.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.b`, `proofManagerLib.backu`
`proofManagerLib.rd`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`,
`proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`,
`proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

read (Tag)

```
Tag.read : string -> tag
```

Make a tag suitable for use by `mk_oracle_thm`.

In order to construct a tag usable by `mk_oracle_thm`, one uses `read`, which takes a string and makes it into a tag.

Failure

The string must be an alphanumeric, i.e., start with an alphabetic character and thereafter consist only of alphabetic or numeric characters.

Example

```
> Tag.read "Shamboozled";
val it = [oracles: #] [axioms: ]: tag
```

See also

`Thm.mk_oracle_thm`, `Thm.tag`

recInduct (bossLib)

```
bossLib.recInduct : thm -> tactic
```

Performs recursion induction.

An invocation `recInduct thm` on a goal `g`, where `thm` is typically an induction scheme returned from an invocation of `Define` or `Hol_defn`, attempts to match the consequent of `thm` to `g` and, if successful, then replaces `g` by the instantiated antecedents of `thm`. The order of quantification of the goal should correspond with the order of quantification in the conclusion of `thm`.

Failure

`recInduct` fails if the goal is not universally quantified in a way corresponding with the quantification of the conclusion of `thm`.

Example

Suppose we had introduced a function for incrementing a number until it no longer can be found in a given list:

```
variant x L = if MEM x L then variant (x + 1) L else x
```

Typically `Hol_defn` would be used to make such a definition, and some subsequent proof would be required to establish termination. Once that work was done, the specified recursion equations would be available as a theorem and, as well, a corresponding induction theorem would also be generated. In the case of `variant`, the induction theorem `variant_ind` is

```
|- !P. (!x L. (MEM x L ==> P (x + 1) L) ==> P x L) ==> !v v1. P v v1
```

Suppose now that we wish to prove that the variant with respect to a list is not in the list:

```
?- !x L. ~MEM (variant x L) L`,
```

One could try mathematical induction, but that won't work well, since `x` gets incremented in recursive calls. Instead, induction with '`variant`-induction' works much better. `recInduct` can be used to apply such theorems in tactic proof. For our example, `recInduct variant_ind` yields the goal

```
?- !x L. (MEM x L ==> ~MEM (variant (x + 1) L) L) ==> ~MEM (variant x L) L
```

A few simple tactic applications then prove this goal.

See also

`bossLib.Induct`, `bossLib.Induct_on`, `bossLib.completeInduct_on`, `bossLib.measureInduct_on`, `Prim_rec.INDUCT_THEN`, `bossLib.Cases`, `bossLib.Hol_datatype`, `proofManagerLib.g`, `proofManagerLib.e`

RED_CONV (reduceLib)

```
reduceLib.RED_CONV : conv
```

Performs arithmetic or boolean reduction at top level if possible.

The conversion `RED_CONV` attempts to apply, at the top level only, one of the following conversions from the `reduce` library (only one can succeed):

```
ADD_CONV  AND_CONV  BEQ_CONV  COND_CONV
DIV_CONV  EXP_CONV   GE_CONV    GT_CONV
IMP_CONV   LE_CONV   LT_CONV   MOD_CONV
MUL_CONV  NEQ_CONV  NOT_CONV    OR_CONV
PRE_CONV  SBC_CONV  SUC_CONV
```

Failure

Fails if none of the above conversions are applicable at top level.

Example

```
#RED_CONV "(2=3) = F";;
|- ((2 = 3) = F) = ~(2 = 3)

#RED_CONV "15 DIV 13";;
|- 15 DIV 13 = 1

#RED_CONV "100 + 100";;
|- 100 + 100 = 200

#RED_CONV "0 + x";;
evaluation failed     RED_CONV
```

See also

reduceLib.REDUCE_CONV, reduceLib.REDUCE_RULE, reduceLib.REDUCE_TAC

REDEPTH_CONSEQ_CONV (ConseqConv)

```
ConseqConv.REDEPTH_CONSEQ_CONV : directed_conseq_conv -> directed_conseq_conv
```

Similar to DEPTH_CONSEQ_CONV, but revisits modified subterms.

See also

ConseqConv.DEPTH_CONSEQ_CONV

REDEPTH_CONV (Conv)

```
Conv.REDEPTH_CONV : (conv -> conv)
```

Applies a conversion bottom-up to all subterms, retraversing changed ones.

`REDEPTH_CONV c tm` applies the conversion `c` repeatedly to all subterms of the term `tm` and recursively applies `REDEPTH_CONV c` to each subterm at which `c` succeeds, until there is no subterm remaining for which application of `c` succeeds.

More precisely, `REDEPTH_CONV c tm` repeatedly applies the conversion `c` to all the subterms of the term `tm`, including the term `tm` itself. The supplied conversion `c` is applied to the subterms of `tm` in bottom-up order and is applied repeatedly (zero or more times, as is done by `REPEATC`) to each subterm until it fails. If `c` is successfully applied at least once to a subterm, `t` say, then the term into which `t` is transformed is retraversed by applying `REDEPTH_CONV c` to it.

Failure

`REDEPTH_CONV c tm` never fails but can diverge if the conversion `c` can be applied repeatedly to some subterm of `tm` without failing.

Example

The following example shows how `REDEPTH_CONV` retraverses subterms:

```
- REDEPTH_CONV BETA_CONV (Term `(\f x. (f x) + 1) (\y.y) 2`);
val it = |- (\f x. (f x) + 1)(\y. y)2 = 2 + 1 : thm
```

Here, `BETA_CONV` is first applied successfully to the (beta-redex) subterm:

```
(\f x. (f x) + 1) (\y.y)
```

This application reduces this subterm to:

```
(\x. ((\y.y) x) + 1)
```

`REDEPTH_CONV BETA_CONV` is then recursively applied to this transformed subterm, eventually reducing it to `(\x. x + 1)`. Finally, a beta-reduction of the top-level term, now the simplified beta-redex `(\x. x + 1) 2`, produces `2 + 1`.

Comments

The implementation of this function uses failure to avoid rebuilding unchanged subterms. That is to say, during execution the exception `QConv.UNCHANGED` may be generated and later trapped. The behaviour of the function is dependent on this use of failure. So, if the conversion given as an argument happens to generate the same exception, the operation of `REDEPTH_CONV` will be unpredictable.

See also

`Conv.DEPTH_CONV`, `Conv.ONCE_DEPTH_CONV`, `Conv.TOP_DEPTH_CONV`

redo (proofManagerLib)

```
proofManagerLib.redo : unit -> proof
```

Restores the proof state, redoing the effects of a previous expansion.

The function `redo` is part of the subgoal package. It may be abbreviated by the function `rd`. It undoes the action of `backup`, returning to a state after an undone state change (caused by calls to `expand`, `rotate` and similar functions). The function that caused the original state change is not re-run; only the final state is restored. Any regular state change will cause the redo stack to be discarded.

Failure

The function `redo` will fail if the redo list is empty.

Example

```
> g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;
val it =
   Proof manager status: 4 proofs.
   4. Completed goalstack:  [.] ⊢ q

   3. Completed goalstack: ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]
```

```
   2. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

   1. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

> e CONJ_TAC;
OK..
2 subgoals:
val it =

   TL [1; 2; 3] = [2; 3]

   HD [1; 2; 3] = 1

> backup();
val it =
   Initial goal:

   HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]: proof

> redo();
val it =

   TL [1; 2; 3] = [2; 3]

   HD [1; 2; 3] = 1
```

Back tracking in a goal-directed proof to undo errors or try different tactics.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.b`, `proofManagerLib.backu`
`proofManagerLib.rd`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`,
`proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`,
`proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

REDUCE_CONV (reduceLib)

```
reduceLib.REDUCE_CONV : conv
```

Performs arithmetic or boolean reduction at all levels possible.

The conversion `REDUCE_CONV` attempts to apply, in bottom-up order to all suitable redexes, arithmetic computation conversions for all of the standard operators from `arithmeticTheory`. The conversions are implemented as rewrites applied by `CBV_CONV`. In particular, it is designed to prove the appropriate reduction for an arbitrarily complicated expression constructed from numerals, those operators, and the boolean constants `T` and `F`, and will do this to all such sub-expressions within a term.

Failure

Never fails, but may give a reflexive equation.

Example

```
> reduceLib.REDUCE_CONV “(2=3) = F”;
val it = ⊢ (2 = 3 ⇔ F) ⇔ T: thm

> reduceLib.REDUCE_CONV “if 100 < 200 then 2 EXP (8 DIV 2)
                         else 3 EXP ((26 EXP 0) * 3)”;
val it = ⊢ (if 100 < 200 then 2 ** (8 DIV 2) else 3 ** (26 ** 0 * 3)) = 16:
   thm

> reduceLib.REDUCE_CONV “(15 = 16) \/ (15 < 16)”;
val it = ⊢ 15 = 16 ∨ 15 < 16 ⇔ T: thm

> reduceLib.REDUCE_CONV “1 + x”;
val it = ⊢ 1 + x = 1 + x: thm

> reduceLib.REDUCE_CONV “!x:num. x = x”;
val it = ⊢ (∀x. x = x) ⇔ ∀x. T: thm
```

Comments

This entry-point is also available as `numLib.REDUCE_CONV`.

See also

`computeLib.CBV_CONV`, `reduceLib.RED_CONV`, `reduceLib.REDUCE_RULE`, `reduceLib.REDUCE_TAC`

REDUCE_RULE (reduceLib)

```
reduceLib.REDUCE_RULE : thm -> thm
```

Performs arithmetic or boolean reduction on a theorem at all levels possible.

`REDUCE_RULE` attempts to transform a theorem by applying `REDUCE_CONV`.

Failure

Never fails, but may just return the original theorem.

Example

```
> reduceLib.REDUCE_RULE (ASSUME “x = 100 + (60 - 17)”);
val it =  [.] ⊢ x = 143: thm

> reduceLib.REDUCE_RULE (REFL “100 + 12 DIV 6”);
val it = ⊢ T: thm
```

See also

`reduceLib.RED_CONV`, `reduceLib.REDUCE_CONV`, `reduceLib.REDUCE_TAC`

REDUCE_TAC (reduceLib)

```
reduceLib.REDUCE_TAC : tactic
```

Performs arithmetic or boolean reduction on a goal at all levels possible.

REDUCE_TAC attempts to transform a goal by applying REDUCE_CONV. It will prove any true goal which is constructed from numerals and the boolean constants T and F.

Failure

Never fails, but may not advance the goal.

Example

The following example takes a couple of minutes' CPU time:

```
> g ‘((1 EXP 3) + (12 EXP 3) = 1729) /\ ((9 EXP 3) + (10 EXP 3) = 1729)’;

> e reduceLib.REDUCE_TAC;;
OK..
val it =
   Initial goal proved
   ⊢ 1 EXP 3 + 12 EXP 3 = 1729 ∧ 9 EXP 3 + 10 EXP 3 = 1729 : proof
```

See also

reduceLib.RED_CONV, reduceLib.REDUCE_CONV, reduceLib.REDUCE_RULE

REFINE_EXISTS_TAC (Q)

```
Q.REFINE_EXISTS_TAC : term quotation -> tactic
```

Attacks existential goals, making the existential variable more concrete.

The tactic Q.REFINE_EXISTS_TAC q parses the quotation q in the context of the (necessarily existential) goal to which it is applied, and uses the resulting term as the witness for the goal. However, if the witness has any variables not already present in the goal, then these are treated as new existentially quantified variables. If there are no such “free” variables, then the behaviour is the same as EXISTS_TAC.

Failure

Fails if the goal is not existential, or if the quotation can not parse to a term of the same type as the existentially quantified variable.

Example

If the quotation doesn't mention any new variables:

```
- Q.REFINE_EXISTS_TAC `n` ([``n > x``], ``?m. m > x``);
> val it =
    ([([``n > x``], ``n > x``)], fn)
    : (term list * term) list * (thm list -> thm)
```

If the quotation does mention any new variables, they are existentially quantified in the new goal:

```
- Q.REFINE_EXISTS_TAC `n + 2` ([``~P 0``], ``?p. P (p - 1)``);
> val it =
    ([([``~P 0``], ``?n. P (n + 2 - 1)``)], fn)
    : (term list * term) list * (thm list -> thm)
```

`Q.REFINE_EXISTS_TAC` is useful if it is clear that an existential goal will be solved by a term of particular form, while it is not yet clear precisely what term this will be. Further proof activity should be able to exploit the additional structure that has appeared in the place of the existential variable.

See also

`Q.LIST_REFINE_EXISTS_TAC`, `Tactic.EXISTS_TAC`

REFL (Thm)

`Thm.REFL : conv`

Returns theorem expressing reflexivity of equality.

`REFL` maps any term `t` to the corresponding theorem `|- t = t`.

Failure

Never fails.

See also

`Conv.ALL_CONV`, `Tactic.REFL_TAC`

REFL_CONSEQ_CONV (ConseqConv)

```
ConseqConv.REFL_CONSEQ_CONV : conseq_conv
```

Given a term `t` of type `bool` this consequence conversion returns the theorem `|- t ==> t`.

See also

`ConseqConv.TRUE_CONSEQ_CONV`, `ConseqConv.FALSE_CONSEQ_CONV`, `ConseqConv.TRUE_FALSE_REFL_CONSEQ_CONV`

REFL_TAC (Tactic)

```
Tactic.REFL_TAC : tactic
```

Solves a goal which is an equation between alpha-equivalent terms.

When applied to a goal `A ?- t = t'`, where `t` and `t'` are alpha-equivalent, `REFL_TAC` completely solves it.

```
  A ?- t = t'
============  REFL_TAC
```

Failure

Fails unless the goal is an equation between alpha-equivalent terms.

See also

`Tactic.ACCEPT_TAC`, `Tactic.MATCH_ACCEPT_TAC`, `Rewrite.REWRITE_TAC`

register_btrace (Feedback)

```
Feedback.register_btrace : string * bool ref -> unit
```

Registers a trace variable for a boolean reference.

A call to `register_btrace(nm, bref)` registers a trace variable called `nm` that can take on two different values (0 and 1), which correspond to the state of the boolean variable `bref`.

Failure

Fails if the given name is already in use as a trace variable.

Comments

This function uses `register_ftrace` to make a boolean variable appear as an integer value.

See also

`Feedback`, `Feedback.current_trace`, `Feedback.register_trace`, `Feedback.register_ftrace`, `Feedback.set_trace`, `Feedback.trace`, `Feedback.traces`

register_frag (simpLib)

```
simpLib.register_frag : ssfrag -> ssfrag
```

Registers a simpset fragment for later use with SF.

A call to simpLib.register_frag sfrag records a mapping from the name of sfrag to the sfrag value. This internal database is then used by simplification tactics when they see theorems created with calls to SF.

Failure

Fails is the fragment sfrag is anonymous.

See also

bossLib.SF

register_ftrace (Feedback)

```
Feedback.register_ftrace :
  (string * ((unit -> int) * (int -> unit)) * int) -> unit
```

Registers a trace that is accessed by a set/get pair of functions.

A call to register_ftrace(nm, (g,s), m) registers an integer-valued trace variable that is updated with the s function and whose value is read with the g function. The variable is given the name nm and the variable's maximum allowed value is m. The trace's default is the value of g(), which is called just once as part of the registration procedure.

Failure

Fails if the given name is already in use as a trace variable, or if the maximum or the default value (returned by g()) is less than zero.

Comments

The two functions provide a more general way of accessing something that may not be actually be an integer reference, even though this is the interface that the various trace functions present.

See also

Feedback, Feedback.current_trace, Feedback.register_trace, Feedback.register_btrace, Feedback.set_trace, Feedback.trace, Feedback.traces

register_trace (Feedback)

```
Feedback.register_trace : (string * int ref * int) -> unit
```

Registers a new tracing variable.

A call to `register_trace(n, r, m)` registers the integer reference variable `r` as a tracing variable associated with name `n`. The integer `m` is its maximum value. Its value at the time of registration is considered its default value, which will be restored by a call to `reset_trace n` or `reset_traces`.

Failure

Fails if there is already a tracing variable registered under the name given, or if either the maximum value or the value in the reference is less than zero.

See also

Feedback, Feedback.register_btrace, Feedback.register_ftrace, Feedback.reset_trace, Feedback.reset_traces, Feedback.trace, Feedback.traces

release (Globals)

```
Globals.release : string
```

The name of the release series of the HOL system being run.

Example

```
> Globals.release;
val it = "Trindemossen": string
```

See also

`Globals.version`

remove_external_wspace (Portable)

```
Portable.remove_external_wspace : string -> string
```

Removes trailing and leading whitespace characters from a string

A call to `remove_external_wspace s` returns a string identical to `s` except that all leading and trailing characters for which `Char.isSpace` is true have been removed. The implementation is (with the Basis's `Substring` structure open):

```
string (dropl Char.isSpace (dropr Char.isSpace (full s)))
```

Failure

Never fails.

See also

`Portable.remove_wspace`

remove_ovl_mapping (Parse)

```
Parse.remove_ovl_mapping: string -> {Name:string,Thy:string} -> unit
```

Removes an overloading mapping between the string and constant specified.

Each grammar maintains two maps internally. One is from strings to non-empty lists of terms, and the other is from terms to strings. The first map is used to resolve overloading when parsing. A string will eventually be turned into one of the terms in the list that it maps to. When printing a constant, the map in the opposite direction is used to turn a term into a string.

A call to `remove_ovl_mapping s {Name,Thy}`, maps the `Name-Thy` record to a constant `c`, and removes the `c-s` pair from both maps.

Failure

Never fails. If the given pair is not in either map, the function silently does nothing.

To prune the overloading maps of unwanted possibilities.

Comments

Note that removing a print-mapping for a constant will result in that constant always printing fully qualified as `thy$name`. This situation will persist until that constant is given a name to map to (either with `overload_on` or `update_overload_maps`).

As with other parsing functions, there is a sister function, `temp_remove_ovl_mapping` that does the same thing, but whose effect is not saved to a theory file.

See also

`Parse.clear_overloads_on`, `Parse.overload_on`, `Parse.update_overload_maps`

remove_rules_for_term (Parse)

```
Parse.remove_rules_for_term : string -> unit
```

Removes parsing/pretty-printing rules from the global grammar.

Calling `remove_rules_for_term s` removes all those rules (if any) in the global grammar that are for the term `s`. The string specifies the name of the term that the rule is for, not a token that may happen to be used in concrete syntax for the term.

Failure

Never fails.

Example

The universal quantifier can have its special binder status removed using this function:

```
- val t = Term`!x. P x /\ ~Q x`;
<<HOL message: inventing new type variable names: 'a.>>
> val t = `!x. P x /\ ~Q x` : term
- remove_rules_for_term "!";
> val it = () : unit
- t;
> val it = `! (\x. P x /\ ~Q x)` : term
```

Similarly, one can remove the two rules for conditional expressions and see the raw syntax as follows:

```
- val t = Term`if p then q else r`;
<<HOL message: inventing new type variable names: 'a.>>
> val t = `if p then q else r` : term
- remove_rules_for_term "COND";
> val it = () : unit
- t;
> val it = `COND p q r` : term
```

Comments

There is a companion `temp_remove_rules_for_term` function, which has the same effect on the global grammar, but which does not cause this effect to persist when the current theory is exported.

See also

`Parse.remove_termtok`

remove_ssfrags (simpLib)

```
simpLib.remove_ssfrags : string list -> simpset -> simpset
```

Removes named simpset fragments from a simpset.

A call to `remove_ssfrags fragnames simpset` returns a simpset that is the same as `simpset` except that the simpset fragments with names in the list `fragnames` are no longer included. As a special case, the empty name `""` matches all unnamed fragments within `simpset` and causes them to be removed.

Failure

If no member of `fragnames` names a fragment in `simpset`, the `Conv.UNCHANGED` exception is raised.

Example

```
> SIMP_CONV (srw_ss()) [] “MAP ($+ 1) [3;4;5]”;
val it = ⊢ MAP ($+ 1) [3; 4; 5] = [1 + 3; 1 + 4; 1 + 5]: thm

> val myss = simpLib.remove_ssfrags ["REDUCE"] (srw_ss());
Exception- UNCHANGED raised

> SIMP_CONV myss [] “MAP ($+ 1) [3;4;5]”;
Exception- Value or constructor (myss) has not been declared
Fail "Static Errors" raised
```

Comparison with `exclude_ssfrags`

`remove_ssfrags` only strips matching fragments from the simpset's history; a subsequent `ss ++ FRAG_ss` would re-add them. By contrast, `exclude_ssfrags` also records the exclusion in the simpset itself, so that subsequent `++` of a fragment with one of the named names is a silent no-op (until cleared via `force_add` / `SF`). `exclude_ssfrags` also never raises `Conv.UNCHANGED`. The `Proof[exclude_frags = ...]` attribute uses `exclude_ssfrags`.

See also

simpLib.exclude_ssfrags, simpLib.force_add, BasicProvers.diminish_srw_ss

remove_termtok (Parse)

```
Parse.remove_termtok : {term_name : string, tok : string} -> unit
```

Removes a rule from the global grammar.

The remove_termtok removes parsing/printing rules from the global grammar. Rules to be removed are those that are for the term with the given name (term_name) and which include the string tok as part of their concrete representation. If multiple rules satisfy this criterion, they are all removed. If none match, the grammar is not changed.

Failure

Never fails.

Example

If one wished to revert to the traditional HOL syntax for conditional expressions, this would be achievable as follows:

```
- remove_termtok {term_name = "COND", tok = "if"};
> val it = () : unit

- Term`if p then q else r`;
<<HOL message: inventing new type variable names: 'a, 'b, 'c, 'd, 'e, 'f.>>
> val it = `if p then q else r` : term

- Term`p => q | r`;
<<HOL message: inventing new type variable names: 'a.>>
> val it = `COND p q r` : term
```

The first invocation of the parser above demonstrates that once the rule for the if-then-else syntax has been removed, a string that used to parse as a conditional

expression then parses as a big function application (the function `if` applied to five arguments).

The fact that the pretty-printer does not print the term using the old-style syntax, even after the `if-then-else` rule has been removed, is due to the fact that the corresponding rule in the grammar does not have its preferred flag set. This can be accomplished with `prefer_form_with_tok` as follows:

```
- prefer_form_with_tok {term_name = "COND", tok = "=>"};
> val it = () : unit

- Term`p => q | r`;
<<HOL message: inventing new type variable names: 'a.>>
> val it = `p => q | r` : term
```

Used to modify the global parsing/pretty-printing grammar by removing a rule, possibly as a prelude to adding another rule which would otherwise clash.

Comments

As with other functions in the `Parse` structure, there is a companion `temp_remove_termtok` function, which has the same effect on the global grammar, but which does not cause this effect to persist when the current theory is exported.

The specification of a rule by `term_name` and one of its tokens is not perfect, but seems adequate in practice.

See also

`Parse.remove_rules_for_term`, `Parse.prefer_form_with_tok`

remove_type_abbrev (Parse)

```
Parse.remove_type_abbrev : string -> unit
```

Remove a type abbreviation from the type grammar.

A call to `remove_type_abbrev s` removes any type abbreviations keyed on string `s`. As with other functions affecting the global grammar, there is a companion function, `temp_remove_type_abbrev`, which affects the grammar but does not cause the effect to be replayed in descendant theories.

If the string `s` is not a qualified name (of the form `"thy$name"`), then all type abbreviations with base name `s` are removed. If `s` does have a qualified name, then only a type abbreviation of that name and theory will be removed (if such exists).

Failure

Fails if the given string is a malformed qualified identifier (e.g., `foo$$`). If the given name is syntactically valid, but there are no abbreviations keyed to the given name, a call to `remove_type_abbrev` will silently do nothing.

Example

The standard theory context (where `pred_set` is loaded), includes an abbreviation mapping ``` ``:'a set`` ``` to ``` ``:'a -> bool`` ```. It doesn't print the abbreviated form back to the user, because its printing has been disabled with `disable_tyabbrev_printing`.

```
> ``:'a set``;
val it = ``:'a -> bool`` : hol_type

> remove_type_abbrev "set";
val it = (): unit

> ``:'a set``;
Exception- HOL_ERR ...
```

See also

`Parse.disable_tyabbrev_printing`, `Parse.type_abbrev`

remove_user_printer (Parse)

```
Parse.remove_user_printer : string * term -> unit
```

Removes a user-defined pretty-printing function associated with a particular name and term-pattern.

This removes the user-defined pretty-printing function that has been associated with a particular name (the name of the code for the function) and pattern (a term).

Failure

Never fails. If there is no user-printing function in the grammar associated with the provided key, the function has no effect.

Comments

As always, there is an accompanying function `temp_remove_user_printer`, which does not affect the grammar exported to disk.

See also

`Parse.add_user_printer`

remove_word_printer (wordsLib)

```
wordsLib.remove_word_printer : unit -> unit
```

Turns off custom pretty-printing for word literals.

The function `remove_word_printer` calls `Parse.remove_user_printer` to remove pretty-printing for ground instances of “n2w n”. This will normally mean that words output in decimal format.

Example

```
> load "wordsLib";
val it = (): unit
> “0x10000000w”;
val it = “268435456w”: term
> wordsLib.remove_word_printer();
```

```
val it = (): unit
> “0x10000000w”;
val it = “0x10000000w”: term
```

See also

Parse.remove_user_printer, wordsLib.output_words_as, wordsLib.output_words_as_dec, wordsLib.output_words_as_bin, wordsLib.output_words_as_oct, wordsLib.output_words_as_hex

remove_wspace (Portable)

```
Portable.remove_wspace : string -> string
```

Removes all whitespace characters from a string

A call to remove_wspace s returns a string identical to s except that all of the characters for which Char.isSpace is true have been removed. The implementation is

```
String.translate (fn c => if Char.isSpace c then "" else str c) s
```

Failure

Never fails.

See also

Portable.remove_external_wspace

RENAME1_TAC (Q)

```
Q.RENAME1_TAC : term quotation -> tactic
```

Finds an instance of a pattern in a goal, and renames to use names from the pattern.

The tactic `Q.RENAME1_TAC q` is defined to be

```
MATCH_RENAME_TAC q ORELSE MATCH_ASSUM_RENAME_TAC q ORELSE
MATCH_GOALSUB_RENAME_TAC q ORELSE MATCH_ASMSUB_RENAME_TAC q
```

Failure

Fails if all of the constituent tactics fail.

Comments

This tactic can be used to force a particular set of names on a goal, hopefully making the resulting tactic more robust in the face of underlying implementation changes. Note though that successful use of this tactic requires that the "new" names in the provided pattern really be fresh for the goal. If one is really uncertain about what names might be appearing in a goal, this condition may be difficult to ensure, particularly as the tactic only looks for one instance of the pattern at a time (but see `Q.RENAME_TAC`).

This tactic is also available under the alias `bossLib.rename1`.

See also

`Q.MATCH_ASMSUB_RENAME_TAC`, `Q.MATCH_ASSUM_RENAME_TAC`, `Q.MATCH_GOALSUB_RENAME_TAC`, `Q.MATCH_RENAME_TAC`

rename_bvar (Term)

```
Term.rename_bvar : string -> term -> term
```

Performs one step of alpha conversion.

If `M` is a lambda abstraction, i.e., has the form `\v.N`, an invocation `rename_bvar s M` performs one step of alpha conversion to obtain `\s. N[s/v]`.

Failure

If `M` is not a lambda abstraction.

Example

```
> rename_bvar "x" (Term `\v. v ==> w`);
val it = “λx. x ⇒ w”: term

> rename_bvar "x" (Term `\y. y /\ x`);
val it = “λx'. x' ∧ x”: term
```

Comments

`rename_bvar` takes constant time in the current implementation.

See also

`Term.aconv`, `Drule.ALPHA_CONV`

RENAME_TAC (Q)

```
Q.RENAME_TAC : term quotation list -> tactic
```

Renames free variables or subterms within a goal.

A call to `RENAME_TAC qs` searches the current goal for matches to the patterns in the list `qs`, and then substitutes out the matches (in the “reverse direction”) so that the goal becomes one that uses the names from `qs`. This can cause subterms within the goal to turn into simple variables, but the usual intention is to rename free variables into the variables that appear in the patterns.

The matching is done without reference to the goal’s existing free variables. If a variable in `qs` clashes with an existing variable in the goal, then the goal’s variable will be renamed away. It is sufficient for variables to have the same name to “clash”; they need not also have the same type. The search for matches begins by attempting to find matches against the whole of the goal, against whole assumptions, for sub-terms within the goal, and then sub-terms of assumptions.

These four different flavours of searching can additionally be controlled by adding comments `(* ... *)` to the end of the relevant quotation. The string in the comment has its whitespace stripped, and is then split into fields using the pipe character `|` as

a separator. If the string consisting of the single character `g` is present, the pattern is checked against the entirety of the goal's conclusion; the string of character `a` causes a check against each individual assumption; the strings `sg` and `sa` cause checks against sub-terms present in the conclusion, or in any assumption respectively.

If multiple matches are possible, a variant tactic, `Q.kRENAME_TAC`, can be used: this tactic takes an additional "continuation" tactic argument that can be used to discriminate between these matches.

Patterns can use underscores to match anything without any change in the goal's naming being introduced. Underscores can match against sub-terms that include bound variables. Matching is first-order.

Failure

Fails if it is impossible to consistently match the combination of patterns in the provided list of quotations (`qs`).

Example

If the goal is of the form

```
x < y, y < z ?- x < f a
```

then invoking `Q.RENAME_TAC [`b < c`, `a < b`]` will produce the sub-goal:

```
a < b, b < c ?- a < f a'
```

where the goal's original `a` variable (which is not even of type `num`), has been renamed away from `a` because that variable occurs in the patterns. (If the right hand side of the inequality was simply `a` and was thus of type num, it would also be renamed to `a'`.)

If `Q.RENAME_TAC [`b < c`]` is invoked on the same initial goal, the result is:

```
b < y, y < z ?- b < c
```

illustrating the way in which variables can eliminate more complicated sub-terms.

The useful way in which underscores in patterns can be used to "dodge" terms including bound variables is illustrated in the next example, where the initial goal is:

```
(!a. f a < z), z < c ?- z < d
```

After applying `Q.RENAME_TAC [`_ < x`, `x < c`]`, the goal becomes

```
(!a. f a < x), x < c' ?- x < c
```

The goal was chosen for the match to the second pattern because the goal is considered first. If the initial goal had been

```
(!a. f a < z), z < c ?- z < d /\ P z
```

then the result of the same application would be

```
(!a. f a < z), z < c ?- x < d /\ P x
```

because whole assumptions are considered before sub-terms of the goal.

The pattern-specification comments can be important when selecting for certain assumptions. Here, `Q.RENAME_TAC [‘x < _ (* a *)’]` will fail because there is no assumption of the required shape, even though the shape appears as a subterm in the assumptions, and is the shape of the goal’s conclusion:

```
~(a < b) ?- c < d
```

Similarly, using the comment `(* a|sg *)` would also fail because the pattern is not present as a sub-term of the goal’s conclusion.

Comments

This function is also available under the name `bossLib.rename`.

Note that `Q.RENAME_TAC [q]` is *not* the same as `Q.RENAME1_TAC q`. The latter pays attention to the goal’s free variables, using these to constrain the match to the pattern. In contrast, `Q.RENAME_TAC` completely ignores all of the goal’s free variables, such that using an existing name in a pattern doesn’t make any difference to the matching behaviour.

See also

`Q.RENAME1_TAC`

RENAME_VARS_CONV (Conv)

```
Conv.RENAME_VARS_CONV : string list -> conv
```

Renames variables underneath a binder.

`RENAME_VARS_CONV` takes a list of strings specifying new names for variables under a binder. More precisely, it will rename variables in abstractions, or bound by universal, existential, unique existence or the select (or Hilbert-choice) "quantifier".

More than one variable can be renamed at once. If variables occur past the first, then the renaming continues on the appropriate sub-term of the first. (That is, if the term is an abstraction, then renaming will continue on the body of the abstraction. If it is one of the supported quantifiers, then renaming will continue on the body of the abstraction that is the argument of the "binder constant".)

If `RENAME_VARS_CONV` is passed the empty list, it is equivalent to `ALL_CONV`. The binders do not need to be of the same type all the way into the term.

Failure

Fails if an attempt is made to rename a variable in a term that is not an abstraction, or is not one of the accepted quantifiers. Also fails if all of the names in the list are not distinct.

Example

```
> RENAME_VARS_CONV ["a", "b"] ``\x y. x /\ y``;
val it = ⊢ (λx y. x ∧ y) = (λa b. a ∧ b): thm
> RENAME_VARS_CONV ["a", "b"] ``!x:'a y. P x /\ P y``;
val it = ⊢ (∀x y. P x ∧ P y) ⇔ ∀a b. P a ∧ P b: thm
> RENAME_VARS_CONV ["a", "b"] ``!x:'a. ?y. P x /\ P y``;
val it = ⊢ (∀x. ∃y. P x ∧ P y) ⇔ ∀a. ∃b. P a ∧ P b: thm
```

Post-processing mangling of names in code implementing derived logical procedures to make names look more appropriate. Changing names can only affect the presentation of terms, not their semantics.

See also

Term.aconv, Thm.ALPHA

render_exn (Feedback)

```
Feedback.render_exn : exn -> 'a
```

Print HOL_ERR exception then recover, or not, according to Globals.interactive.

The variable Globals.interactive is used by programs to tell whether the HOL4 system is running interactively (i.e. is in the Read-Eval-Print loop) or not (is running in batch mode under Holmake). When the contents of Globals.interactive is true render_exn raises the given exception. If the exception is not otherwise dealt with in user code, the REPL will handle it and print the message contents before resuming the top-level loop.

When the contents of Globals.interactive is false, render_exn prints a message derived from the contents of its argument exception then exits to the host operating system.

Example

```
> val () = render_exn (mk_HOL_ERR "S" "f" "-");
Exception- HOL_ERR (at S.f: -) raised

> Globals.interactive := false; (* for example purposes only! *)
val it = (): unit

> val () = render_exn (mk_HOL_ERR "S" "f" "-");
Exception raised at S.f: -

Process HOL exited abnormally with code 1
```

Comment

render_exn attempts to display non-HOL_ERR exceptions sensibly.

See also

`Feedback`, `Feedback.HOL_ERR`, `Globals.interactive`

REORDER_ANTS (Drule)

```
Drule.REORDER_ANTS : (term list -> term list) -> thm -> thm
```

Strips universal quantifiers and antecedents of implications and reorders the antecedents

```
|- !x. a1 ==> !y. a2 ==> !z. a3 ==> !u. t
---------------------------------------- REORDER_ANTS rev
    |- a3 ==> a2 ==> a1 ==> t
```

Failure

No failure. Can leave the supplied theorem unchanged.

But a choice of `f` other than reordering a list of terms will give a result with assumptions remaining or superfluous antecedents

Comments

For simplicity, doesn't try to reinsert quantifiers in appropriate places. If required, apply GEN_ALL to the resulting theorem.

See also

`Drule.REORDER_ANTS_MOD`, `Drule.SPEC_ALL`, `Drule.GEN_ALL`, `Thm.UNDISCH`, `Drule.DISCH`

REORDER_ANTS_MOD (Drule)

```
Drule.REORDER_ANTS_MOD : (term list -> term list) -> (thm -> thm) -> thm -> thm
```

Strips universal quantifiers and antecedents of implications, modifies the conclusion, and reorders the antecedents

`REORDER_ANTS_MOD f g` combines the effects of `REORDER_ANTS_MOD f` and applies the function `g` to the ultimate consequent of the theorem, as does `underAIs`.

Failure

Fails if `g` fails when applied to the consequent

See also

`Drule.DISCH`, `Drule.GEN_ALL`, `Drule.REORDER_ANTS`, `Drule.SPEC_ALL`, `Drule.underAIs`, `Thm.UNDISCH`

repeat (Lib)

```
Lib.repeat : ('a -> 'a) -> 'a -> 'a
```

Iteratively apply a function until it fails.

An invocation `repeat f x` expands to `repeat f (f x)`. Thus it unrolls to `f(...(f x)...)`, returning the most recent argument to `f` before application fails.

Failure

The evaluation of `repeat f x` fails only if interrupted, or machine resources are exhausted.

Example

The following gives a simple-minded way of calculating the largest integer on the machine.

```
> fun incr x = x+1;
val incr = fn: int -> int
```

(Caution: in some ML implementations, the type `int` is not implemented by machine words, but by 'bignum' techniques that allow numbers of arbitrary size, in which case the example above will not return for a very long time.)

See also

`Lib.funpow`

REPEAT (Tactical)

```
Tactical.REPEAT : (tactic -> tactic)
```

Repeatedly applies a tactic until it fails.

The tactic `REPEAT T` is a tactic which applies `T` to a goal, and while it succeeds, continues applying it to all subgoals generated.

Failure

The application of `REPEAT` to a tactic never fails, and neither does the composite tactic, even if the basic tactic fails immediately.

See also

`Tactical.EVERY`, `Tactical.FIRST`, `Tactical.ORELSE`, `Tactical.THEN`, `Tactical.THENL`

REPEAT_GTCL (Thm_cont)

```
Thm_cont.REPEAT_GTCL : (thm_tactical -> thm_tactical)
```

Applies a theorem-tactical until it fails when applied to a goal.

When applied to a theorem-tactical, a theorem-tactic, a theorem and a goal:

```
REPEAT_GTCL ttl ttac th goal
```

REPEAT_GTCL repeatedly modifies the theorem according to ttl till the result of handing it to ttac and applying it to the goal fails (this may be no times at all).

Failure

Fails iff the theorem-tactic fails immediately when applied to the theorem and the goal.

Example

The following tactic matches th's antecedents against the assumptions of the goal until it can do so no longer, then puts the resolvents onto the assumption list:

```
REPEAT_GTCL (IMP_RES_THEN ASSUME_TAC) th
```

See also

Thm_cont.REPEAT_TCL, Thm_cont.THEN_TCL

REPEAT_LT (Tactical)

```
Tactical.REPEAT_LT : (list_tactic -> list_tactic)
```

Repeatedly applies a list-tactic until it fails.

The list-tactic REPEAT_LT ltac is a list-tactic which applies ltac to a goal list, and while it succeeds, continues applying it to the resulting subgoal list.

Failure

The application of REPEAT_LT to a list-tactic never fails, and neither does the composite list-tactic, even if the basic list-tactic fails immediately.

See also

Tactical.REPEAT, Tactical.THEN_LT

REPEAT_TCL (Thm_cont)

```
Thm_cont.REPEAT_TCL : (thm_tactical -> thm_tactical)
```

Repeatedly applies a theorem-tactical until it fails when applied to the theorem.

When applied to a theorem-tactical, a theorem-tactic and a theorem:

```
REPEAT_TCL ttl ttac th
```

`REPEAT_TCL` repeatedly modifies the theorem according to `ttl` until it fails when given to the theorem-tactic `ttac`.

Failure

Fails iff the theorem-tactic fails immediately when applied to the theorem.

Example

It is often desirable to repeat the action of basic theorem-tactics. For example `CHOOSE_THEN` strips off a single existential quantification, so one might use `REPEAT_TCL CHOOSE_THEN` to get rid of them all.

Alternatively, one might want to repeatedly break apart a theorem which is a nested conjunction and apply the same theorem-tactic to each conjunct. For example the following goal:

```
?- ((0 = w) /\ (0 = x)) /\ (0 = y) /\ (0 = z) ==> (w + x + y + z = 0)
```

might be solved by

```
DISCH_THEN (REPEAT_TCL CONJUNCTS_THEN (SUBST1_TAC o SYM)) THEN
REWRITE_TAC[ADD_CLAUSES]
```

See also

`Thm_cont.REPEAT_GTCL`, `Thm_cont.THEN_TCL`

REPEATC (Conv)

```
Conv.REPEATC : conv -> conv
```

Repeatedly apply a conversion (zero or more times) until it fails.

If `c` is a conversion effects a transformation of a term `t` to a term `t'`, that is if `c` maps `t` to the theorem `|- t = t`, then `REPEATC c` is the conversion that repeats this transformation as often as possible. More exactly, if `c` maps the term ``ti`` to `|- ti=t(i+1)` for `i` from `1` to `n`, but fails when applied to the `n+1`th term ``t(n+1)``, then `REPEATC c ``t1`` returns `|- t1 = t(n+1)`. And if `c ``t`` fails, them `REPEATC c ``t`` returns `|- t = t`.

Further, if `c ``t`` raises the `UNCHANGED` exception, then `REPEATC c ``t`` also raises the same exception (rather than go into an infinite loop).

Failure

Never fails, but can diverge if the supplied conversion never fails.

REPLICATE_CONV (listLib)

```
listLib.REPLICATE_CONV : conv
```

Computes by inference the result of replicating an element a given number of times to form a list.

For an arbitrary expression `x` and numeral constant `n`, the result of evaluating

```
REPLICATE_CONV “REPLICATE n x”
```

is the theorem

```
|- REPLICATE n x = [x;x;...;x]
```

where the list `[x;x;...;x]` is of length `n`.

Example

Evaluating `REPLICATE_CONV "REPLICATE 3 [0;1;2;3]"` will return the following theorem:

```
|- REPLICATE 3 [0;1;2;3] = [[0;1;2;3]; [0;1;2;3]; [0;1;2;3]]
```

Failure

`REPLICATE_CONV tm` fails if `tm` is not of the form described above.

RES_CANON (Drule)

```
Drule.RES_CANON : (thm -> thm list)
```

Put an implication into canonical form for resolution.

All the HOL resolution tactics (e.g. `IMP_RES_TAC`) work by using modus ponens to draw consequences from an implicative theorem and the assumptions of the goal. Some of these tactics derive this implication from a theorem supplied explicitly the user (or otherwise from 'outside' the goal) and some obtain it from the assumptions of the goal itself. But in either case, the supplied theorem or assumption is first transformed into a list of implications in 'canonical' form by the function `RES_CANON`.

The theorem argument to `RES_CANON` should be either be an implication (which can be universally quantified) or a theorem from which an implication can be derived using the transformation rules discussed below. Given such a theorem, `RES_CANON` returns a list of implications in canonical form. It is the implications in this resulting list that are used by the various resolution tactics to infer consequences from the assumptions of a goal.

The transformations done by `RES_CANON th` to the theorem `th` are as follows. First, if `th` is a negation `A |- ~t`, this is converted to the implication `A |- t ==> F`. The following inference rules are then applied repeatedly, until no further rule applies. Conjunctions are split into their components and equivalence (boolean equality) is split into implication in both directions:

```
   A |- t1 /\ t2                      A |- t1 = t2
-------------------          ---------------------------------
 A |- t1   A |- t2            A |- t1 ==> t2   A |- t2 ==> t1
```

Conjunctive antecedents are transformed by:

```
              A |- (t1 /\ t2) ==> t
-------------------------------------------------
 A |- t1 ==> (t2 ==> t)     A |- t2 ==> (t1 ==> t)
```

and disjunctive antecedents by:

```
     A |- (t1 \/ t2) ==> t
--------------------------------
 A |- t1 ==> t    A |- t2 ==> t
```

The scope of universal quantifiers is restricted, if possible:

```
 A |- !x. t1 ==> t2
--------------------          [if x is not free in t1]
 A |- t1 ==> !x. t2
```

and existentially-quantified antecedents are eliminated by:

```
   A |- (?x. t1) ==> t2
---------------------------  [x' chosen so as not to be free in t2]
 A |- !x'. t1[x'/x] ==> t2
```

Finally, when no further applications of the above rules are possible, and the theorem is an implication:

```
A |- !x1...xn. t1 ==> t2
```

then the theorem `A u {t1} |- t2` is transformed by a recursive application of `RES_CANON` to get a list of theorems:

```
[A u {t1} |- t21 , ... , A u {t1} |- t2n]
```

and the result of discharging `t1` from these theorems:

```
[A |- !x1...xn. t1 ==> t21 , ... , A |- !x1...xn. t1 ==> t2n]
```

is returned. That is, the transformation rules are recursively applied to the conclusions of all implications.

Failure

RES_CANON th fails if no implication(s) can be derived from th using the transformation rules shown above.

Example

The uniqueness of the remainder k MOD n is expressed in HOL by the built-in theorem MOD_UNIQUE:

```
|- !n k r. (?q. (k = (q * n) + r) /\ r < n) ==> (k MOD n = r)
```

For this theorem, the canonical list of implications returned by RES_CANON is as follows:

```
- RES_CANON MOD_UNIQUE;
> val it =
    [|- !r n q k. (k = q * n + r) ==> r < n ==> (k MOD n = r),
     |- !n r. r < n ==> !q k. (k = q * n + r) ==> (k MOD n = r)] : thm list
```

The existentially-quantified, conjunctive, antecedent has given rise to two implications, and the scope of universal quantifiers has been restricted to the conclusions of the resulting implications wherever possible.

The primary use of RES_CANON is for the (internal) pre-processing phase of the built-in resolution tactics IMP_RES_TAC, IMP_RES_THEN, RES_TAC, and RES_THEN. But the function RES_CANON is also made available at top-level so that users can call it to see the actual form of the implications used for resolution in any particular case.

See also

Tactic.IMP_RES_TAC, Thm_cont.IMP_RES_THEN, Tactic.RES_TAC, Thm_cont.RES_THEN

RES_EXISTS_CONV (res_quanLib)

```
res_quanLib.RES_EXISTS_CONV : conv
```

Converts a restricted existential quantification to a conjunction.

When applied to a term of the form `?x::P. Q[x]`, the conversion `RES_EXISTS_CONV` returns the theorem:

```
|- ?x::P. Q[x] = (?x. x IN P /\ Q[x])
```

which is the underlying semantic representation of the restricted existential quantification.

Failure

Fails if applied to a term not of the form `?x::P. Q`.

See also

`res_quanLib.RES_FORALL_CONV`, `res_quanLib.RESQ_EXISTS_TAC`

RES_EXISTS_UNIQUE_CONV (res_quanLib)

`res_quanLib.RES_EXISTS_UNIQUE_CONV : conv`

Converts a restricted unique existential quantification to a conjunction.

When applied to a term of the form `?!x::P. Q[x]`, the conversion `RES_EXISTS_UNIQUE_CONV` returns the theorem:

```
|- ?!x::P. Q[x] = (?x::P. Q[x]) /\ (!x y::P. Q[x] /\ Q[y] ==> (x = y))
```

which is the underlying semantic representation of the restricted unique existential quantification.

Failure

Fails if applied to a term not of the form `?x!::P. Q`.

See also

`res_quanLib.RES_FORALL_CONV`, `res_quanLib.RES_EXISTS_CONV`

RES_FORALL_AND_CONV (res_quanLib)

`res_quanLib.RES_FORALL_AND_CONV : conv`

Splits a restricted universal quantification across a conjunction.

When applied to a term of the form `!x::P. Q /\ R`, the conversion `RES_FORALL_AND_CONV` returns the theorem:

```
|- (!x::P. Q /\ R) = ((!x::P. Q) /\ (!x::P. R))
```

Failure

Fails if applied to a term not of the form `!x::P. Q /\ R`.

RES_FORALL_CONV (res_quanLib)

`res_quanLib.RES_FORALL_CONV : conv`

Converts a restricted universal quantification to an implication.

When applied to a term of the form `!x::P. Q`, the conversion `RES_FORALL_CONV` returns the theorem:

```
|- !x::P. Q = (!x. x IN P ==> Q)
```

which is the underlying semantic representation of the restricted universal quantification.

Failure

Fails if applied to a term not of the form `!x::P. Q`.

See also

res_quanLib.IMP_RES_FORALL_CONV

RES_FORALL_SWAP_CONV (res_quanLib)

res_quanLib.RES_FORALL_SWAP_CONV : conv

Changes the order of two restricted universal quantifications.

When applied to a term of the form !x::P. !y::Q. R, the conversion RES_FORALL_SWAP_CONV returns the theorem:

```
|- (!x::P. !y::Q. R) =  !y::Q. !x::P. R
```

providing that x does not occur free in Q and y does not occur free in P.

Failure

Fails if applied to a term not of the correct form.

See also

res_quanLib.RES_FORALL_CONV

RES_SELECT_CONV (res_quanLib)

res_quanLib.RES_SELECT_CONV : conv

Converts a restricted choice quantification to a conjunction.

When applied to a term of the form @x::P. Q[x], the conversion RES_SELECT_CONV returns the theorem:

```
|- @x::P. Q[x] = (@x. x IN P /\ Q[x])
```

which is the underlying semantic representation of the restricted choice quantification.

Failure

Fails if applied to a term not of the form `@x::P. Q`.

See also

`res_quanLib.RES_FORALL_CONV`, `res_quanLib.RES_EXISTS_CONV`

RES_TAC (Tactic)

```
Tactic.RES_TAC : tactic
```

Enriches assumptions by repeatedly resolving them against each other.

`RES_TAC` searches for pairs of assumed assumptions of a goal (that is, for a candidate implication and a candidate antecedent, respectively) which can be 'resolved' to yield new results. The conclusions of all the new results are returned as additional assumptions of the subgoal(s). The effect of `RES_TAC` on a goal is to enrich the assumptions set with some of its collective consequences.

When applied to a goal `A ?- g`, the tactic `RES_TAC` uses `RES_CANON` to obtain a set of implicative theorems in canonical form from the assumptions `A` of the goal. Each of the resulting theorems (if there are any) will have the form:

```
A |- u1 ==> u2 ==> ... ==> un ==> v
```

`RES_TAC` then tries to repeatedly 'resolve' these theorems against the assumptions of a goal by attempting to match the antecedents `u1`, `u2`, `...`, `un` (in that order) to some assumption of the goal (i.e. to some candidate antecedents among the assumptions). If all the antecedents can be matched to assumptions of the goal, then an instance of the theorem

```
A u {a1,...,an} |- v
```

called a 'final resolvent' is obtained by repeated specialization of the variables in the implicative theorem, type instantiation, and applications of modus ponens. If only the first `i` antecedents `u1`, `...`, `ui` can be matched to assumptions and then no further matching is possible, then the final resolvent is an instance of the theorem:

```
A u {a1,...,ai} |- u(i+1) ==> ... ==> v
```

All the final resolvents obtained in this way (there may be several, since an antecedent `ui` may match several assumptions) are added to the assumptions of the goal, in the stripped form produced by using `STRIP_ASSUME_TAC`. If the conclusion of any final resolvent is a contradiction 'F' or is alpha-equivalent to the conclusion of the goal, then `RES_TAC` solves the goal.

Failure

`RES_TAC` cannot fail and so should not be unconditionally `REPEAT`ed. However, since the final resolvents added to the original assumptions are never used as 'candidate antecedents' it is sometimes necessary to apply `RES_TAC` more than once to derive the desired result.

See also

`Tactic.IMP_RES_TAC`, `Thm_cont.IMP_RES_THEN`, `Drule.RES_CANON`, `Thm_cont.RES_THEN`

RES_THEN (Thm_cont)

```
Thm_cont.RES_THEN : (thm_tactic -> tactic)
```

Resolves all implicative assumptions against the rest.

Like the basic resolution function `IMP_RES_THEN`, the resolution tactic `RES_THEN` performs a single-step resolution of an implication and the assumptions of a goal. `RES_THEN` differs from `IMP_RES_THEN` only in that the implications used for resolution are taken from the assumptions of the goal itself, rather than supplied as an argument.

When applied to a goal `A ?- g`, the tactic `RES_THEN ttac` uses `RES_CANON` to obtain a set of implicative theorems in canonical form from the assumptions `A` of the goal. Each of the resulting theorems (if there are any) will have the form:

```
ai |- !x1...xn. ui ==> vi
```

where `ai` is one of the assumptions of the goal. Having obtained these implications, `RES_THEN` then attempts to match each antecedent `ui` to each assumption `aj |- aj` in the assumptions `A`. If the antecedent `ui` of any implication matches the conclusion `aj` of any assumption, then an instance of the theorem `ai, aj |- vi`, called a 'resolvent', is obtained by specialization of the variables `x1`, ..., `xn` and type instantiation, followed by an application of modus ponens. There may be more than one canonical implication derivable from the assumptions of the goal and each such implication is tried against every assumption, so there may be several resolvents (or, indeed, none).

Tactics are produced using the theorem-tactic `ttac` from all these resolvents (failures of `ttac` at this stage are filtered out) and these tactics are then applied in an unspecified sequence to the goal. That is,

```
RES_THEN ttac (A ?- g)
```

has the effect of:

```
MAP_EVERY (mapfilter ttac [... ; (ai,aj |- vi) ; ...]) (A ?- g)
```

where the theorems `ai,aj |- vi` are all the consequences that can be drawn by a (single) matching modus-ponens inference from the assumptions `A` and the implications derived using `RES_CANON` from the assumptions. The sequence in which the theorems `ai,aj |- vi` are generated and the corresponding tactics applied is unspecified.

Failure

Evaluating `RES_THEN ttac th` fails with '`no implication`' if no implication(s) can be derived from the assumptions of the goal by the transformation process described under the entry for `RES_CANON`. Evaluating `RES_THEN ttac (A ?- g)` fails with '`no resolvents`' if no assumption of the goal `A ?- g` can be resolved with the derived implication or implications. Evaluation also fails, with '`no tactics`', if there are resolvents, but for every resolvent `ai,aj |- vi` evaluating the application `ttac (ai,aj |- vi)` fails—that is, if for every resolvent `ttac` fails to produce a tactic. Finally, failure is propagated if any of the tactics that are produced from the resolvents by `ttac` fails when applied in sequence to the goal.

See also

`Tactic.IMP_RES_TAC`, `Thm_cont.IMP_RES_THEN`, `Drule.MATCH_MP`, `Drule.RES_CANON`, `Tactic.RES_TAC`

reset (Lib)

```
Lib.reset : ('a,'b) istream -> ('a,'b) istream
```

Restart an istream.

An application `reset istrm` replaces the current state of `istrm` with the value supplied when `istrm` was constructed.

Failure

Never fails.

Example

```
> reset(next(next
  (mk_istream (fn x => x+1) 0 (concat "gsym" o int_to_string))));
Exception- Type error in function application.
   Function: concat : string list -> string
   Argument: "gsym" : string
   Reason:
      Can't unify string list (*In Basis*) with string (*In Basis*)
         (Different type constructors)
Type error in function application.
   Function: o : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
   Argument: (concat "gsym", int_to_string) : string * (int -> string)
   Reason: Can't unify string to 'a -> 'b (Incompatible types)
Fail "Static Errors" raised

> state it;
Exception- Type error in function application.
   Function: state : ('a, 'b) istream -> 'b
   Argument: it : unit
   Reason: Can't unify ('a, 'b) istream to {} (Incompatible types)
Fail "Static Errors" raised
```

Comments

Perhaps the type of `reset` should be `('a,'b) istream -> unit`.

See also

`Lib.mk_istream`, `Lib.next`, `Lib.state`

reset_trace (Feedback)

```
Feedback.reset_trace : string -> unit
```

Resets a tracing variable to its default value.

A call to `reset_trace n` resets the tracing variable associated with the name `n` to its default value, i.e., the value of the expression `!r` when `n` was registered with `register_trace n r`.

Failure

Fails if the name given is not associated with a registered tracing variable, or if a `set` function associated with a "functional" trace (see `register_ftrace`) fails.

See also

`Feedback`, `Feedback.register_trace`, `Feedback.set_trace`, `Feedback.reset_traces`, `Feedback.trace`, `Feedback.traces`

reset_traces (Feedback)

```
Feedback.reset_traces : unit -> unit
```

Resets all registered tracing variables to their default values.

Failure

Fails if a `set` function associated with a "functional" trace (see `register_ftrace`) fails.

See also

`Feedback`, `Feedback.set_trace`, `Feedback.register_trace`, `Feedback.reset_trace`, `Feedback.trace`, `Feedback.traces`

RESORT_EXISTS_CONV (Conv)

```
Conv.RESORT_EXISTS_CONV : (term list -> term list) -> conv
```

Reorders bound variables under existential quantifiers.

A call to `RESORT_EXISTS_CONV f t` strips the outer existentially-quantified variables of `t`, giving a list `vs`, such that `t` is of the form `?vs. body`. The list `vs` is then passed to the function argument `f`. The result of the call `f vs` is expected to be a new list of variables `vs'`, and the result of the conversion is the theorem

```
|- (?vs. body) <=> (?vs'. body)
```

The function `f` is generally expected to return a permutation of the variables appearing in the term `vs`, but may in fact introduce fresh variables that are fresh for `body`, and may also remove variables from `vs` that also don't appear in `body`.

Failure

Given a term `t`, fails if `t` is not of boolean type. Fails if when applied to the outermost existentially quantified variables (permitted to be the empty list) the function `f` returns a list of terms that are not all variables. Also fails if either `f` returns a list that does not include variables from `vs` that appear in the body of `t`, or if it includes variables that are in the body, but which were not originally bound.

See also

`Conv.RESORT_FORALL_CONV`, `HolKernel.sort_vars`

RESORT_FORALL_CONV (Conv)

```
Conv.RESORT_FORALL_CONV : (term list -> term list) -> conv
```

Reorders bound variables under universal quantifiers.

A call to `RESORT_FORALL_CONV f t` strips the outer universally-quantified variables of `t`, giving a list `vs`, such that `t` is of the form `!vs. body`. The list `vs` is then passed to the function argument `f`. The result of the call `f vs` is expected to be a new list of variables `vs'`, and the result of the conversion is the theorem

```
|- (!vs. body) <=> (!vs'. body)
```

The function `f` is generally expected to return a permutation of the variables appearing in the term `vs`, but may in fact introduce fresh variables that are fresh for `body`, and may also remove variables from `vs` that also don't appear in `body`.

Failure

Given a term `t`, fails if `t` is not of boolean type. Fails if when applied to the outermost universally quantified variables (permitted to be the empty list) the function `f` returns a list of terms that are not all variables. Also fails if either `f` returns a list that does not include variables from `vs` that appear in the body of `t`, or if it includes variables that are in the body, but which were not originally bound.

See also

`Conv.RESORT_EXISTS_CONV`, `HolKernel.sort_vars`

RESQ_EXISTS_TAC (res_quanLib)

```
res_quanLib.RESQ_EXISTS_TAC : term -> tactic
```

Strips the outermost restricted existential quantifier from the conclusion of a goal.

When applied to a goal `A ?- ?x::P. t`, the tactic `RESQ_EXISTS_TAC` reduces it to a new subgoal `A ?- P x' /\ t[x'/x]` where `x'` is a variant of `x` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `x'` is just `x`.

```
  A ?- ?x::P. t
====================  RESQ_EXISTS_TAC
 A ?- P x' /\ t[x'/x]
```

Failure

Fails unless the goal's conclusion is a restricted extistential quantification.

RESQ_GEN_TAC (res_quanLib)

```
res_quanLib.RESQ_GEN_TAC : tactic
```

Strips the outermost restricted universal quantifier from the conclusion of a goal.

When applied to a goal `A ?- !x::P. t`, the tactic `RESQ_GEN_TAC` reduces it to a new goal `A,P x' ?- t[x'/x]` where `x'` is a variant of `x` chosen to avoid clashing with any variables free in the goal's assumption list. Normally `x'` is just `x`.

```
  A ?- !x::P. t
==================  RESQ_GEN_TAC
 A,P x' ?- t[x'/x]
```

Failure

Fails unless the goal's conclusion is a restricted universal quantification.

The tactic `REPEAT RESQ_GEN_TAC` strips away a series of restricted universal quantifiers, and is commonly used before tactics relying on the underlying term structure.

See also

`res_quanLib.RESQ_SPEC`, `res_quanLib.RESQ_SPECL`, `Tactic.STRIP_TAC`, `Tactic.GEN_TAC`, `Tactic.X_GEN_TAC`

RESQ_HALF_SPEC (res_quanLib)

```
res_quanLib.RESQ_HALF_SPEC : thm -> thm
```

Strip a restricted universal quantification in the conclusion of a theorem.

When applied to a theorem `A |- !x::P. t`, the derived inference rule `RESQ_HALF_SPEC` returns the theorem `A |- !x. x IN P ==> t`, i.e., it transforms the restricted universal quantification to its underlying semantic representation.

```
    A |- !x::P. t
--------------------  RESQ_HALF_SPEC
 A |- !x. x IN P ==> t
```

Failure

Fails if the theorem's conclusion is not a restricted universal quantification.

See also

`res_quanLib.RESQ_SPEC`, `res_quanLib.RESQ_SPECL`

RESQ_IMP_RES_TAC (res_quanLib)

```
res_quanLib.RESQ_IMP_RES_TAC : thm_tactic
```

Repeatedly resolves a restricted universally quantified theorem with the assumptions of a goal.

The function `RESQ_IMP_RES_TAC` performs repeatedly resolution using a restricted quantified theorem. It takes a restricted quantified theorem and transforms it into an implication. This resulting theorem is used in the resolution.

Given a theorem `th`, the theorem-tactic `RESQ_IMP_RES_TAC` applies `RESQ_IMP_RES_THEN` repeatedly to resolve the theorem with the assumptions.

Failure

Never fails

See also

res_quanLib.RESQ_IMP_RES_THEN, res_quanLib.RESQ_RES_THEN, res_quanLib.RESQ_RES_TAC, Thm_cont.IMP_RES_THEN, Tactic.IMP_RES_TAC, Drule.MATCH_MP, Drule.RES_CANON, Tactic.RES_TAC, Thm_cont.RES_THEN

RESQ_IMP_RES_THEN (res_quanLib)

```
res_quanLib.RESQ_IMP_RES_THEN : thm_tactical
```

Resolves a restricted universally quantified theorem with the assumptions of a goal.

The function RESQ_IMP_RES_THEN is the basic building block for resolution using a restricted quantified theorem. It takes a restricted quantified theorem and transforms it into an implication. This resulting theorem is used in the resolution.

Given a theorem-tactic ttac and a theorem th, the theorem-tactical RESQ_IMP_RES_THEN transforms the theorem into an implication th'. It then passes th' together with ttac to IMP_RES_THEN to carry out the resolution.

Failure

Evaluating RESQ_IMP_RES_THEN ttac th fails if the supplied theorem th is not restricted universally quantified, or if the call to IMP_RES_THEN fails.

See also

res_quanLib.RESQ_IMP_RES_TAC, res_quanLib.RESQ_RES_THEN, res_quanLib.RESQ_RES_TAC, Thm_cont.IMP_RES_THEN, Tactic.IMP_RES_TAC, Drule.MATCH_MP, Drule.RES_CANON, Tactic.RES_TAC, Thm_cont.RES_THEN

RESQ_MATCH_MP (res_quanLib)

```
res_quanLib.RESQ_MATCH_MP : (thm -> thm -> thm)
```

Eliminating a restricted universal quantification with automatic matching.

When applied to theorems `A1 |- !x::P. Q[x]` and `A2 |- P x'`, the derived inference rule `RESQ_MATCH_MP` matches `x'` to `x` by instantiating free or universally quantified variables in the first theorem (only), and returns a theorem `A1 u A2 |- Q[x'/x]`. Polymorphic types are also instantiated if necessary.

```
 A1 |- !x::P.Q[x]   A2 |- P x'
-------------------------------------  RESQ_MATCH_MP
       A1 u A2 |- Q[x'/x]
```

Failure

Fails unless the first theorem is a (possibly repeatedly) restricted universal quantification whose quantified variable can be instantiated to match the conclusion of the second theorem, without instantiating any variables which are free in `A1`, the first theorem's assumption list.

See also

`Drule.MATCH_MP`, `res_quanLib.RESQ_HALF_SPEC`

RESQ_RES_TAC (res_quanLib)

```
res_quanLib.RESQ_RES_TAC : tactic
```

Enriches assumptions by repeatedly resolving restricted universal quantifications in them against the others.

`RESQ_RES_TAC` uses those assumptions which are restricted universal quantifications in resolution in a way similar to `RES_TAC`. It calls `RESQ_RES_THEN` repeatedly until there is

no more resolution can be done. The conclusions of all the new results are returned as additional assumptions of the subgoal(s). The effect of RESQ_RES_TAC on a goal is to enrich the assumption set with some of its collective consequences.

Failure

RESQ_RES_TAC cannot fail and so should not be unconditionally REPEATed.

See also

res_quanLib.RESQ_IMP_RES_TAC, res_quanLib.RESQ_IMP_RES_THEN, res_quanLib.RESQ_RES_THEN, Tactic.IMP_RES_TAC, Thm_cont.IMP_RES_THEN, Drule.RES_CANON, Thm_cont.RES_THEN, Tactic.RES_TAC

RESQ_RES_THEN (res_quanLib)

```
res_quanLib.RESQ_RES_THEN : thm_tactic -> tactic
```

Resolves all restricted universally quantified assumptions against other assumptions of a goal.

Like the function RESQ_IMP_RES_THEN, the function RESQ_RES_THEN performs a single step resolution. The difference is that the restricted universal quantification used in the resolution is taken from the assumptions.

Given a theorem-tactic ttac, applying the tactic RESQ_RES_THEN ttac to a goal (asml,gl) has the effect of:

```
MAP_EVERY (mapfilter ttac [... ; (ai,aj |- vi) ; ...]) (amsl ?- g)
```

where the theorems ai,aj |- vi are all the consequences that can be drawn by a (single) matching modus-ponens inference from the assumptions amsl and the implications derived from the restricted universal quantifications in the assumptions.

Failure

Evaluating RESQ_RES_TAC ttac th fails if there are no restricted universal quantifications in the assumptions, or if the theorem-tactic ttac applied to all the consequences fails.

See also

res_quanLib.RESQ_IMP_RES_TAC, res_quanLib.RESQ_IMP_RES_THEN, res_quanLib.RESQ_RES_TAC, Thm_cont.IMP_RES_THEN, Tactic.IMP_RES_TAC, Drule.MATCH_MP, Drule.RES_CANON, Tactic.RES_TAC, Thm_cont.RES_THEN

RESQ_REWR_CANON (res_quanLib)

```
res_quanLib.RESQ_REWR_CANON : thm -> thm
```

Transform a theorem into a form accepted for rewriting.

RESQ_REWR_CANON transforms a theorem into a form accepted by COND_REWR_TAC. The input theorem should be headed by a series of restricted universal quantifications in the following form

```
!x1::P1. ... !xn::Pn. u[xi] = v[xi])
```

Other variables occurring in u and v may be universally quantified. The output theorem will have all ordinary universal quantifications moved to the outer most level with possible renaming to prevent variable capture, and have all restricted universal quantifications converted to implications. The output theorem will be in the form accepted by COND_REWR_TAC.

Failure

This function fails is the input theorem is not in the correct form.

See also

res_quanLib.RESQ_REWRITE1_TAC, res_quanLib.RESQ_REWRITE1_CONV, Cond_rewrite.COND_REWR_CANON, Cond_rewrite.COND_REWR_TAC, Cond_rewrite.COND_REWR_CONV

RESQ_REWRITE1_CONV (res_quanLib)

```
res_quanLib.RESQ_REWRITE1_CONV : thm list -> thm -> conv
```

Rewriting conversion using a restricted universally quantified theorem.

RESQ_REWRITE1_CONV is a rewriting conversion similar to COND_REWRITE1_CONV. The only difference is the rewriting theorem it takes. This should be an equation with restricted universal quantification at the outer level. It is converted to a theorem in the form accepted by the conditional rewriting conversion.

Suppose that th is the following theorem

```
A |- !x::P. Q[x] = R[x])
```

evaluating RESQ_REWRITE1_CONV thms th "t[x']" will return a theorem

```
A, P x' |- t[x'] = t'[x']
```

where t' is the result of substituting instances of R[x'/x] for corresponding instances of Q[x'/x] in the original term t[x]. All instances of P x' which do not appear in the original assumption asml are added to the assumption. The theorems in the list thms are used to eliminate the instances P x' if it is possible.

Failure

RESQ_REWRITE1_CONV fails if th cannot be transformed into the required form by the function RESQ_REWR_CANON. Otherwise, it fails if no match is found or the theorem cannot be instantiated.

See also

res_quanLib.RESQ_REWRITE1_TAC, res_quanLib.RESQ_REWR_CANON, Cond_rewrite.COND_REWR_TAC, Cond_rewrite.COND_REWRITE1_CONV, Cond_rewrite.COND_REWR_CONV, Cond_rewrite.COND_REWR_CANON, Cond_rewrite.search_top_down

RESQ_REWRITE1_TAC (res_quanLib)

```
res_quanLib.RESQ_REWRITE1_TAC : thm_tactic
```

Rewriting with a restricted universally quantified theorem.

RESQ_REWRITE1_TAC takes an equational theorem which is restricted universally quantified at the outer level. It calls RESQ_REWR_CANON to convert the theorem to the form accepted by COND_REWR_TAC and passes the resulting theorem to this tactic which carries out conditional rewriting.

Suppose that th is the following theorem

```
A |- !x::P. Q[x] = R[x])
```

Applying the tactic RESQ_REWRITE1_TAC th to a goal (asml,gl) will return a main subgoal (asml',gl') where gl' is obtained by substituting instances of R[x'/x] for corresponding instances of Q[x'/x] in the original goal gl. All instances of P x' which do not appear in the original assumption asml are added to it to form asml', and they also become new subgoals (asml,P x').

Failure

RESQ_REWRITE1_TAC th fails if th cannot be transformed into the required form by the function RESQ_REWR_CANON. Otherwise, it fails if no match is found or the theorem cannot be instantiated.

See also

res_quanLib.RESQ_REWRITE1_CONV, res_quanLib.RESQ_REWR_CANON, Cond_rewrite.COND_REWR_TAC, Cond_rewrite.COND_REWRITE1_CONV, Cond_rewrite.COND_REWR_CONV, Cond_rewrite.COND_REWR_CANON, Cond_rewrite.search_top_down

RESQ_SPEC (res_quanLib)

```
res_quanLib.RESQ_SPEC : term -> thm -> thm
```

Specializes the conclusion of a possibly-restricted universally quantified theorem.

When applied to a term u and a theorem A |- !x::P. t, RESQ_SPEC returns the theorem A, u IN P |- t[u/x]. If necessary, variables will be renamed prior to the specialization to ensure that u is free for x in t, that is, no variables free in u become bound after substitution.

```
    A |- !x::P. t
--------------------  RESQ_SPEC "u"
 A, u IN P |- t[u/x]
```

Additionally, if the input theorem is a standard universal quantification, then RESQ_SPEC behaves like SPEC.

Failure

Fails if the theorem's conclusion is not restricted universally quantified, or if type instantiation fails.

Example

The following example shows how RESQ_SPEC renames bound variables if necessary, prior to substitution: a straightforward substitution would result in the clearly invalid theorem (\y. 0 < y) y |- y = y.

```
> val th = RESQ_GEN ``x:num`` ``\y. 0 < y`` (REFL ``x:num``);
val th = |- !x :: \y. 0 < y. x = x : thm

> RESQ_SPEC ``y:num`` th;
val it = (\y'. 0 < y') y |- y = y : thm
```

See also

res_quanLib.RESQ_HALF_SPECL, res_quanLib.RESQ_SPECL

RESQ_SPECL (res_quanLib)

```
res_quanLib.RESQ_SPECL : (term list -> thm -> thm)
```

Specializes zero or more variables in the conclusion of a restricted universally quantified theorem.

When applied to a term list [u1;...;un] and a theorem A |- !x1::P1. ... !xn::Pn. t, the inference rule RESQ_SPECL returns the theorem

```
A,P1 u1,...,Pn un |- t[u1/x1]...[un/xn]
```

where the substitutions are made sequentially left-to-right in the same way as for RESQ_SPEC, with the same sort of alpha-conversions applied to t if necessary to ensure that no variables which are free in ui become bound after substitution.

```
        A |- !x1::P1. ... !xn::Pn. t
------------------------------------------  RESQ_SPECL "[u1;...;un]"
  A,P1 u1, ..., Pn un |- t[u1/x1]...[un/xn]
```

It is permissible for the term-list to be empty, in which case the application of RESQ_SPECL has no effect.

Failure

Fails if one of the specialization of the restricted universally quantified variable in the original theorem fails.

See also

res_quanLib.RESQ_GEN_TAC, res_quanLib.RESQ_SPEC

restart (proofManagerLib)

```
proofManagerLib.restart : unit -> proof
```

Returns the proof state to the initial goal.

The function restart is part of the subgoal package. A call to restart clears the proof history and returns to the initial goal. After a call to restart, the proof state is the same as it was after the inital call to set_goal (or g). For a description of the subgoal package, see set_goal.

Failure

The function restart only fails if no goalstack is being managed.

Restarting an interactive proof.

See also

proofManagerLib.set_goal, proofManagerLib.restart, proofManagerLib.backup, proofManagerLib.redo, proofManagerLib.restore, proofManagerLib.save, proofManagerLib.s backup, proofManagerLib.expand, proofManagerLib.expandf, proofManagerLib.p, proofManagerLib.top_thm, proofManagerLib.top_goal

restore (proofManagerLib)

```
proofManagerLib.restore : unit -> proof
```

Restores the proof state of the last save point, undoing the effects of expansions after the save point.

The function restore is part of the subgoal package. A call to restore restores the proof state to the last save point (a proof state saved by save). If the current state is a save point then restore clears the current save point and returns to the last save point. If there are no save points in the history, then restore returns to the initial goal and is equivalent to restart. For a description of the subgoal package, see set_goal.

Failure

The function restore will fail only if no goalstack is being managed.

Back tracking in a goal-directed proof to a user-defined save point.

See also

proofManagerLib.set_goal, proofManagerLib.restart, proofManagerLib.backup, proofManagerLib.redo, proofManagerLib.restore, proofManagerLib.save, proofManagerLib.s backup, proofManagerLib.expand, proofManagerLib.expandf, proofManagerLib.p, proofManagerLib.top_thm, proofManagerLib.top_goal

RESTR_EVAL_CONV (computeLib)

```
computeLib.RESTR_EVAL_CONV : term list -> conv
```

Symbolically evaluate a term, except for specified constants.

An application `RESTR_EVAL_CONV [c1, ..., cn] M` evaluates the term `M` in the call-by-value style of `EVAL`. When a type instance `c` of any element in `c1,...,cn` is encountered, `c` is not expanded by `RESTR_EVAL_CONV`. The effect is that evaluation stops at `c` (even though any arguments to `c` may be evaluated). This facility can be used to control `EVAL_CONV` to some extent.

Failure

Never fails, but may diverge.

Example

In the following, we first attempt to map the factorial function `FACT` over a list of variables. This attempt goes into a loop, because the conditional statement in the evaluation rule for `FACT` is never determine when the argument is equal to zero. However, if we suppress the evaluation of `FACT`, then we can return a useful answer.

```
- EVAL (Term `MAP FACT [x; y; z]`);   (* loops! *)
> Interrupted.

- val [FACT] = decls "FACT";   (* find FACT constant *)
> val FACT = `FACT` : term

- RESTR_EVAL_CONV [FACT] (Term `MAP FACT [x; y; z]`);

> val it = |- MAP FACT [x; y; z] = [FACT x; FACT y; FACT z] : thm
```

Controlling symbolic evaluation when it loops or becomes exponential.

See also

`bossLib.EVAL`, `computeLib.RESTR_EVAL_TAC`, `computeLib.RESTR_EVAL_RULE`, `Term.decls`

RESTR_EVAL_RULE (computeLib)

```
computeLib.RESTR_EVAL_RULE : term list -> thm -> thm
```

Symbolically evaluate a theorem, except for specified constants.

This is a version of RESTR_EVAL_CONV that works on theorems.

Failure

As for RESTR_EVAL_CONV.

Controlling symbolic evaluation when it loops or becomes exponential.

See also

bossLib.EVAL, bossLib.EVAL_RULE, computeLib.RESTR_EVAL_CONV, computeLib.RESTR_EVAL_TAC

RESTR_EVAL_TAC (computeLib)

```
computeLib.RESTR_EVAL_TAC : term list -> tactic
```

Symbolically evaluate a theorem, except for specified constants.

This is a tactic version of RESTR_EVAL_CONV.

Failure

As for RESTR_EVAL_CONV.

Controlling symbolic evaluation when it loops or becomes exponential.

See also

bossLib.EVAL, bossLib.EVAL_RULE, bossLib.EVAL_TAC, computeLib.RESTR_EVAL_CONV, computeLib.RESTR_EVAL_RULE

rev_assoc (Lib)

```
Lib.rev_assoc : ''a -> ('b * ''a) list -> 'b
```

Searches a list of pairs for a pair whose second component equals a specified value.

An invocation `rev_assoc y [(x1,y1),...,(xn,yn)]` locates the first `(xi,yi)` in a left-to-right scan of the list such that `yi` equals `y`. Then `xi` is returned. The lookup is done on an eqtype, i.e., the SML implementation must be able to decide equality for the type of `y`.

Failure

Fails if no matching pair is found. This will always be the case if the list is empty.

Example

```
> rev_assoc 2 [(1,4),(3,2),(2,5),(2,6)];
val it = 3: int
```

See also

`Lib.assoc`, `Lib.assoc1`, `Lib.assoc2`, `Lib.mem`, `Lib.tryfind`, `Lib.exists`, `Lib.all`

rev_assoc (hol88Lib)

```
hol88Lib.rev_assoc : ''a -> ('b * ''a) list -> 'b * ''a
```

Searches a list of pairs for a pair whose second component equals a specified value.

`rev_assoc y [(x1,y1),...,(xn,yn)]` returns the first `(xi,yi)` in the list such that `yi` equals `y`. The lookup is done on an eqtype, i.e., the SML implementation must be able to decide equality for the type of `y`.

Failure

Fails if no matching pair is found. This will always be the case if the list is empty.

Example

```
- rev_assoc 2 [(1,4),(3,2),(2,5),(2,6)];
(3, 2) : (int * int)
```

Comments

Superseded by `Lib.rev_assoc` and `Lib.assoc2`.

See also

`hol88Lib.assoc`, `Lib.rev_assoc`, `Lib.assoc2`

REV_FULL_SIMP_TAC (bossLib)

```
bossLib.REV_FULL_SIMP_TAC : simpset -> thm list -> tactic
```

Simplifies the goal (assumptions as well as conclusion) with the given simpset.

`REV_FULL_SIMP_TAC` is the same as `FULL_SIMP_TAC` except that it simplifies the assumptions in the opposite order.

That is, in `REV_FULL_SIMP_TAC`, each assumption is used to rewrite higher-numbered assumptions, whereas in `FULL_SIMP_TAC`, each assumption is used to rewrite lower-numbered assumptions.

See also

`bossLib.FULL_SIMP_TAC`, `bossLib.ASM_SIMP_TAC`, `bossLib.SIMP_TAC`

rev_itlist (Lib)

```
Lib.rev_itlist : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Applies a binary function between adjacent elements of the reverse of a list.

`rev_itlist f [x1,...,xn] b` returns `f xn ( ... (f x2 (f x1 b))...)`. It returns `b` if the second argument is an empty list.

Failure

Fails if some application of `f` fails.

Example

```
> rev_itlist (curry op * ) [1,2,3,4] 1;
val it = 24: int
```

See also

`Lib.itlist`, `Lib.itlist2`, `Lib.rev_itlist2`, `Lib.end_itlist`

rev_itlist2 (Lib)

```
Lib.rev_itlist2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
```

Applies a function to corresponding elements of 2 lists.

`rev_itlist2 f [x1,...,xn] [y1,...,yn] z` returns

```
   f xn yn (f xn-1 yn-1 ... (f x1 y1 z)...)
```

It returns `z` if both lists are empty.

Failure

Fails if the two lists are of different lengths, or if an application of `f` raises an exception.

Example

```
> rev_itlist2 (fn x => fn y => cons (x,y)) [1,2] [3,4] [];
val it = [(2, 4), (1, 3)]: (int * int) list
```

See also

`Lib.itlist`, `Lib.rev_itlist`, `Lib.itlist2`, `Lib.end_itlist`

reveal (Parse)

```
Parse.reveal : string -> unit
```

Restores recognition of a constant by the quotation parser.

A call `reveal c`, where `c` the name of a (perhaps) hidden constant, will 'unhide' the constant, that is, will make the quotation parser map the identifier `c` to all current constants with the same name (there may be more than one such as different theories may re-use the same name).

Failure

Never fails, but prints a warning message if the string does not correspond to an actual constant.

Comments

The hiding of a constant only affects the quotation parser; the constant is still there in a theory. If the parameter `c` is already overloaded so as to map to other constants, these overloadings are not altered.

See also

`Parse.hide`, `Parse.hidden`, `Parse.remove_ovl_mapping`, `Parse.update_overload_maps`

REVERSE (Tactical)

```
Tactical.REVERSE : (tactic -> tactic)
```

Reverses the order of the generated subgoals.

The tactic `REVERSE T` is a tactic which applies `T` to a goal, and reverses the order of the subgoals generated by `T`.

Failure

The application of `REVERSE` to a tactic `T` never fails. The resulting composite tactic `REVERSE T` fails when applied to a goal if and only if `T` fails.

Comments

Intended for use with `THEN1` to pick the 'easy' subgoal.

Example

Given a goal

```
G1 /\ G2
```

use

```
CONJ_TAC THEN1 T0
THEN ...
```

if the first conjunct is easily dispatched with `T0`, and

```
REVERSE CONJ_TAC THEN1 T0
THEN ...
```

if it is the second conjunct that yields.

See also

Tactical.EVERY, Tactical.FIRST, Tactical.ORELSE, Tactical.THEN, Tactical.THEN1, Tactical.THENL

REVERSE_CONV (listLib)

listLib.REVERSE_CONV : conv

Computes by inference the result of reversing a list.

REVERSE_CONV takes a term tm in the following form:

```
REVERSE [x0;...xn]
```

It returns the theorem

```
|- REVERSE [x0;...xn] = [xn;...x0]
```

where the right-hand side is the list in the reverse order.

Failure

REVERSE_CONV tm fails if tm is not of the form described above.

Example

Evaluating

```
REVERSE_CONV “REVERSE [0;1;2;3;4]”;
```

returns the following theorem:

```
|- REVERSE [0;1;2;3;4] = [4;3;2;1;0]
```

See also

`listLib.FOLDL_CONV`, `listLib.FOLDR_CONV`, `listLib.list_FOLD_CONV`

REVERSE_LT (Tactical)

```
Tactical.REVERSE_LT : list_tactic
```

Reverses the order of a list of subgoals.

The list-tactic `REVERSE_LT` reverses the order of a list of subgoals.

Failure

Never fails.

Example

Where `tac` is a tactic, `tac THEN_LT REVERSE_LT` is equivalent to `REVERSE tac`

See also

`Tactical.THEN_LT`, `Tactical.REVERSE`, `Tactical.ROTATE_LT`

REWR_CONV (Conv)

```
Conv.REWR_CONV : (thm -> conv)
```

Uses an instance of a given equation to rewrite a term.

`REWR_CONV` is one of the basic building blocks for the implementation of rewriting in the HOL system. In particular, the term replacement or rewriting done by all the built-in rewriting rules and tactics is ultimately done by applications of `REWR_CONV` to appropriate subterms. The description given here for `REWR_CONV` may therefore be taken

as a specification of the atomic action of replacing equals by equals that is used in all these higher level rewriting tools.

The first argument to REWR_CONV is expected to be an equational theorem which is to be used as a left-to-right rewrite rule. The general form of this theorem is:

```
A |- t[x1,...,xn] = u[x1,...,xn]
```

where x1, ..., xn are all the variables that occur free in the left-hand side of the conclusion of the theorem but do not occur free in the assumptions. Any of these variables may also be universally quantified at the outermost level of the equation, as for example in:

```
A |- !x1...xn. t[x1,...,xn] = u[x1,...,xn]
```

Note that REWR_CONV will also work, but will give a generally undesirable result (see below), if the right-hand side of the equation contains free variables that do not also occur free on the left-hand side, as for example in:

```
A |- t[x1,...,xn] = u[x1,...,xn,y1,...,ym]
```

where the variables y1, ..., ym do not occur free in t[x1,...,xn].

If th is an equational theorem of the kind shown above, then REWR_CONV th returns a conversion that maps terms of the form t[e1,...,en/x1,...,xn], in which the terms e1, ..., en are free for x1, ..., xn in t, to theorems of the form:

```
A |- t[e1,...,en/x1,...,xn] = u[e1,...,en/x1,...,xn]
```

That is, REWR_CONV th tm attempts to match the left-hand side of the rewrite rule th to the term tm. If such a match is possible, then REWR_CONV returns the corresponding substitution instance of th.

If REWR_CONV is given a theorem th:

```
A |- t[x1,...,xn] = u[x1,...,xn,y1,...,ym]
```

where the variables y1, ..., ym do not occur free in the left-hand side, then the result of applying the conversion REWR_CONV th to a term t[e1,...,en/x1,...,xn] will be:

```
A |- t[e1,...,en/x1,...,xn] = u[e1,...,en,v1,...,vm/x1,...,xn,y1,...,ym]
```

where `v1`, ..., `vm` are variables chosen so as to be free nowhere in `th` or in the input term. The user has no control over the choice of the variables `v1`, ..., `vm`, and the variables actually chosen may well be inconvenient for other purposes. This situation is, however, relatively rare; in most equations the free variables on the right-hand side are a subset of the free variables on the left-hand side.

In addition to doing substitution for free variables in the supplied equational theorem (or 'rewrite rule'), `REWR_CONV th tm` also does type instantiation, if this is necessary in order to match the left-hand side of the given rewrite rule `th` to the term argument `tm`. If, for example, `th` is the theorem:

```
A |- t[x1,...,xn] = u[x1,...,xn]
```

and the input term `tm` is (a substitution instance of) an instance of `t[x1,...,xn]` in which the types `ty1`, ..., `tyi` are substituted for the type variables `vty1`, ..., `vtyi`, that is if:

```
tm = t[ty1,...,tyn/vty1,...,vtyn][e1,...,en/x1,...,xn]
```

then `REWR_CONV th tm` returns:

```
A |- (t = u)[ty1,...,tyn/vty1,...,vtyn][e1,...,en/x1,...,xn]
```

Note that, in this case, the type variables `vty1`, ..., `vtyi` must not occur anywhere in the hypotheses `A`. Otherwise, the conversion will fail.

Failure

`REWR_CONV th` fails if `th` is not an equation or an equation universally quantified at the outermost level. If `th` is such an equation:

```
th = A |- !v1....vi. t[x1,...,xn] = u[x1,...,xn,y1,...,yn]
```

then `REWR_CONV th tm` fails unless the term `tm` is alpha-equivalent to an instance of the left-hand side `t[x1,...,xn]` which can be obtained by instantiation of free type variables (i.e. type variables not occurring in the assumptions `A`) and substitution for the free variables `x1`, ..., `xn`.

As noted, `REWR_CONV th` will fail rather than substitute for variables or type variables which appear in the hypotheses `A`. To allow substitution in the hypotheses, use `REWR_CONV_A th`.

Example

The following example illustrates a straightforward use of REWR_CONV. The supplied rewrite rule is polymorphic, and both substitution for free variables and type instantiation may take place. EQ_SYM_EQ is the theorem:

```
|- !x:'a. !y. (x = y) = (y = x)
```

and REWR_CONV EQ_SYM_EQ behaves as follows:

```
- REWR_CONV EQ_SYM_EQ (Term `1 = 2`);
> val it = |- (1 = 2) = (2 = 1) : thm

- REWR_CONV EQ_SYM_EQ (Term `1 < 2`);
! Uncaught exception:
! HOL_ERR
```

The second application fails because the left-hand side x = y of the rewrite rule does not match the term to be rewritten, namely 1 < 2.

In the following example, one might expect the result to be the theorem A |- f 2 = 2, where A is the assumption of the supplied rewrite rule:

```
- REWR_CONV (ASSUME (Term `!x:'a. f x = x`)) (Term `f 2:num`);
! Uncaught exception:
! HOL_ERR
```

The application fails, however, because the type variable 'a appears in the assumption of the theorem returned by ASSUME (Term `!x:'a. f x = x`).

Failure will also occur in situations like:

```
- REWR_CONV (ASSUME (Term `f (n:num) = n`)) (Term `f 2:num`);
! Uncaught exception:
! HOL_ERR
```

where the left-hand side of the supplied equation contains a free variable (in this case n) which is also free in the assumptions, but which must be instantiated in order to match the input term.

See also

Conv.REWR_CONV_A, Rewrite.REWRITE_CONV

REWR_CONV_A (Conv)

```
Conv.REWR_CONV_A : (thm -> conv)
```

Uses an instance of a given equation to rewrite a term.

`REWR_CONV_A th` behaves as `REWR_CONV th` except that it allows substitution of variables or type variables which appear in the hypotheses of `th`.

Example

Consider the theorem `th`

```
[0 < n] |- (a * n = b * (n : int)) <=> (a = b)
```

and the term `tm`

```
f (a * m = b * (m : int)) x
```

Then `DEPTH_CONV (REWR_CONV_A th) tm` returns

```
[0 < m] |- f (a * m = b * m) x <=> f (a = b) x
```

Likewise, when the goal is `tm` above, `e (VALIDATE (CONV_TAC (DEPTH_CONV (REWR_CONV_A th))))` gives the two subgoals:

```
f (a = b) x

0 < m
```

See also

Conv.REWR_CONV, Rewrite.REWRITE_CONV, Conv.DEPTH_CONV, Tactic.CONV_TAC, Tactical.VALIDATE

REWRITE_CONV (Rewrite)

```
Rewrite.REWRITE_CONV : (thm list -> conv)
```

Rewrites a term including built-in tautologies in the list of rewrites.

Rewriting a term using REWRITE_CONV utilizes as rewrites two sets of theorems: the tautologies in the ML list implicit_rewrites and the ones supplied by the user. The rule searches top-down and recursively for subterms which match the left-hand side of any of the possible rewrites, until none of the transformations are applicable. There is no ordering specified among the set of rewrites.

Variants of this conversion allow changes in the set of equations used: PURE_REWRITE_CONV and others in its family do not rewrite with the theorems in implicit_rewrites.

The top-down recursive search for matches may not be desirable, as this may increase the number of inferences being made or may result in divergence. In this case other rewriting tools such as ONCE_REWRITE_CONV and GEN_REWRITE_CONV can be used, or the set of theorems given may be reduced.

See GEN_REWRITE_CONV for the general strategy for simplifying theorems in HOL using equational theorems.

Failure

Does not fail, but may diverge if the sequence of rewrites is non-terminating.

Used to manipulate terms by rewriting them with theorems. While resulting in high degree of automation, REWRITE_CONV can spawn a large number of inference steps. Thus, variants such as PURE_REWRITE_CONV, or other rules such as SUBST_CONV, may be used instead to improve efficiency.

See also

Rewrite.GEN_REWRITE_CONV, Rewrite.ONCE_REWRITE_CONV, Rewrite.PURE_REWRITE_CONV, Conv.REWR_CONV, Drule.SUBST_CONV

REWRITE_RULE (Rewrite)

```
Rewrite.REWRITE_RULE : (thm list -> thm -> thm)
```

Rewrites a theorem including built-in tautologies in the list of rewrites.

Rewriting a theorem using REWRITE_RULE utilizes as rewrites two sets of theorems: the tautologies in the ML list implicit_rewrites and the ones supplied by the user. The rule searches top-down and recursively for subterms which match the left-hand side of any of the possible rewrites, until none of the transformations are applicable. There is no ordering specified among the set of rewrites.

Variants of this rule allow changes in the set of equations used: PURE_REWRITE_RULE and others in its family do not rewrite with the theorems in implicit_rewrites. Rules such as ASM_REWRITE_RULE add the assumptions of the object theorem (or a specified subset of these assumptions) to the set of possible rewrites.

The top-down recursive search for matches may not be desirable, as this may increase the number of inferences being made or may result in divergence. In this case other rewriting tools such as ONCE_REWRITE_RULE and GEN_REWRITE_RULE can be used, or the set of theorems given may be reduced.

See GEN_REWRITE_RULE for the general strategy for simplifying theorems in HOL using equational theorems.

Failure

Does not fail, but may diverge if the sequence of rewrites is non-terminating.

Used to manipulate theorems by rewriting them with other theorems. While resulting in high degree of automation, REWRITE_RULE can spawn a large number of inference steps. Thus, variants such as PURE_REWRITE_RULE, or other rules such as SUBST, may be used instead to improve efficiency.

See also

Rewrite.ASM_REWRITE_RULE, Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Conv.REWR_CONV, Rewrite.REWRITE_CONV, Thm.SUBST

REWRITE_TAC (Rewrite)

```
Rewrite.REWRITE_TAC : (thm list -> tactic)
```

Rewrites a goal including built-in tautologies in the list of rewrites.

Rewriting tactics in HOL provide a recursive left-to-right matching and rewriting facility that automatically decomposes subgoals and justifies segments of proof in which equational theorems are used, singly or collectively. These include the unfolding of definitions, and the substitution of equals for equals. Rewriting is used either to advance or to complete the decomposition of subgoals.

REWRITE_TAC transforms (or solves) a goal by using as rewrite rules (i.e. as left-to-right replacement rules) the conclusions of the given list of (equational) theorems, as well as a set of built-in theorems (common tautologies) held in the ML variable implicit_rewrites. Recognition of a tautology often terminates the subgoaling process (i.e. solves the goal).

The equational rewrites generated are applied recursively and to arbitrary depth, with matching and instantiation of variables and type variables. A list of rewrites can set off an infinite rewriting process, and it is not, of course, decidable in general whether a rewrite set has that property. The order in which the rewrite theorems are applied is unspecified, and the user should not depend on any ordering.

See GEN_REWRITE_TAC for more details on the rewriting process. Variants of REWRITE_TAC allow the use of a different set of rewrites. Some of them, such as PURE_REWRITE_TAC, exclude the basic tautologies from the possible transformations. ASM_REWRITE_TAC and others include the assumptions at the goal in the set of possible rewrites.

Still other tactics allow greater control over the search for rewritable subterms. Several of them such as ONCE_REWRITE_TAC do not apply rewrites recursively. GEN_REWRITE_TAC allows a rewrite to be applied at a particular subterm.

Failure

REWRITE_TAC does not fail. Certain sets of rewriting theorems on certain goals may cause a non-terminating sequence of rewrites. Divergent rewriting behaviour results from a term t being immediately or eventually rewritten to a term containing t as a sub-term. The exact behaviour depends on the HOL implementation.

Example

The arithmetic theorem `GREATER_DEF`, `|- !m n. m > n = n < m`, is used below to advance a goal:

```
- REWRITE_TAC [GREATER_DEF] ([],``5 > 4``);
> ([([], ``4 < 5``)], -) : subgoals
```

It is used below with the theorem `LESS_0`, `|- !n. 0 < (SUC n)`, to solve a goal:

```
- val (gl,p) =
    REWRITE_TAC [GREATER_DEF, LESS_0] ([],``(SUC n) > 0``);
> val gl = [] : goal list
> val p = fn : proof

- p[];
> val it = |- (SUC n) > 0 : thm
```

Rewriting is a powerful and general mechanism in HOL, and an important part of many proofs. It relieves the user of the burden of directing and justifying a large number of minor proof steps. `REWRITE_TAC` fits a forward proof sequence smoothly into the general goal-oriented framework. That is, (within one subgoaling step) it produces and justifies certain forward inferences, none of which are necessarily on a direct path to the desired goal.

`REWRITE_TAC` may be more powerful a tactic than is needed in certain situations; if efficiency is at stake, alternatives might be considered. On the other hand, if more power is required, the simplification functions (`SIMP_TAC` and others) may be appropriate.

See also

`Rewrite.ASM_REWRITE_TAC`, `Rewrite.GEN_REWRITE_TAC`, `Rewrite.FILTER_ASM_REWRITE_TAC`, `Rewrite.FILTER_ONCE_ASM_REWRITE_TAC`, `Rewrite.ONCE_ASM_REWRITE_TAC`, `Rewrite.ONCE_REWRITE_TAC`, `Rewrite.PURE_ASM_REWRITE_TAC`, `Rewrite.PURE_ONCE_ASM_REWRITE_TAC`, `Rewrite.PURE_ONCE_REWRITE_TAC`, `Rewrite.PURE_REWRITE_TAC`, `Conv.REWR_CONV`, `Rewrite.REWRITE_CONV`, `simpLib.SIMP_TAC`, `Tactic.SUBST_TAC`

rewrites (bossLib)

```
bossLib.rewrites : thm list -> ssfrag
```

Also exported as `simpLib.rewrites`.

Creates an `ssfrag` value consisting of the given theorems as rewrites.

Failure

Never fails.

Example

Instead of writing the simpler `SIMP_CONV std_ss thmlist`, one could write

```
SIMP_CONV (std_ss ++ rewrites thmlist) []
```

More plausibly, `rewrites` can be used to create commonly used `ssfrag` values containing a great number of rewrites. This is how the basic system's various `ssfrag` values are constructed where those values consist only of rewrite theorems.

See also

`bossLib.++`, `simpLib.mk_simpset`, `simpLib.SSFRAG`, `bossLib.SIMP_CONV`

rewrites (simpLib)

```
simpLib.rewrites : thm list -> ssfrag
```

Re-exported from `bossLib.rewrites`. See that entry for full documentation.

rhs (boolSyntax)

```
boolSyntax.rhs : term -> term
```

Returns the right-hand side of an equation.

If `M` has the form `t1 = t2` then `rhs M` returns `t2`.

Failure

Fails if term is not an equality.

See also

`boolSyntax.lhs`, `boolSyntax.dest_eq`

RHS_CONV (Conv)

```
Conv.RHS_CONV : conv -> conv
```

Applies a conversion to the right-hand argument of an equality.

If `c` is a conversion that maps a term `t2` to the theorem `|- t2 = t2'`, then the conversion `RHS_CONV c` maps applications of the form `t1 = t2` to theorems of the form:

```
|- (t1 = t2) = (t1 = t2')
```

Failure

`RHS_CONV c tm` fails if `tm` is not an an equality `t1 = t2`, or if `tm` has this form but the conversion `c` fails when applied to the term `t2`. The function returned by `RHS_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

Example

```
> RHS_CONV reduceLib.REDUCE_CONV (Term`7 = (3 + 5)`);
val it = ⊢ 7 = 3 + 5 ⇔ 7 = 8: thm
```

Comments

RAND_CONV is similar, but works for any binary operator

See also

Conv.BINOP_CONV, Conv.LHS_CONV, numLib.REDUCE_CONV, Conv.RAND_CONV

RIGHT_AND_EXISTS_CONV (Conv)

```
Conv.RIGHT_AND_EXISTS_CONV : conv
```

Moves an existential quantification of the right conjunct outwards through a conjunction.

When applied to a term of the form `P /\ (?x.Q)`, the conversion RIGHT_AND_EXISTS_CONV returns the theorem:

```
|- P /\ (?x.Q) = (?x'. P /\ (Q[x'/x]))
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `P /\ (?x.Q)`.

See also

Conv.AND_EXISTS_CONV, Conv.EXISTS_AND_CONV, Conv.LEFT_AND_EXISTS_CONV

RIGHT_AND_FORALL_CONV (Conv)

```
Conv.RIGHT_AND_FORALL_CONV : conv
```

Moves a universal quantification of the right conjunct outwards through a conjunction.

When applied to a term of the form `P /\ (!x.Q)`, the conversion `RIGHT_AND_FORALL_CONV` returns the theorem:

```
|- P /\ (!x.Q) = (!x'. P /\ (Q[x'/x]))
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `P /\ (!x.Q)`.

See also

`Conv.AND_FORALL_CONV`, `Conv.FORALL_AND_CONV`, `Conv.LEFT_AND_FORALL_CONV`

RIGHT_AND_PEXISTS_CONV (PairRules)

`PairRules.RIGHT_AND_PEXISTS_CONV : conv`

Moves a paired existential quantification of the right conjunct outwards through a conjunction.

When applied to a term of the form `t /\ (?p. t)`, the conversion `RIGHT_AND_PEXISTS_CONV` returns the theorem:

```
|- t /\ (?p. u) = (?p'. t /\ (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables free in the input term.

Failure

Fails if applied to a term not of the form `t /\ (?p. u)`.

See also

Conv.RIGHT_AND_EXISTS_CONV, PairRules.AND_PEXISTS_CONV, PairRules.PEXISTS_AND_CONV, PairRules.LEFT_AND_PEXISTS_CONV

RIGHT_AND_PFORALL_CONV (PairRules)

PairRules.RIGHT_AND_PFORALL_CONV : conv

Moves a paired universal quantification of the right conjunct outwards through a conjunction.

When applied to a term of the form `t /\ (!p. u)`, the conversion RIGHT_AND_PFORALL_CONV returns the theorem:

```
|- t /\ (!p. u) = (!p'. t /\ (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables free in the input term.

Failure

Fails if applied to a term not of the form `t /\ (!p. u)`.

See also

Conv.RIGHT_AND_FORALL_CONV, PairRules.AND_PFORALL_CONV, PairRules.PFORALL_AND_CONV, PairRules.LEFT_AND_PFORALL_CONV

RIGHT_BETA (Drule)

Drule.RIGHT_BETA : (thm -> thm)

Beta-reduces a top-level beta-redex on the right-hand side of an equation.

When applied to an equational theorem, `RIGHT_BETA` applies beta-reduction at top level to the right-hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
 A |- s = (\x. t1) t2
---------------------  RIGHT_BETA
  A |- s = t1[t2/x]
```

Failure

Fails unless the theorem is equational, with its right-hand side being a top-level beta-redex.

See also

`Thm.BETA_CONV`, `Conv.BETA_RULE`, `Tactic.BETA_TAC`, `Drule.RIGHT_LIST_BETA`

RIGHT_CONV_RULE (Conv)

```
Conv.RIGHT_CONV_RULE : (conv -> thm -> thm)
```

Applies a conversion to the right-hand side of an equational theorem.

If `c` is a conversion that maps a term `"t2"` to the theorem `|- t2 = t2'`, then the rule `RIGHT_CONV_RULE c` infers `|- t1 = t2'` from the theorem `|- t1 = t2`. That is, if `c "t2"` returns `A' |- t2 = t2'`, then:

```
    A |- t1 = t2
--------------------  RIGHT_CONV_RULE c
 A u A' |- t1 = t2'
```

Note that if the conversion `c` returns a theorem with assumptions, then the resulting inference rule adds these to the assumptions of the theorem it returns.

Failure

`RIGHT_CONV_RULE c th` fails if the conclusion of the theorem `th` is not an equation, or if `th` is an equation but `c` fails when applied its right-hand side. The function returned by `RIGHT_CONV_RULE c` will also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

See also

`Conv.CONV_RULE`

RIGHT_ETA (Drule)

```
Drule.RIGHT_ETA : thm -> thm
```

Perform one step of eta-reduction on the right hand side of an equational theorem.

```
    A |- M = (\x. (N x))
   ----------------------   x not free in N
    A |- M = N
```

Failure

If the right hand side of the equation is not an eta-redex, or if the theorem is not an equation.

Example

```
> val INC_DEF = new_definition ("INC_DEF", Term`INC = \x. 1 + x`);
val INC_DEF = ⊢ INC = (λx. 1 + x): thm

> RIGHT_ETA INC_DEF;
val it = ⊢ INC = $+ 1: thm
```

See also

`Drule.ETA_CONV`, `Term.eta_conv`

RIGHT_IMP_EXISTS_CONV (Conv)

```
Conv.RIGHT_IMP_EXISTS_CONV : conv
```

Moves an existential quantification of the consequent outwards through an implication.

When applied to a term of the form `P ==> (?x.Q)`, the conversion `RIGHT_IMP_EXISTS_CONV` returns the theorem:

```
|- P ==> (?x.Q) = (?x'. P ==> (Q[x'/x]))
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `P ==> (?x.Q)`.

See also

`Conv.EXISTS_IMP_CONV`, `Conv.LEFT_IMP_FORALL_CONV`

RIGHT_IMP_FORALL_CONV (Conv)

```
Conv.RIGHT_IMP_FORALL_CONV : conv
```

Moves a universal quantification of the consequent outwards through an implication.

When applied to a term of the form `P ==> (!x.Q)`, the conversion `RIGHT_IMP_FORALL_CONV` returns the theorem:

```
|- P ==> (!x.Q) = (!x'. P ==> (Q[x'/x]))
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `P ==> (!x.Q)`.

See also

`Conv.FORALL_IMP_CONV`, `Conv.LEFT_IMP_EXISTS_CONV`

RIGHT_IMP_PEXISTS_CONV (PairRules)

`PairRules.RIGHT_IMP_PEXISTS_CONV : conv`

Moves a paired existential quantification of the consequent outwards through an implication.

When applied to a term of the form `t ==> (?p. u)`, `RIGHT_IMP_PEXISTS_CONV` returns the theorem:

```
|- t ==> (?p. u) = (?p'. t ==> (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure

Fails if applied to a term not of the form `t ==> (?p. u)`.

See also

`Conv.RIGHT_IMP_EXISTS_CONV`, `PairRules.PEXISTS_IMP_CONV`, `PairRules.LEFT_IMP_PFORALL_CONV`

RIGHT_IMP_PFORALL_CONV (PairRules)

`PairRules.RIGHT_IMP_PFORALL_CONV : conv`

Moves a paired universal quantification of the consequent outwards through an implication.

When applied to a term of the form `t ==> (!p. u)`, the conversion `RIGHT_IMP_FORALL_CONV` returns the theorem:

```
|- t ==> (!p. u) = (!p'. t ==> (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure

Fails if applied to a term not of the form `t ==> (!p. u)`.

See also

`Conv.RIGHT_IMP_FORALL_CONV`, `PairRules.PFORALL_IMP_CONV`, `PairRules.LEFT_IMP_EXISTS_CONV`

RIGHT_LIST_BETA (Drule)

```
Drule.RIGHT_LIST_BETA : (thm -> thm)
```

Iteratively beta-reduces a top-level beta-redex on the right-hand side of an equation.

When applied to an equational theorem, `RIGHT_LIST_BETA` applies beta-reduction over a top-level chain of beta-redexes to the right hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
 A |- s = (\x1...xn. t) t1 ... tn
---------------------------------  RIGHT_LIST_BETA
    A |- s = t[t1/x1]...[tn/xn]
```

Failure

Fails unless the theorem is equational, with its right-hand side being a top-level beta-redex.

See also

Thm.BETA_CONV, Conv.BETA_RULE, Tactic.BETA_TAC, Drule.LIST_BETA_CONV, Drule.RIGHT_-BETA

RIGHT_LIST_PBETA (PairRules)

```
PairRules.RIGHT_LIST_PBETA : (thm -> thm)
```

Iteratively beta-reduces a top-level paired beta-redex on the right-hand side of an equation.

When applied to an equational theorem, RIGHT_LIST_PBETA applies paired beta-reduction over a top-level chain of beta-redexes to the right-hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
 A |- s = (\p1...pn. t) q1 ... qn
---------------------------------  RIGHT_LIST_BETA
    A |- s = t[q1/p1]...[qn/pn]
```

Failure

Fails unless the theorem is equational, with its right-hand side being a top-level paired beta-redex.

See also

Drule.RIGHT_LIST_BETA, PairRules.PBETA_CONV, PairRules.PBETA_RULE, PairRules.PBETA_-TAC, PairRules.LIST_PBETA_CONV, PairRules.RIGHT_PBETA, PairRules.LEFT_PBETA, PairRules.LEFT_LIST_PBETA

RIGHT_OR_EXISTS_CONV (Conv)

```
Conv.RIGHT_OR_EXISTS_CONV : conv
```

Moves an existential quantification of the right disjunct outwards through a disjunction.

When applied to a term of the form `P \/ (?x.Q)`, the conversion `RIGHT_OR_EXISTS_CONV` returns the theorem:

```
|- P \/ (?x.Q) = (?x'. P \/ (Q[x'/x]))
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `P \/ (?x.Q)`.

See also

`Conv.OR_EXISTS_CONV`, `Conv.EXISTS_OR_CONV`, `Conv.LEFT_OR_EXISTS_CONV`

RIGHT_OR_FORALL_CONV (Conv)

```
Conv.RIGHT_OR_FORALL_CONV : conv
```

Moves a universal quantification of the right disjunct outwards through a disjunction.

When applied to a term of the form `P \/ (!x.Q)`, the conversion `RIGHT_OR_FORALL_CONV` returns the theorem:

```
|- P \/ (!x.Q) = (!x'. P \/ (Q[x'/x]))
```

where `x'` is a primed variant of `x` that does not appear free in the input term.

Failure

Fails if applied to a term not of the form `P \/ (!x.Q)`.

See also

`Conv.OR_FORALL_CONV`, `Conv.FORALL_OR_CONV`, `Conv.LEFT_OR_FORALL_CONV`

RIGHT_OR_PEXISTS_CONV (PairRules)

```
PairRules.RIGHT_OR_PEXISTS_CONV : conv
```

Moves a paired existential quantification of the right disjunct outwards through a disjunction.

When applied to a term of the form `t \/ (?p. u)`, the conversion `RIGHT_OR_PEXISTS_CONV` returns the theorem:

```
|- t \/ (?p. u) = (?p'. t \/ (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables free in the input term.

Failure

Fails if applied to a term not of the form `t \/ (?p. u)`.

See also

`Conv.RIGHT_OR_EXISTS_CONV`, `PairRules.OR_PEXISTS_CONV`, `PairRules.PEXISTS_OR_CONV`, `PairRules.LEFT_OR_PEXISTS_CONV`

RIGHT_OR_PFORALL_CONV (PairRules)

```
PairRules.RIGHT_OR_PFORALL_CONV : conv
```

Moves a paired universal quantification of the right disjunct outwards through a disjunction.

When applied to a term of the form `t \/ (!p. u)`, the conversion `RIGHT_OR_FORALL_CONV` returns the theorem:

```
|- t \/ (!p. u) = (!p'. t \/ (u[p'/p]))
```

where `p'` is a primed variant of the pair `p` that does not contain any variables that appear free in the input term.

Failure

Fails if applied to a term not of the form `t \/ (!p. u)`.

See also

`Conv.RIGHT_OR_FORALL_CONV`, `PairRules.OR_PFORALL_CONV`, `PairRules.PFORALL_OR_CONV`, `PairRules.LEFT_OR_PFORALL_CONV`

RIGHT_PBETA (PairRules)

```
PairRules.RIGHT_PBETA : (thm -> thm)
```

Beta-reduces a top-level paired beta-redex on the right-hand side of an equation.

When applied to an equational theorem, `RIGHT_PBETA` applies paired beta-reduction at top level to the right-hand side (only). Variables are renamed if necessary to avoid free variable capture.

```
 A |- s = (\p. t1) t2
---------------------  RIGHT_PBETA
  A |- s = t1[t2/p]
```

Failure

Fails unless the theorem is equational, with its right-hand side being a top-level paired beta-redex.

See also

Drule.RIGHT_BETA, PairRules.PBETA_CONV, PairRules.PBETA_RULE, PairRules.PBETA_TAC, PairRules.RIGHT_LIST_PBETA, PairRules.LEFT_PBETA, PairRules.LEFT_LIST_PBETA

ROTATE_LT (Tactical)

```
Tactical.ROTATE_LT : int -> list_tactic
```

Rotates a list of goals

`ROTATE_LT n gl` rotates a goal list `gl` by `n` places. For `n >= 0`, this means moving the first `n` goals to the end of the list. A negative `n` means rotating the list in the opposite direction.

Failure

Never fails.

Example

To bring the third goal to first position, leaving the others in order, use

```
SPLIT_LT 3 (ROTATE_LT ~1, ALL_LT)
```

Comments

Using SPLIT_LT, ROTATE_LT and REVERSE_LT, any reordering of a list of goals is possible.

See also

`proofManagerLib.rotate`, `proofManagerLib.r`, `Tactical.SPLIT_LT`, `Tactical.REVERSE_LT`, `Tactical.ALL_LT`

rpair (Lib)

```
Lib.rpair : 'a -> 'b -> 'b * 'a
```

Makes two values into a pair, in reverse order.

`rpair x y` returns `(y,x)`.

Failure

Never fails.

See also

`Lib.pair`, `Lib.swap`, `Lib.fst`, `Lib.snd`, `Lib.curry`, `Lib.uncurry`

Rsyntax (structure)

```
Rsyntax
```

A structure that restores a record-style environment for term manipulation.

If one has opened the `Psyntax` structure, one can open the Rsyntax structure to get record-style functions back.

Each function in the `Rsyntax` structure has a corresponding function in the Psyntax structure, and vice versa. One can flip-flop between the two structures by opening one and then the other. One can also use long identifiers in order to use both syntaxes at once.

Failure

Never fails.

Example

The following shows how to open the Rsyntax structure and the functions that subsequently become available in the top level environment. Documentation for each of these functions is available online.

```
- open Rsyntax;
open Rsyntax
val INST = fn : term subst -> thm -> thm
val INST_TYPE = fn : hol_type subst -> thm -> thm
val INST_TY_TERM = fn : term subst * hol_type subst -> thm -> thm
val SUBST = fn : {thm:thm, var:term} list -> term -> thm -> thm
val SUBST_CONV = fn : {thm:thm, var:term} list -> term -> term -> thm
val define_new_type_bijections = fn
  : {ABS:string, REP:string, name:string, tyax:thm} -> thm
val dest_abs = fn : term -> {Body:term, Bvar:term}
val dest_comb = fn : term -> {Rand:term, Rator:term}
val dest_cond = fn : term -> {cond:term, larm:term, rarm:term}
val dest_conj = fn : term -> {conj1:term, conj2:term}
val dest_cons = fn : term -> {hd:term, tl:term}
val dest_const = fn : term -> {Name:string, Ty:hol_type}
val dest_disj = fn : term -> {disj1:term, disj2:term}
val dest_eq = fn : term -> {lhs:term, rhs:term}
val dest_exists = fn : term -> {Body:term, Bvar:term}
val dest_forall = fn : term -> {Body:term, Bvar:term}
val dest_imp = fn : term -> {ant:term, conseq:term}
val dest_let = fn : term -> {arg:term, func:term}
val dest_list = fn : term -> {els:term list, ty:hol_type}
val dest_pabs = fn : term -> {body:term, varstruct:term}
val dest_pair = fn : term -> {fst:term, snd:term}
val dest_select = fn : term -> {Body:term, Bvar:term}
val dest_type = fn : hol_type -> {Args:hol_type list, Tyop:string}
val dest_var = fn : term -> {Name:string, Ty:hol_type}
val inst = fn : hol_type subst -> term -> term
val match_term = fn : term -> term -> term subst * hol_type subst
val match_type = fn : hol_type -> hol_type -> hol_type subst
```

```
val mk_abs = fn : {Body:term, Bvar:term} -> term
val mk_comb = fn : {Rand:term, Rator:term} -> term
val mk_cond = fn : {cond:term, larm:term, rarm:term} -> term
val mk_conj = fn : {conj1:term, conj2:term} -> term
val mk_cons = fn : {hd:term, tl:term} -> term
val mk_const = fn : {Name:string, Ty:hol_type} -> term
val mk_disj = fn : {disj1:term, disj2:term} -> term
val mk_eq = fn : {lhs:term, rhs:term} -> term
val mk_exists = fn : {Body:term, Bvar:term} -> term
val mk_forall = fn : {Body:term, Bvar:term} -> term
val mk_imp = fn : {ant:term, conseq:term} -> term
val mk_let = fn : {arg:term, func:term} -> term
val mk_list = fn : {els:term list, ty:hol_type} -> term
val mk_pabs = fn : {body:term, varstruct:term} -> term
val mk_pair = fn : {fst:term, snd:term} -> term
val mk_primed_var = fn : {Name:string, Ty:hol_type} -> term
val mk_select = fn : {Body:term, Bvar:term} -> term
val mk_type = fn : {Args:hol_type list, Tyop:string} -> hol_type
val mk_var = fn : {Name:string, Ty:hol_type} -> term
val new_binder = fn : {Name:string, Ty:hol_type} -> unit
val new_constant = fn : {Name:string, Ty:hol_type} -> unit
val new_infix = fn : {Name:string, Prec:int, Ty:hol_type} -> unit
val new_recursive_definition = fn
  : {def:term, fixity:fixity, name:string, rec_axiom:thm} -> thm
val new_specification = fn
  : {consts:{const_name:string, fixity:fixity} list,
     name:string, sat_thm:thm}
    -> thm
val new_type = fn : {Arity:int, Name:string} -> unit
val new_type_definition = fn
  : {inhab_thm:thm, name:string, pred:term} -> thm
val subst = fn : term subst -> term -> term
val subst_occs = fn : int list list -> term subst -> term -> term
val type_subst = fn : hol_type subst -> hol_type -> hol_type
```

See also

Psyntax

RULE_ASSUM_TAC (Tactic)

```
Tactic.RULE_ASSUM_TAC : ((thm -> thm) -> tactic)
```

Maps an inference rule over the assumptions of a goal.

When applied to an inference rule `f` and a goal `({A1,...,An} ?- t)`, the `RULE_ASSUM_TAC` tactical applies the inference rule to each of the `ASSUME`d assumptions to give a new goal.

```
            {A1,...,An} ?- t
   ==================================  RULE_ASSUM_TAC f
    {f(A1 |- A1),...,f(An |- An)} ?- t
```

Failure

The application of `RULE_ASSUM_TAC f` to a goal fails iff `f` fails when applied to any of the assumptions of the goal.

Comments

It does not matter if the goal has no assumptions, but in this case `RULE_ASSUM_TAC` has no effect.

See also

`Tactic.RULE_L_ASSUM_TAC`, `Tactical.ASSUM_LIST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`, `Tactical.POP_ASSUM_LIST`

RULE_L_ASSUM_TAC (Tactic)

```
Tactic.RULE_L_ASSUM_TAC : ((thm -> thm list) -> tactic)
```

Maps an inference rule, which produces a list of result theorems, over the assumptions of a goal.

When applied to an inference rule `f` and a goal `({A1,...,An} ?- t)`, the `RULE_L_ASSUM_TAC` tactical applies the inference rule to each of the `ASSUME`d assumptions to give a new goal.

```
          {A1,...,An} ?- t
==================================  RULE_L_ASSUM_TAC f
 {f(A1 |- A1),...,f(An |- An)} ?- t
```

Here each `f(Ai |- Ai)` is a list of assumptions.

Failure

The application of `RULE_L_ASSUM_TAC f` to a goal fails iff `f` fails when applied to any of the assumptions of the goal.

Comments

It does not matter if the goal has no assumptions, but in this case `RULE_L_ASSUM_TAC` has no effect.

Example

With a goal

```
g
------------------------------------
  0.  a /\ b
  1.  c
  2.  d /\ e /\ f
```

the tactic `RULE_L_ASSUM_TAC CONJUNCTS` gives the new goal

```
g
------------------------------------
  0.  a
  1.  b
```

2. c
3. d
4. e
5. f

RULE_L_ASSUM_TAC can also be used to delete unwanted assumptions: let `f ass = [ass]` for an assumption which is to be kept, and `f ass = []` for an assumption which is to be deleted.

See also

Tactic.RULE_ASSUM_TAC, Tactical.ASSUM_LIST, Tactical.MAP_EVERY, Tactical.MAP_FIRST, Tactical.POP_ASSUM_LIST

RW_TAC (BasicProvers)

```
BasicProvers.RW_TAC : simpset -> thm list -> tactic
```

Re-exported from bossLib.RW_TAC. See that entry for full documentation.

RW_TAC (bossLib)

```
bossLib.RW_TAC : simpset -> thm list -> tactic
```

Also exported as BasicProvers.RW_TAC.

Simplification with case-splitting and built-in knowledge of declared datatypes.

RW_TAC is a simplification tactic that provides conditional and contextual rewriting, and automatic invocation of conversions and decision procedures in the course of simplification. An application `RW_TAC ss thl` adds the theorems in `thl` to the simpset `ss` and proceeds to simplify the goal.

The process is based upon the simplification procedures in simpLib, but is more persistent in attempting to apply rewrite rules. It automatically incorporates relevant results from

datatype declarations (the most important of these are injectivity and distinctness of constructors). It uses the current hypotheses when rewriting the goal. It automatically performs case-splitting on conditional expressions in the goal. It simplifies any equation between constructors occurring in the goal or the hypotheses. It automatically substitutes through the goal any assumption that is an equality `v = M` or `M = v`, if `v` is a variable not occurring in `M`. It eliminates any boolean variable or negated boolean variable occurring as a hypothesis. It breaks down any conjunctions, disjunctions, double negations, or existentials occurring as hypotheses. It keeps the goal in "stripped" format so that the resulting goal will not be an implication or universally quantified.

Failure

Never fails, but may diverge.

Comments

The case splits arising from conditionals and disjunctions can result in many unforeseen subgoals. In that case, `SIMP_TAC` or even `REWRITE_TAC` should be used.

The automatic incorporation of datatype facts can be slow when operating in a context with many datatypes (or a few large datatypes). In such cases, `SRW_TAC` is preferable to `RW_TAC`.

See also

`bossLib.SRW_TAC`, `bossLib.SIMP_TAC`, `Rewrite.REWRITE_TAC`, `bossLib.Hol_datatype`

S (Lib)

```
Lib.S : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Generalized function composition: `S f g x` equals `f x (g x)`.

Failure

`S f` never fails and `S f g` never fails, but `S f g x` fails if `g x` fails or `f x (g x)` fails.

See also

Lib, Lib.##, Lib.C, Lib.I, Lib.K, Lib.W

same_const (Term)

```
Term.same_const : term -> term -> bool
```

Constant time equality check for constants.

In many cases, one needs to check that a constant is an instance of the generic constant originally introduced into the signature, or that two constants are both type instantiations of another. This can be achieved by taking the constants apart with dest_thy_const and comparing their name and theory. However, this is relatively inefficient. Instead, one can invoke same_const, which takes constant time.

Failure

Never fails.

Example

```
> same_const boolSyntax.universal (rator (concl BOOL_CASES_AX));
val it = true: bool
```

See also

Term.aconv, Term.dest_thy_const, Term.match_term

SAT_PROVE (HolSatLib)

```
HolSatLib.SAT_PROVE : Term.term -> Thm.thm
```

Proves that the supplied term is a tautology, or provides a counterexample.

The supplied term should be purely propositional, i.e., atoms must be Boolean variables or constants, and conditionals must be Boolean-valued. `SAT_PROVE` uses the MiniSat SAT solver's proof logging feature to construct and verify a resolution refutation for the negation of the supplied term.

Failure

Fails if the supplied term is not a tautology. In this case, a theorem providing a satisfying assignment for the negation of the input term is returned, wrapped in an exception.

Example

```
> load "HolSatLib"; open HolSatLib;
val it = (): unit
> SAT_PROVE ``(a ==> b) /\ (b ==> a) ==> (a=b)``;
val it = ⊢ (a ⇒ b) ∧ (b ⇒ a) ⇒ (a ⇔ b): thm
> SAT_PROVE ``~((a ==> b) /\ (b ==> a) ==> (a=b))``
   handle HolSatLib.SAT_cex th => th;
val it = ⊢ ¬b ∧ a ⇒ ¬¬((a ⇒ b) ∧ (b ⇒ a) ⇒ (a ⇔ b)): thm
```

Comments

If MiniSat terminates abnormally, or if the MiniSat binary cannot be located or executed, SAT_PROVE falls back to a slower propositional tautology prover implemented in SML. For low-level use of SAT solver facilities and other details, see the section on the HolSat library in the HOL Description.

save (proofManagerLib)

```
proofManagerLib.save : unit -> proof
```

Marks the current proof state as a save point, and clears the automatic undo history.

The function `save` is part of the subgoal package. A call to `save` clears the automatic proof history and marks the current state as a save point. A call to `backup` at a save point

will fail. In contrast to `forget_history`, however, `save` does not clear the initial goal or any previous save points. Therefore a call to `restore` or `restart` at a save point will succeed. For a description of the subgoal package, see `set_goal`.

Failure

The function `save` will fail only if no goalstack is being managed.

Creating save points in an interactive proof, to allow user-directed back tracking. This is in contrast to the automatic back tracking facilitated by `backup`.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.s` `backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

save_thm (Theory)

```
Theory.save_thm : string * thm -> thm
```

Stores a theorem in the current theory segment.

The call `save_thm(name, th)` adds the theorem `th` to the current theory segment under the name `name`. The theorem is also the return value of the call. When the current segment `thy` is exported, things are arranged in such a way that, if `thyTheory` is loaded into a later session, the ML variable `thyTheory.name` will have `th` as its value.

Failure

If `th` is out-of-date, then `save_thm` will fail. If `name` is not a valid ML alphanumeric identifier, `save_thm` will not fail, but `export_theory` will (printing an informative error message first).

Example

```
> val savethm_demo = save_thm("savethm_demo", REFL (Term `x:bool`));
val savethm_demo = ⊢ x ⇔ x: thm

> current_theorems();
val it =
   [("savethm_demo", ⊢ x ⇔ x),
    ("foo_ind", ⊢ ∀P. (∀x. (x ≠ 0 ⇒ P (x − 1)) ⇒ P x) ⇒ ∀v. P v),
    ("foo_def",
     ⊢ ∀x. foo$old45->foo<-old x =
           if x = 0 then 1 else x * foo$old45->foo<-old (x − 1)),
    ("fact_def",
     ⊢ ∀x. foo$fact x = if x = 0 then 1 else x * foo$fact (x − 1))]:
   (string * thm) list
```

Comments

If a theorem is already saved under `name` in the current theory segment, it will be overwritten.

The results of `new_axiom`, and definition principle (such as `new_definition`, `new_type_definition`, and `new_specification`) are automatically stored in the current theory: one does not have to call `save_thm` on them.

Saving important theorems for eventual export. Binding the result of `save_thm` to an ML variable makes it easy to access the theorem in the remainder of the current session.

See also

`Theory.new_theory`, `Tactical.store_thm`, `DB.fetch`, `DB.thy`, `Theory.current_definitions`, `Theory.current_theorems`, `Theory.uptodate_thm`, `Theory.new_axiom`, `Definition.new_type_definition`, `Definition.new_definition`, `Definition.new_specification`

say (Lib)

```
Lib.say : string -> unit
```

Print a string.

An application `say s` prints the string `s` on the standard output.

Failure

Never fails.

Comments

The Standard ML Basis Library structure `TextIO` offers related functions.

SBC_CONV (reduceLib)

```
reduceLib.SBC_CONV : conv
```

Calculates by inference the difference of two numerals.

If `m` and `n` are numerals (e.g. `0`, `1`, `2`, `3`,...), then `SBC_CONV "m - n"` returns the theorem:

```
|- m - n = s
```

where `s` is the numeral that denotes the difference of the natural numbers denoted by `m` and `n`.

Failure

`SBC_CONV tm` fails unless `tm` is of the form `"m - n"`, where `m` and `n` are numerals.

Example

```
#SBC_CONV "25 - 30";;
|- 25 - 30 = 0

#SBC_CONV "200 - 200";;
|- 200 - 200 = 0
```

```
#SBC_CONV "60 - 17";;
|- 60 - 17 = 43
```

SCANL_CONV (listLib)

```
listLib.SCANL_CONV : conv -> conv
```

Computes by inference the result of applying a function to the elements of a list.

`SCANL_CONV` takes a conversion `conv` and a term `tm` in the following form:

```
SCANL f e0 [x1;...xn]
```

It returns the theorem

```
|- SCANL f e0 [x1;...xn] = [e0; e1; ...;en]
```

where `ei` is the result of applying the function `f` to the result of the previous iteration and the current element, i.e., `ei = f e(i-1) xi`. The iteration starts from the left-hand side (the head) of the list. The user supplied conversion `conv` is used to derive a theorem

```
|- f e(i-1) xi = ei
```

which is used in the next iteration.

Failure

`SCANL_CONV conv tm` fails if `tm` is not of the form described above, or failure occurs when evaluating `conv` "`f e(i-1) xi`".

Example

To sum the elements of a list and save the result at each step, one can use `SCANL_CONV` with `ADD_CONV` from the library `num_lib`.

```
- load_library_in_place num_lib;
- SCANL_CONV Num_lib.ADD_CONV “SCANL $+ 0 [1;2;3]”;
|- SCANL $+ 0[1;2;3] = [0;1;3;6]
```

In general, if the function `f` is an explicit lambda abstraction `(\x x'. t[x,x'])`, the conversion should be in the form

```
((RATOR_CONV BETA_CONV) THENC BETA_CONV THENC conv'))
```

where `conv'` applied to `t[x,x']` returns the theorem

```
|-t[x,x'] = e''.
```

See also

`listLib.SCANR_CONV`, `listLib.FOLDL_CONV`, `listLib.FOLDR_CONV`, `listLib.list_FOLD_CONV`

SCANR_CONV (listLib)

```
listLib.SCANR_CONV : conv -> conv
```

Computes by inference the result of applying a function to the elements of a list.

`SCANR_CONV` takes a conversion `conv` and a term `tm` in the following form:

```
SCANR f e0 [xn;...;x1]
```

It returns the theorem

```
|- SCANR f e0 [xn;...;x1] = [en; ...;e1;e0]
```

where `ei` is the result of applying the function `f` to the result of the previous iteration and the current element, i.e., `ei = f e(i-1) xi`. The iteration starts from the right-hand side (the tail) of the list. The user supplied conversion `conv` is used to derive a theorem

```
|- f e(i-1) xi = ei
```

which is used in the next iteration.

Failure

`SCANR_CONV conv tm` fails if `tm` is not of the form described above, or failure occurs when evaluating `conv "f e(i-1) xi"`.

Example

To sum the elements of a list and save the result at each step, one can use `SCANR_CONV` with `ADD_CONV` from the library `num_lib`.

```
- load_library_in_place num_lib;
- SCANR_CONV Num_lib.ADD_CONV "SCANR $+ 0 [1;2;3]";
|- SCANR $+ 0[1;2;3] = [6;5;3;0]
```

In general, if the function `f` is an explicit lambda abstraction `(\x x'. t[x,x'])`, the conversion should be in the form

```
((RATOR_CONV BETA_CONV) THENC BETA_CONV THENC conv'))
```

where `conv'` applied to `t[x,x']` returns the theorem

```
|-t[x,x'] = e''.
```

See also

`listLib.SCANL_CONV`, `listLib.FOLDL_CONV`, `listLib.FOLDR_CONV`, `listLib.list_FOLD_CONV`

scrub (Theory)

```
Theory.scrub : unit -> unit
```

Remove all out-of-date elements from the current theory segment.

An invocation `scrub()` goes through the current theory segment and removes all out-of-date elements.

Failure

Never fails.

Example

In the following, we define a concrete type and examine the current theory segment to see what consequences of this definition have been stored there. Then we delete the type, which turns all those consequences into garbage. An query, like `current_theorems`, shows that this garbage is not collected automatically. A manual invocation of `scrub` is necessary to show the true state of play.

```
- Hol_datatype `foo = A | B of 'a`;
<<HOL message: Defined type: "foo">>
> val it = () : unit

- current_theorems();
> val it =
    [("foo_induction", |- !P. P A /\ (!a. P (B a)) ==> !f. P f),
     ("foo_Axiom", |- !f0 f1. ?fn. (fn A = f0) /\ !a. fn (B a) = f1 a),
     ("foo_nchotomy", |- !f. (f = A) \/ ?a. f = B a),
     ("foo_case_cong",
      |- !M M' v f.
           (M = M') /\ ((M' = A) ==> (v = v')) /\
           (!a. (M' = B a) ==> (f a = f' a)) ==>
           (case v f M = case v' f' M')),
     ("foo_distinct", |- !a. ~(A = B a)),
     ("foo_11", |- !a a'. (B a = B a') = (a = a'))] : (string * thm) list

- delete_type "foo";
> val it = () : unit

- current_theorems();
> val it =
    [("foo_induction", |- !P. P A /\ (!a. P (B a)) ==> !f. P f),
     ("foo_Axiom", |- !f0 f1. ?fn. (fn A = f0) /\ !a. fn (B a) = f1 a),
     ("foo_nchotomy", |- !f. (f = A) \/ ?a. f = B a),
     ("foo_case_cong",
      |- !M M' v f.
           (M = M') /\ ((M' = A) ==> (v = v')) /\
```

```
             (!a. (M' = B a) ==> (f a = f' a)) ==>
             (case v f M = case v' f' M')),
       ("foo_distinct", |- !a. ~(A = B a)),
       ("foo_11", |- !a a'. (B a = B a') = (a = a'))] : (string * thm) list

- scrub();
> val it = () : unit

- current_theorems();
> val it = [] : (string * thm) list
```

When `export_theory` is called, it uses `scrub` to prepare the current segment for export. Users can also call `scrub` to find out what setting they are really working in.

See also

`Theory.uptodate_type`, `Theory.uptodate_term`, `Theory.uptodate_thm`, `Theory.delete_type`, `Theory.delete_const`

search_top_down (Cond_rewrite)

```
Cond_rewrite.search_top_down
 : (term -> term -> ((term # term) list # (type # type) list) list)
```

Search a term in a top-down fashion to find matches to another term.

`search_top_down tm1 tm2` returns a list of instantiations which make the whole or part of `tm2` match `tm1`. The first term should not have a quantifier at the outer most level. `search_top_down` first attempts to match the whole second term to `tm1`. If this fails, it recursively descend into the subterms of `tm2` to find all matches.

The length of the returned list indicates the number of matches found. An empty list means no match can be found between `tm1` and `tm2` or any subterms of `tm2`. The instantiations returned in the list are in the same format as for the function `match`. Each instantiation is a pair of lists: the first is a list of term pairs and the second is a list of type pairs. Either of these lists may be empty. The situation in which both lists are empty indicates that there is an exact match between the two terms, i.e., no instantiation is required to make the entire `tm2` or a part of `tm2` the same as `tm1`.

Failure

Never fails.

Example

```
#search_top_down "x = y:*" "3 = 5";;
[([("5", "y"); ("3", "x")], [(":num", ":*")])]
: ((term # term) list # (type # type) list) list

#search_top_down "x = y:*" "x =y:*";;
[([], [])] : ((term # term) list # (type # type) list) list

#search_top_down "x = y:*" "0 < p ==> (x <= p = y <= p)";;
[([("y <= p", "y"); ("x <= p", "x")], [(":bool", ":*")])]
: ((term # term) list # (type # type) list) list
```

The first example above shows the entire `tm2` matching `tm1`. The second example shows the two terms match exactly. No instantiation is required. The last example shows that a subterm of `tm2` can be instantiated to match `tm1`.

See also

`Db.match`

SEG_CONV (listLib)

```
listLib.SEG_CONV : conv
```

Computes by inference the result of taking a segment of a list.

For any object language list of the form "`[x0;...x(n-1)]`" , the result of evaluating

```
SEG_CONV "SEG m k [x0;...;x(n-1)]"
```

is the theorem

```
|- SEG m k [x0;...;x(n-1)] = [xk;...;x(m+k-1)]
```

Failure

`SEG_CONV tm` fails if `tm` is not in the form described above or the indexes `m` and `k` are not in the correct range, i.e., `m + k <= n`.

Example

Evaluating the expression

```
SEG_CONV “SEG 2 3[0;1;2;3;4;5]”;
```

returns the following theorem

```
|- SEG 2 3[0;1;2;3;4;5] = [3;4]
```

See also

`listLib.FIRSTN_CONV`, `listLib.LASTN_CONV`, `listLib.BUTFIRSTN_CONV`, `listLib.BUTLASTN_CONV`, `listLib.LAST_CONV`, `listLib.BUTLAST_CONV`

select (boolSyntax)

`boolSyntax.select : term`

Constant denoting Hilbert’s choice operator.

The ML variable `boolSyntax.select` is bound to the term `min$@`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universa`
`boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`,
`boolSyntax.negation`, `boolSyntax.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`

SELECT_CONV (Conv)

```
Conv.SELECT_CONV : conv
```

Eliminates an epsilon term by introducing an existential quantifier.

The conversion SELECT_CONV expects a boolean term of the form P[@x.P[x]/x], which asserts that the epsilon term @x.P[x] denotes a value, x say, for which P[x] holds. This assertion is equivalent to saying that there exists such a value, and SELECT_CONV applied to a term of this form returns the theorem |- P[@x.P[x]/x] = ?x. P[x].

Failure

Fails if applied to a term that is not of the form P[@x.P[x]/x].

Example

```
SELECT_CONV (Term `(@n. n < m) < m`);
val it = |- (@n. n < m) < m = (?n. n < m) : thm
```

Particularly useful in conjunction with CONV_TAC for proving properties of values denoted by epsilon terms. For example, suppose that one wishes to prove the goal

```
([0 < m], (@n. n < m) < SUC m)
```

Using the built-in arithmetic theorem

```
LESS_SUC  |- !m n. m < n ==> m < (SUC n)
```

this goal may be reduced by the tactic MATCH_MP_TAC LESS_SUC to the subgoal

```
([0 < m], (@n. n < m) < m)
```

This is now in the correct form for using CONV_TAC SELECT_CONV to eliminate the epsilon term, resulting in the existentially quantified goal

```
([0 < m], ?n. n < m)
```

which is then straightforward to prove.

See also

Drule.SELECT_ELIM, Drule.SELECT_INTRO, Drule.SELECT_RULE

SELECT_ELIM (Drule)

```
Drule.SELECT_ELIM : thm -> term * thm -> thm
```

Eliminates an epsilon term, using deduction from a particular instance.

SELECT_ELIM expects two arguments, a theorem th1, and a pair (v,th2): term * thm. The conclusion of th1 should have the form P($@ P), which asserts that the epsilon term $@ P denotes some value at which P holds. In th2, the variable v appears only in the assumption P v. The conclusion of the resulting theorem matches that of th2, and the hypotheses include the union of all hypotheses of the premises excepting P v.

```
 A1 |- P($@ P)     A2 u {P v} |- t
---------------------------------  SELECT_ELIM th1 (v,th2)
          A1 u A2 |- t
```

where v is not free in A2. The argument to P in the conclusion of th1 may actually be any term x. If v appears in the conclusion of th2, this argument x (usually the epsilon term) will NOT be eliminated, and the conclusion will be t[x/v].

Failure

Fails if the first theorem is not of the form A1 |- P x, or if the variable v occurs free in any other assumption of th2.

Example

If a property of functions is defined by:

```
INCR = |- !f. INCR f = (!t1 t2. t1 < t2 ==> (f t1) < (f t2))
```

The following theorem can be proved.

```
th1 = |- INCR(@f. INCR f)
```

Additionally, if such a function is assumed to exist, then one can prove that there also exists a function which is injective (one-to-one) but not surjective (onto).

```
th2 = [ INCR g ] |- ?h. ONE_ONE h /\ ~ONTO h
```

These two results may be combined using SELECT_ELIM to give a new theorem:

```
- SELECT_ELIM th1 (``g:num->num``, th2);
val it = |- ?h. ONE_ONE h /\ ~ONTO h : thm
```

This rule is rarely used. The equivalence of P($@ P) and $? P makes this rule fundamentally similar to the ?-elimination rule CHOOSE.

See also

Thm.CHOOSE, Conv.SELECT_CONV, Tactic.SELECT_ELIM_TAC, Drule.SELECT_INTRO, Drule.SELECT_RULE

SELECT_ELIM_TAC (Tactic)

```
Tactic.SELECT_ELIM_TAC : tactic
```

Eliminates a Hilbert-choice ("selection") term from the goal.

SELECT_ELIM_TAC searches the goal it is applied to for terms involving the Hilbert-choice operator, of the form @x. .... If such a term is found, then the tactic will produce a new goal, where the choice term has disappeared. The resulting goal will require the proof of the non-emptiness of the choice term's predicate, and that any possible choice of value from that predicate will satisfy the original goal.

Thus, SELECT_ELIM_TAC can be seen as a higher-order match against the theorem

```
|- !P Q. (?x. P x) /\ (!x. P x ==> Q x) ==> Q ($@ P)
```

where the new goal is the antecedent of the implication. (This theorem is SELECT_ELIM_THM, from theory bool.)

Example

When applied to this goal,

```
- SELECT_ELIM_TAC ([], ``(@x. x < 4) < 5``);
> val it = ([([], ``(?x. x < 4) /\ !x. x < 4 ==> x < 5``)], fn) :
    (term list * term) list * (thm list -> thm)
```

the resulting goal requires the proof of the existence of a number less than 4, and also that any such number is also less than 5.

Failure

Fails if there are no choice terms in the goal.

Comments

If the choice-term's predicate is everywhere false, goals involving such terms will be true only if the choice of term makes no difference at all. Such situations can be handled with the use of SPEC_TAC. Note also that the choice of select term to eliminate is made in an unspecified manner.

See also

Tactic.DEEP_INTRO_TAC, Drule.SELECT_ELIM, Drule.SELECT_INTRO, Drule.SELECT_RULE, Tactic.SPEC_TAC

SELECT_EQ (Drule)

```
Drule.SELECT_EQ : (term -> thm -> thm)
```

Applies epsilon abstraction to both terms of an equation.

Effects the extensionality of the epsilon operator @.

```
      A |- t1 = t2
------------------------  SELECT_EQ "x"      [where x is not free in A]
 A |- (@x.t1) = (@x.t2)
```

Failure

Fails if the conclusion of the theorem is not an equation, or if the variable `x` is free in `A`.

Example

Given a theorem which shows the equivalence of two distinct forms of defining the property of being an even number:

```
th = |- (x MOD 2 = 0) = (?y. x = 2 * y)
```

A theorem giving the equivalence of the epsilon abstraction of each form is obtained:

```
- SELECT_EQ (Term `x:num`) th;
> val it = |- (@x. x MOD 2 = 0) = (@x. ?y. x = 2 * y) : thm
```

See also

`Thm.ABS`, `Thm.AP_TERM`, `Drule.EXISTS_EQ`, `Drule.FORALL_EQ`, `Conv.SELECT_CONV`, `Drule.SELECT_ELIM`, `Drule.SELECT_INTRO`

SELECT_INTRO (Drule)

```
Drule.SELECT_INTRO : (thm -> thm)
```

Introduces an epsilon term.

`SELECT_INTRO` takes a theorem with an applicative conclusion, say `P x`, and returns a theorem with the epsilon term `$@ P` in place of the original operand `x`.

```
  A |- P x
--------------  SELECT_INTRO
 A |- P($@ P)
```

The returned theorem asserts that `$@ P` denotes some value at which `P` holds.

Failure

Fails if the conclusion of the theorem is not an application.

Example

Given the theorem

```
th1 = |- (\n. m = n)m
```

applying `SELECT_INTRO` replaces the second occurrence of `m` with the epsilon abstraction of the operator:

```
- val th2 = SELECT_INTRO th1;
val th2 = |- (\n. m = n)(@n. m = n) : thm
```

This theorem could now be used to derive a further result:

```
- EQ_MP (BETA_CONV(concl th2)) th2;
val it = |- m = (@n. m = n) : thm
```

See also

`Thm.EXISTS`, `Conv.SELECT_CONV`, `Drule.SELECT_ELIM`, `Drule.SELECT_RULE`

SELECT_LT (Tactical)

```
Tactical.SELECT_LT : tactic -> list_tactic
```

Applies a tactic to the all the goals in the goal-list for which the tactic succeeds.

Given a list of goals `gl`, an application of `SELECT_LT tac` to `gl` will try to apply `tac` to each goal in `gl` in turn. If no goal lets `tac` succeed, the goal state remains unchanged. Otherwise, the goals for which `tac` succeeds will generate (possibly empty) list(s) of new sub-goals. These new sub-goals are pushed onto the front of the rest of `gl`.

Failure

The application of SELECT_LT to a tactic never fails. The resulting list-tactic also never fails.

Example

```
> SELECT_LT CONJ_TAC [([], “r ∧ s”), ([], “p ⇒ q”), ([“a ∨ b”], “p ∧ q”)]
val it =
   ([([], “r”), ([], “s”), ([“a ∨ b”], “p”), ([“a ∨ b”], “q”), ([], “p ⇒ q”)],
    fn): goal list * list_validation
```

See also

Tactical.SELECT_LT_THEN, Tactical.FIRST_LT, Tactical.THEN_LT, Tactical.HEADGOAL

SELECT_LT_THEN (Tactical)

```
Tactical.SELECT_LT_THEN : tactic -> tactic -> list_tactic
```

Applies the first tactic to all goals in the goal-list for which the tactic succeeds. Then applies the second tactic to the goals resulting from the first tactic.

Given a list of goals gl, an application of SELECT_LT tac1 tac2 to gl will try to apply tac1 to each goal in gl in turn. If no goal lets tac1 succeed, the goal state remains unchanged. Otherwise, the goals for which tac1 succeeds will generate (possibly empty) list(s) of new sub-goals. tac2 will be applied to each of these new sub-goals. The resulting subgoals after applying tac2 are pushed onto the front of the rest of gl.

Failure

The application of SELECT_LT_THEN to tactic arguments tac1, tac2 never fails. The resulting list-tactic fails only when tac2 fails on a subgoal produced by applying tac1 to the current goals.

Example

```
> SELECT_LT_THEN DISJ1_TAC ALL_TAC
  [([], “T ∨ s”), ([], “p ⇒ q”), ([“a ∨ b”], “p ∨ q”)]
val it = ([([], “T”), ([“a ∨ b”], “p”), ([], “p ⇒ q”)], fn):
   goal list * list_validation

> SELECT_LT_THEN DISJ1_TAC (ACCEPT_TAC TRUTH)
  [([], “T ∨ s”), ([], “p ⇒ q”), ([“a ∨ b”], “p ∨ q”)]
Exception- HOL_ERR (at Tactical.SELECT_LT_THEN: Could not apply second tactic) raised
```

See also

`Tactical.SELECT_LT`, `Tactical.FIRST_LT`, `Tactical.THEN_LT`, `Tactical.HEADGOAL`

SELECT_RULE (Drule)

```
Drule.SELECT_RULE : thm -> thm
```

Introduces an epsilon term in place of an existential quantifier.

The inference rule `SELECT_RULE` expects a theorem asserting the existence of a value `x` such that `P` holds. The equivalent assertion that the epsilon term `@x.P` denotes a value of `x` for which `P` holds is returned as a theorem.

```
      A |- ?x. P
  ------------------  SELECT_RULE
   A |- P[(@x.P)/x]
```

Failure

Fails if applied to a theorem the conclusion of which is not existentially quantified.

Example

The axiom `INFINITY_AX` in the theory `ind` is of the form:

```
|- ?f. ONE_ONE f /\ ~ONTO f
```

Applying SELECT_RULE to this theorem returns the following.

```
- SELECT_RULE INFINITY_AX;
> val it =
  |- ONE_ONE (@f. ONE_ONE f /\ ~ONTO f) /\ ~ONTO @f. ONE_ONE f /\ ~ONTO f
  : thm
```

May be used to introduce an epsilon term to permit rewriting with a constant defined using the epsilon operator.

See also

Thm.CHOOSE, Conv.SELECT_CONV, Drule.SELECT_ELIM, Drule.SELECT_INTRO

selectDB (DB)

```
DB.selectDB : DB.selector list -> DB.public_data list
```

Searches the theorem database with multiple conjoined selectors

A call to DB.selectDB [sel1, sel2, ..., seln] returns a list of theorems from the theorem database that match all of the criteria embodied by sel1, sel2, etc. The selectors are of three different forms:

```
SelTM term | SelNM string | SelTHY string
```

The selector SelTM t matches any theorem that has a sub-term matching the term t. The selector SelNM s matches any theorem whose name matches the string s, using the regular-expression-like matching syntax described in the documentation for DB.find. Finally, SelThy thy matches a theorem if that theorem comes from theory thy.

Failure

Never fails.

Example

```
> DB.selectDB [SelTM "_ /\ _", SelTHY "bool", SelNM "ASSOC"];
val it =
   [(("bool", "CONJ_ASSOC"),
     (⊢ ∀t1 t2 t3. t1 ∧ t2 ∧ t3 ⇔ (t1 ∧ t2) ∧ t3, Thm,
      Located
       {exact = true, linenum = 766, scriptpath =
        "$(HOLDIR)/src/bool/boolScript.sml"}))]: public_data list
```

Comments

Allows for more powerful searches than other entrypoints in `DB`.

See also

`DB.find`, `DB.match`

sep_type_unify (boolSyntax)

```
boolSyntax.sep_type_unify : hol_type -> hol_type ->
        (hol_type,hol_type) subst * (hol_type,hol_type) subst
```

Calculates a pair of substitutions `(theta1, theta2)` such that instantiating the first argument with `theta1` equals the second argument instatiated with `theta2`.

If `sep_type_unify ty1 ty2` succeeds, then

```
    type_subst (sep_type_unify ty1 ty2 |> fst) ty1 =
      type_subst (sep_type_unify ty1 ty2 |> snd) ty2
```

Failure

If no such substitution can be found.

Example

```
> let val alpha_list = Type `:'a list`
  in
    sep_type_unify alpha alpha_list
  end;
val it = ([{redex = “:α”, residue = “:α list”}], []):
   (hol_type, hol_type) Lib.subst * (hol_type, hol_type) Lib.subst

> let val ty1 = Type `:'a -> 'b -> 'b`
     val ty2 = Type `:'a list -> 'b list -> 'a list`
  in
   sep_type_unify ty1 ty2
  end;
val it =
   ([{redex = “:β”, residue = “:α list”},
     {redex = “:α”, residue = “:α list”}], [{redex = “:β”, residue = “:α”}]):
   (hol_type, hol_type) Lib.subst * (hol_type, hol_type) Lib.subst
```

Note that in these examples, `type_unify` would fail due to an occurs check:

```
> let val ty1 = Type `:'a -> 'b -> 'b`
    val ty2 = Type `:'a list -> 'b list -> 'a list`
  in
  type_unify ty1 ty2
  end;
Exception- HOL_ERR (at boolSyntax.type_unify: occurs check) raised
```

Comments

`sep_type_unify` is similar to `type_unify`, but does not run into problems with occurs checks. It first renames all type variables, then attempt to unify the argument types, returning two separate substitutions as a result.

See also

`boolSyntax.type_unify`, `Type.type_subst`, `Term.inst`

set_backup (proofManagerLib)

```
proofManagerLib.set_backup : int -> unit
```

Limits the number of proof states saved on the subgoal package backup list.

The assignable variable `set_backup` is initially set to 12. Its value is one less than the maximum number of proof states that may be saved on the backup list. Adding a new proof state (by, for example, a call to `expand`) after the maximum is reached causes the earliest proof state on the list to be discarded. For a description of the subgoal package, see `set_goal`.

Example

```
> set_backup 0;
val it = (): unit

> g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;
val it =
   Proof manager status: 5 proofs.
   5. Completed goalstack:  [.] ⊢ q

   4. Completed goalstack: ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

   3. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

   2. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

        Current goal:
        HD [1; 2; 3] = 1

   1. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]
```

```
> e CONJ_TAC;
OK..
2 subgoals:
val it =

   TL [1; 2; 3] = [2; 3]

   HD [1; 2; 3] = 1

> e (REWRITE_TAC[listTheory.HD]);
OK..

Goal proved.
⊢ HD [1; 2; 3] = 1

Remaining subgoals:
val it =

   TL [1; 2; 3] = [2; 3]

> b();
val it =

   TL [1; 2; 3] = [2; 3]

   HD [1; 2; 3] = 1

> b();
val it =
   Initial goal:

   HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]: proof
```

See also

proofManagerLib.set_goal, proofManagerLib.restart, proofManagerLib.backup, proofManagerLib.redo, proofManagerLib.restore, proofManagerLib.save, proofManagerLib.s backup, proofManagerLib.expand, proofManagerLib.expandf, proofManagerLib.p, proofManagerLib.top_thm, proofManagerLib.top_goal

set_diff (Lib)

```
Lib.set_diff : ''a list -> ''a list -> ''a list
```

Computes the set-theoretic difference of two 'sets'.

`set_diff l1 l2` returns a list consisting of those elements of `l1` that do not appear in `l2`. It is identical to `Lib.subtract`.

Failure

Never fails.

Example

```
> set_diff [] [1,2];
val it = []: int list

> set_diff [1,2,3] [2,1];
val it = [3]: int list
```

Comments

The order in which the elements occur in the resulting list should not be depended upon.

A high-performance implementation of finite sets may be found in structure `HOLset`.

ML equality types are used in the implementation of `union` and its kin. This limits its applicability to types that allow equality. For other types, typically abstract ones, use the 'op_' variants.

See also

`Lib.op_set_diff`, `Lib.subtract`, `Lib.mk_set`, `Lib.set_eq`, `Lib.union`, `Lib.intersect`

set_eq (Lib)

```
Lib.set_eq : ''a list -> ''a list -> bool
```

Tells whether two lists have the same elements.

An application `set_eq l1 l2` returns `true` just in case `l1` and `l2` are permutations of each other when duplicate elements within each list are ignored.

Failure

Never fails.

Example

```
> set_eq [1,2,1] [1,2,2,1];
val it = true: bool

> set_eq [1,2,1] [2,1];
val it = true: bool
```

Comments

A high-performance implementation of finite sets may be found in structure `HOLset`.

ML equality types are used in the implementation of `set_eq` and its kin. This limits its applicability to types that allow equality. For other types, typically abstract ones, use the 'op_' variants.

See also

`Lib.intersect`, `Lib.union`, `Lib.U`, `Lib.mk_set`, `Lib.mem`, `Lib.insert`, `Lib.set_diff`

set_fixity (Parse)

```
Parse.set_fixity : string -> fixity -> unit
```

Allows the fixity of tokens to be updated.

The `set_fixity` function is used to change the fixity of single tokens. It implements this functionality rather crudely. When called on to set the fixity of `t` to `f`, it removes all rules mentioning `t` from the global (term) grammar, and then adds a new rule to the grammar. The new rule maps occurrences of `t` with the given fixity to terms of the same name.

Failure

This function fails if the new fixity causes a clash with existing rules, as happens if the precedence level of the specified fixity is already taken by rules using a fixity of a different type. Even if the application of `set_fixity` succeeds, it may cause the next subsequent application of the `Term` parsing function to complain about precedence conflicts in the operator precedence matrix. These problems may cause the parser to behave oddly on terms involving the token whose fixity was set. Excessive parentheses will usually cure even these problems.

Example

After a new constant is defined, `set_fixity` can be used to give it an appropriate parse status:

```
- val thm = Psyntax.new_recursive_definition
               prim_recTheory.num_Axiom "f"
               (Term`(f 0 n = n) /\ (f (SUC n) m = SUC (SUC (f n m)))`);
> val thm =
    |- (!n. f 0 n = n) /\ !n m. f (SUC n) m = SUC (SUC (f n m))
    : thm
- set_fixity "f" (Infixl 500);
> val it = () : unit
- thm;
> val it =
    |- (!n. 0 f n = n) /\ !n m. SUC n f m = SUC (SUC (n f m)) : thm
```

The same function can be used to alter the fixities of existing constants:

```
- val t = Term`2 + 3 + 4 - 6`;
> val t = `2 + 3 + 4 - 6` : term
- set_fixity "+" (Infixr 501);
> val it = () : unit
- t;
```

```
> val it = `(2 + 3) + 4 - 6` : term
- dest_comb (Term`3 - 1 + 2`);
> val it = (`$- 3`, `1 + 2`) : term * term
```

Comments

This function is of no use if multiple-token rules (such as those for conditional expressions) are desired, or if the token does not correspond to the name of the constant or variable that is to be produced. (For the latter case, use `set_mapped_fixity`.)

As with other functions in the `Parse` structure, there is a companion `temp_set_fixity` function, which has the same effect on the global grammar, but which does not cause this effect to persist when the current theory is exported.

See also

`Parse.add_rule`, `Parse.add_infix`, `Parse.remove_rules_for_term`, `Parse.remove_termtok`, `Parse.set_mapped_fixity`

set_goal (proofManagerLib)

```
proofManagerLib.set_goal : term list * term -> unit
```

Initializes the subgoal package with a new goal.

The function `set_goal` initializes the subgoal management package. A proof state of the package consists of either a goal stack and a justification stack if a proof is in progress, or a theorem if a proof has just been completed. `set_goal` sets a new proof state consisting of an empty justification stack and a goal stack with the given goal as its sole goal. The goal is printed.

Failure

Fails unless all terms in the goal are of type `bool`.

Example

```
> set_goal([], Term `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`);
val it =
   Proof manager status: 6 proofs.
   6. Completed goalstack:  [.] ⊢ q

   5. Completed goalstack: ⊢ HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

   4. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

   3. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

        Current goal:
        HD [1; 2; 3] = 1

   2. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]

   1. Incomplete goalstack:
        Initial goal:
        HD [1; 2; 3] = 1 ∧ TL [1; 2; 3] = [2; 3]
```

Starting an interactive proof session with the subgoal package.

The subgoal package implements a simple framework for interactive goal-directed proof. When conducting a proof that involves many subgoals and tactics, the user must keep track of all the justifications and compose them in the correct order. While this is feasible even in large proofs, it is tedious. The subgoal package provides a way of building and traversing the tree of subgoals top-down, stacking the justifications and applying them properly.

The package maintains a proof state consisting of either a goal stack of outstanding goals and a justification stack, or a theorem. Tactics are used to expand the current goal (the one on the top of the goal stack) into subgoals and justifications. These are pushed onto the goal stack and justification stack, respectively, to form a new proof state. Several preceding proof states are saved and can be returned to if a mistake is made in the

proof. The goal stack is divided into levels, a new level being created each time a tactic is successfully applied to give new subgoals. Alternatively a list-tactic can process the entire list of goals of the current level to change that level (rather than creating a new level). The subgoals of the current level may be considered in any order. Levels of the goal stack may be collapsed so that subgoals of a previous level appear as part of of the current level.

If a tactic solves the current goal (returns an empty subgoal list), then its justification is used to prove a corresponding theorem. This theorem is then incorporated into the justification of the parent goal. If the subgoal was the last subgoal of the level, the level is removed and the parent goal is proved using its (new) justification. This process is repeated until a level with unproven subgoals is reached. The next goal on the goal stack then becomes the current goal. If all the subgoals are proved, the resulting proof state consists of the theorem proved by the justifications. This theorem may be accessed and saved.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.s` `backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.expand_-` `list`, `proofManagerLib.flatn`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLi` `goal`

set_implicit_rewrites (Rewrite)

```
Rewrite.set_implicit_rewrites: rewrites -> unit
```

Allows the user to control the built-in database of simplifications used in rewriting.

Failure

Never fails.

See also

`Rewrite.empty_rewrites`, `Rewrite.add_rewrites`

SET_INDUCT_TAC (pred_setLib)

```
pred_setLib.SET_INDUCT_TAC : tactic
```

Tactic for induction on finite sets.

SET_INDUCT_TAC is an induction tacic for proving properties of finite sets. When applied to a goal of the form

```
!s. FINITE s ==> P[s]
```

SET_INDUCT_TAC reduces this goal to proving that the property `\s.P[s]` holds of the empty set and is preserved by insertion of an element into an arbitrary finite set. Since every finite set can be built up from the empty set `{}` by repeated insertion of values, these subgoals imply that the property `\s.P[s]` holds of all finite sets.

The tactic specification of SET_INDUCT_TAC is:

```
              A ?- !s. FINITE s ==> P
=========================================================  SET_INDUCT_TAC
  A |- P[{{}}/s]
  A u {FINITE s', P[s'/s], ~e IN s'} ?- P[e INSERT s'/s]
```

where `e` is a variable chosen so as not to appear free in the assumptions `A`, and `s'` is a primed variant of `s` that does not appear free in `A` (usually, `s'` is just `s`).

Failure

SET_INDUCT_TAC `(A,g)` fails unless `g` has the form `!s. FINITE s ==> P`, where the variable `s` has type `ty->bool` for some type `ty`.

set_known_constants (Parse)

```
Parse.set_known_constants : string list -> unit
```

Specifies the list of names that should be parsed as constants.

One of the final phases of parsing is the resolution of free names in putative terms as either variables, constants or overloaded constants. If such a free name is not overloaded, then the list of known constants is consulted to determine whether or not to treat it as a constant. If the name is not present in the list, then it will be treated as a free variable.

Failure

Never fails. If a name is specified in the list of constants that is not in fact a constant, a warning message is printed, and that name is ignored.

Example

```
> known_constants();
val it =
   ["", "!", "##", "%%", "&", "()", "*", "**", "*,", "+", "++", "+++", ",",
    "-", "-->", "/\\", "0", ":-", ":>", "<", "<<=", "<=", "<=/=>", "<=>",
    "<>", "=", "===>", "==>", ">", ">=", "?", "?!", "?!!", "@", "ABS_DIFF",
    "ABS_num", "ABS_prod", "ABS_sum", "ALL_DISTINCT", "ALL_EL", "AND_EL",
    "APPEND", "APPLICATIVE_FAPPLY", "APPLY_REDUNDANT_ROWS_INFO", "ARB",
    "ASM_MARKER", "ASSOC", "Abbrev", "BIGINTER", "BIGUNION", "BIJ", "BIT1",
    "BIT2", "BOUNDED", "BUTFIRSTN", "BUTLAST", "BUTLASTN", "CARD",
    "CEILING_DIV", "CEILING_MOD", "CHOICE", "COMM", "COMPL", "COND", "CONS",
    "COUNTABLE", "COUNT_LIST", "COUNT_LIST_AUX", "CR", "CROSS", "CURRY",
    "Case", "Cong", "DATATYPE", "DELETE", "DELETE_ELEMENT", "DFUNSET",
    "DIFF", "DISJOINT", "DIV", "DIV2", "DIVMOD", "DROP", "EL", "ELL",
    "EMPTY", "EMPTY_REL", "EQC", "EQUIV", "EVEN", "EVERY", "EVERY2",
    "EVERYi", "EXISTS", "EXP", "EXTENSIONAL", "EXT_POINT", "Exclude",
    "ExcludeFrag", "F", ...]: string list
> Term`p /\ q`;
val it = “p ∧ q”: term
> set_known_constants (Lib.subtract (known_constants()) ["/\\"]);
val it = (): unit
> Term`p /\ q`;
val it = “p ∧ q”: term
> strip_comb it;
val it = (“$/\”, [“p”, “q”]): term * term list
> dest_var (#1 it);
val it = ("/\\", “:α -> β -> γ”): string * hol_type
```

When writing library code that calls the parser, it can be useful to know exactly what constants the parser will "recognise".

Comments

This function does not affect the contents of a theory. A constant made invisible using this call is still really present in the theory; it is just harder to find.

See also

`Parse.hidden`, `Parse.hide`, `Parse.known_constants`, `Parse.reveal`

set_mapped_fixity (Parse)

```
Parse.set_mapped_fixity :
  {tok : string, term_name : string, fixity : fixity} -> unit
```

Allows the fixity of tokens to be updated.

The `set_mapped_fixity` function is used to change the fixity of a single token, simultaneously mapping forms using that token name to a different name. Apart from the additional `term_name` field, the behaviour is similar to that of `set_fixity`.

Failure

This function fails if the new fixity causes a clash with existing rules, as happens if the precedence level of the specified fixity is already taken by rules using a fixity of a different type. Even if the application of `set_mapped_fixity` succeeds, it may cause the next subsequent application of the `Term` parsing function to complain about precedence conflicts in the operator precedence matrix. These problems may cause the parser to behave oddly on terms involving the token whose fixity was set. Excessive parentheses will usually cure even these problems.

Comments

This function is of no use if multiple-token rules (such as those for conditional expressions) are desired.

As with other functions in the Parse structure, there is a companion temp_set_mapped_fixity function, which has the same effect on the global grammar, but which does not cause this effect to persist when the current theory is exported.

See also

Parse.add_rule, Parse.set_fixity

set_message (Feedback)

```
set_message : string -> hol_error -> hol_error
```

Overwrite message component of a hol_error value.

Example

```
> set_message "BIG problem!"
    (mk_hol_error "Foo" "bar" locn.Loc_Unknown "bad input");
val it = at Foo.bar: BIG problem!: hol_error
```

Failure

Never fails.

See also

Feedback, Feedback.mk_hol_error, Feedback.message_of, Feedback.set_top_function

set_MLname (Theory)

```
Theory.set_MLname : string -> string -> unit
```

Change the name attached to an element of the current theory.

It can happen that an axiom, definition, or theorem gets stored in the current theory segment under a name that wouldn't be a suitable ML identifier. For example, some advanced definition mechanisms in HOL automatically construct names to bind the results of making a definition. In some cases, particularly when symbolic identifiers are involved, a binding name can be generated that is not a valid ML identifier.

In such cases, we don't want to fail and lose the work done by the definition mechanism. Instead, when `export_theory` is invoked, all names binding axioms, definitions, and theorems are examined to see if they are valid ML identifiers. If not, an informative error message is generated, and it is up to the user to change the names in the offending bindings. The function `set_MLname s1 s2` will replace a binding with name `s1` by one with name `s2`.

Failure

Never fails, although will give a warning if `s1` is not the name of a binding in the current theory segment.

Example

We inductively define a predicate telling if a number is odd in the following. The name is admittedly obscure, however it illustrates our point.

```
- Hol_reln `(%% 1) /\ (!n. %% n ==> %% (n+2))`;
> val it =
    (|- %% 1 /\ !n. %% n ==> %% (n + 2),
     |- !%%'. %%' 1 /\ (!n. %%' n ==> %%' (n + 2)) ==> !a0. %% a0 ==> %%' a0,
     |- !a0. %% a0 = (a0 = 1) \/ ?n. (a0 = n + 2) /\ %% n) : thm * thm * thm

- export_theory();
<<HOL message: The following ML binding names in the theory to be exported:
"%%_rules", "%%_ind", "%%_cases"
 are not acceptable ML identifiers.
   Use `set_MLname <bad> <good>' to change each name.>>
! Uncaught exception:
! HOL_ERR

- (set_MLname "%%_rules" "odd_rules";   (* OK, do what it says *)
   set_MLname "%%_ind"   "odd_ind";
```

```
   set_MLname "%%_cases" "odd_cases");
> val it = () : unit

- export_theory();
Exporting theory "scratch" ... done.
> val it = () : unit
```

Comments

The definition principles that currently have the potential to make problematic bindings are `Hol_datatype` and `Hol_reln`.

It is slightly awkward to have to repair the names in a post-hoc fashion. It is probably simpler to proceed by using alphanumeric names when defining constants, and to use overloading to get the desired name.

See also

`bossLib.Hol_reln`, `bossLib.Hol_datatype`, `Theory.export_theory`, `Theory.current_definitions`, `Theory.current_theorems`, `Theory.current_axioms`, `DB.thy`, `DB.dest_theory`

set_prover (Tactical)

```
Tactical.set_prover : (goal * tactic -> thm) -> unit
```

Specifies the function to be used by `prove_goal` and `prove`.

A call to `set_prover f` sets the function used by `prove_goal` and `prove` to be `f`. The initial value is `TAC_PROOF`. The `prove` function takes a single term `t` as argument and passes this along with an empty list of assumptions to `f`.

Failure

Never fails.

Comments

This function is used by `Holmake` when called with the `--noqof` and `--fast` options to set the prove function to one that will call `cheat` as appropriate (if a tactic otherwise fails, or in all cases, respectively).

See also

`Tactical.prove`, `Tactical.prove_goal`.

SET_RULE (bossLib)

```
bossLib.SET_RULE : term -> thm
```

Automatically prove a set-theoretic theorem by reduction to FOL.

An application `DECIDE M`, where `M` is a set-theoretic term, attempts to automatically prove `M` by reducing basic set-theoretic operators (`IN`, `SUBSET`, `PSUBSET`, `INTER`, `UNION`, `INSERT`, `DELETE`, `REST`, `DISJOINT`, `BIGINTER`, `BIGUNION`, `IMAGE`, `SING` and `GSPEC`) in `M` to their definitions in first-order logic. With `SET_RULE`, many simple set-theoretic results can be directly proved without finding needed lemmas in `pred_setTheory`.

Example

```
> SET_RULE ``!s t c. DISJOINT s t ==> DISJOINT (s INTER c) (t INTER c)``;
metis: r[+0+5]+0+0+0+0+1#
val it = ⊢ ∀s t c. DISJOINT s t ⇒ DISJOINT (s ∩ c) (t ∩ c): thm
```

Failure

Fails if the underlying resolution machinery used by `METIS_TAC` cannot prove the goal, e.g. when there are other set operators in the term.

Comments

`SET_RULE` calls `SET_TAC` without extra lemmas.

See also

bossLib.SET_TAC, bossLib.ASM_SET_TAC, bossLib.METIS_TAC

SET_SPEC_CONV (pred_setLib)

```
pred_setLib.SET_SPEC_CONV : conv
```

Axiom-scheme of specification for set abstractions.

The conversion SET_SPEC_CONV expects its term argument to be an assertion of the form t IN {E | P}. Given such a term, the conversion returns a theorem that defines the condition under which this membership assertion holds. When E is just a variable v, the conversion returns:

```
|- t IN {v | P} = P[t/v]
```

and when E is not a variable but some other expression, the theorem returned is:

```
|- t IN {E | P} = ?x1...xn. (t = E) /\ P
```

where x1, ..., xn are the variables that occur free both in the expression E and in the proposition P.

Example

```
> pred_setLib.SET_SPEC_CONV ``12 IN {n | n > N}``;
val it = ⊢ 12 ∈ {n | n > N} ⇔ 12 > N: thm

> pred_setLib.SET_SPEC_CONV ``p IN {(n,m) | n < m}``;
val it = ⊢ p ∈ {(n,m) | n < m} ⇔ ∃n m. p = (n,m) ∧ n < m: thm
```

Failure

Fails if applied to a term that is not of the form t IN {E | P}.

SET_TAC (bossLib)

```
bossLib.SET_TAC : thm list -> tactic
```

Tactic to automate some routine pred_set theory by reduction to FOL, using the given theorems as additional assumptions in the search.

SET_TAC reduces basic set-theoretic operators (IN, SUBSET, PSUBSET, INTER, UNION, INSERT, DELETE, REST, DISJOINT, BIGINTER, BIGUNION, IMAGE, SING and GSPEC) in the goal to their definitions in first-order logic (FOL) and then call METIS_TAC to solve it. With SET_TAC, many simple set-theoretic results can be directly proved without finding needed lemmas in pred_setTheory.

Failure

Fails if the underlying resolution machinery (METIS_TAC) cannot prove the goal, or the supplied theorems are not enough for the FOL reduction, e.g., when there are other set-theoretic operators in the goal.

Example

A simple theorem about disjoint sets:

```
Theorem DISJOINT_RESTRICT_L :
  !s t c. DISJOINT s t ==> DISJOINT (s INTER c) (t INTER c)
Proof SET_TAC []
QED
```

SET_TAC can only progress the goal to a successful proof of the (whole) goal or not at all. SET_RULE can be used to prove an intermediate set-theoretic lemma (there is no way to provide extra lemmas, however).

Comments

The assumptions of a goal are ignored when SET_TAC is applied. To include assumptions use ASM_SET_TAC.

See also

bossLib.ASM_SET_TAC, bossLib.SET_RULE, bossLib.METIS_TAC

set_top_function (Feedback)

```
set_top_function : string -> hol_error -> hol_error
```

Overwrite last function name added to a hol_error.

An invocation set_top_function holerr replaces the function name of the top origin element held in holerr.

Example

```
> set_top_function "new_name"
    (mk_hol_error "Foo" "bar" locn.Loc_Unknown "bad input");
val it = at Foo.new_name: bad input: hol_error
```

Failure

The call fails if the origin stack of the hol_error is empty.

See also

Feedback, Feedback.mk_hol_error, Feedback.top_function_of, Feedback.set_message

set_trace (Feedback)

```
Feedback.set_trace : string -> int -> unit
```

Set a tracing level for a registered trace.

Invoking `set_trace n i` sets the level of the tracing mechanism registered under `n` to be `i`. These settings control the verboseness of various tools within the system. This can be useful both when debugging proofs (with the simplifier for example), and also as a guide to how an automatic proof is proceeding (with `mesonLib` for example).

There is no single interpretation of what activity a tracing level should evoke: each tool registered for tracing can treat its trace level in its own way.

Failure

A call to `set_trace n i` fails if `n` has not previously been registered via `register_trace`. It also fails if `i` is less than zero, or if it is greater than the trace's specified maximum value.

Example

```
- set_trace "Rewrite" 1;

- PURE_REWRITE_CONV [AND_CLAUSES] (Term `x /\ T /\ y`);

<<HOL message: Rewrite:
|- T /\ y = y.>>

> val it = |- x /\ T /\ y = x /\ y : thm
```

See also

`Feedback`, `Feedback.register_trace`, `Feedback.reset_trace`, `Feedback.reset_traces`, `Feedback.trace`, `Feedback.traces`

SF (bossLib)

```
bossLib.SF : ssfrag -> thm
```

Presents a simpset fragment as a theorem for inclusion in simplification

A call to `SF sfrag` creates a theorem that encodes (by way of an indirection through a global register of fragments) the simpset fragment `sfrag`. If this theorem is then passed to a simplification tactic (or conversion), the simplification tactic will add the given fragment to the simpset underpinning the simplification.

Failure

Fails if the given fragment doesn't have a name.

Comments

If the given fragment has a name, but has not been previously registered, it is registered at the time the simplification tactic or conversion is called. Given that this registration probably happens as part of a script's execution, this registration is unlikely to persist.

Example

```
> SIMP_CONV bool_ss [SF ETA_ss] “P (λx. f x) ∧ Q”;
val it = ⊢ P (λx. f x) ∧ Q ⇔ P f ∧ Q: thm

> simp[SF ETA_ss] ([], “P (λx. f x) ∧ Q”);
val it = ([([], “P f ∧ Q”)], fn): goal list * validation
```

See also

`simpLib.AC`, `simpLib.Cong`, `simpLib.register_frag`

show_numeral_types (Parse)

```
Globals.show_numeral_types : bool ref
```

A flag which causes numerals to be printed with suffix annotation when true.

This flag controls the pretty-printing of numeral forms that have been added to the global grammar with the function `add_numeral_form`. If the flag is true, then all numeric values are printed with the single-letter suffixes that identify which type the value is.

Failure

Never fails, as it is just an SML value.

Example

```
> Term`~3`;
val it = “-3”: term

> show_numeral_types := true;
val it = (): unit

> Term`~3`;
val it = “-3i”: term
```

Can help to disambiguate terms involving numerals.

See also

`Parse.add_numeral_form`, `Globals.show_types`

show_tags (Globals)

```
Globals.show_tags : bool ref
```

Flag for controlling display of tags in theorem prettyprinter.

The flag `show_tags` controls the display of values of type `thm`. When set to `true`, the tag attached to a theorem will be printed when the theorem is printed. When set to `false`, no indication of the presence or absence of a tag will be displayed.

Example

```
> show_tags := false;
val it = (): unit
```

```
> pairTheory.PAIR_MAP_THM;
val it = ⊢ ∀f g x y. (f ## g) (x,y) = (f x,g y): thm

> mk_thm ([],F);
val it = ⊢ F: thm

> show_tags := true;
val it = (): unit

> pairTheory.PAIR_MAP_THM;
val it =
   [oracles: DISK_THM] [axioms: ] [] ⊢ ∀f g x y. (f ## g) (x,y) = (f x,g y):
   thm

> mk_thm ([],F);
val it = [oracles: MK_THM] [axioms: ] [] ⊢ F: thm
```

Comments

The initial value of `show_tags` is `false`.

See also

`Thm.tag`, `Thm.mk_oracle_thm`, `Thm.mk_thm`

show_types (Globals)

```
Globals.show_types : bool ref
```

Flag controlling printing of HOL types (in terms).

Normally HOL types in terms are not printed, since this makes terms hard to read. Type printing is enabled by `show_types := true`, and disabled by `show_types := false`. When printing of types is enabled, not all variables and constants are annotated with a type. The intention is to provide sufficient type information to remove any ambiguities without swamping the term with type information.

Failure

Never fails.

Example

```
> BOOL_CASES_AX;;
val it = ⊢ ∀t. (t ⇔ T) ∨ (t ⇔ F): thm

> show_types := true;
val it = (): unit

> BOOL_CASES_AX;;
val it = ⊢ ∀(t :bool). (t ⇔ T) ∨ (t ⇔ F): thm
```

Comments

It is possible to construct an abstraction in which the bound variable has the same name but a different type to a variable in the body. In such a case the two variables are considered to be distinct. Without type information such a term can be very misleading, so it might be a good idea to provide type information for the free variable whether or not printing of types is enabled.

See also

`Parse.print_term`

SIMP_CONV (bossLib)

```
bossLib.SIMP_CONV : simpset -> thm list -> conv
```

Also exported as `simpLib.SIMP_CONV`.

Applies a simpset and a list of rewrite rules to simplify a term.

`SIMP_CONV` is the fundamental engine of the HOL simplification library. It repeatedly applies the transformations included in the provided simpset (which is augmented with

the given rewrite rules) to a term, ultimately yielding a theorem equating the original term to another.

Values of the `simpset` type embody a suite of different transformations that might be applicable to given terms. These "transformational components" are rewrites, conversions, AC-rules, congruences, decision procedures and a filter, which is used to modify the way in which rewrite rules are added to the simpset. The exact types for these components, known as simpset fragments, and the way they can be combined to create simpsets is given in the reference entry for `SSFRAG`.

Rewrite rules are used similarly to the way in they are used in the rewriting system (`REWRITE_TAC` et al.). These are equational theorems oriented to rewrite from left-hand-side to right-hand-side. Further, `SIMP_CONV` handles obvious problems. If a rewrite rule is of the general form `[...] |- x = f x`, then it will be discarded, and a message is printed to this effect. On the other hand, if the right-hand-side is a permutation of the pattern on the left, as in `|- x + y = y + x` and `|- x INSERT (y INSERT s) = y INSERT (x INSERT s)`, then such rules will only be applied if the term to which they are being applied is strictly reduced according to some term ordering.

Rewriting is done using a form of higher-order matching, and also uses conditional rewriting. This latter means that theorems of the form `|- P ==> (x = y)` can be used as rewrites. If a term matching `x` is found, the simplifier will attempt to satisfy the side-condition `P`. If it is able to do so, then the rewriting will be performed. In the process of attempting to rewrite `P` to true, further side conditions may be generated. The simplifier limits the size of the stack of side conditions to be solved (the reference variable `Cond_rewr.stack_limit` holds this limit), so this will not introduce an infinite loop.

Rewrite rules can always be added "on the fly" as all of the simplification functions take a `thm list` argument where these rules can be specified. If a set of rewrite rules is frequently used, then these should probably be made into a `ssfrag` value with the `rewrites` function and then added to an existing simpset with `++`.

The conversions which are part of simpsets are useful for situations where simple rewriting is not enough to transform certain terms. For example, the `BETA_CONV` conversion is not expressible as a standard first order rewrite, but is part of the `bool_ss` simpset and the application of this simpset will thus simplify all occurrences of `(\x. e1) e2`.

In fact, conversions in simpsets are not typically applied indiscriminately to all sub-terms. (If a conversion is applied to an inappropriate sub-term and fails, this failure is caught by the simplifier and ignored.) Instead, conversions in simpsets are accompanied by a term-pattern which specifies the sort of situations in which they should be applied. This facility is used in the definition of `bool_ss` to include `ETA_CONV`, but stop it from transforming `!x. P x` into `$! P`.

AC-rules allow simpsets to be constructed that automatically normalise terms involving associative and commutative operators, again according to some arbitrary term ordering metric.

Congruence rules allow SIMP_CONV to assume additional context as a term is rewritten. In a term such as `P ==> Q /\ f x` the truth of term P may be assumed as an additional piece of context in the rewriting of `Q /\ f x`. The congruence theorem that states this is valid is (IMP_CONG):

```
|- (P = P') ==> (P' ==> (Q = Q')) ==> ((P ==> Q) = (P' ==> Q'))
```

Other congruence theorems can be part of simpsets. The system provides IMP_CONG above and COND_CONG as part of the CONG_ss ssfrag value. (These simpset fragments can be incorporated into simpsets with the ++ function.) Other congruence theorems are already proved for operators such as conjunction and disjunction, but use of these in standard simpsets is not recommended as the computation of all the additional contexts for a simple chain of conjuncts or disjuncts can be very computationally intensive.

Decision procedures in simpsets are similar to conversions. They are arbitrary pieces of code that are applied to sub-terms at low priority. They are given access to the wider context through a list of relevant theorems. The arith_ss simpset includes an arithmetic decision procedure implemented in this way.

Failure

SIMP_CONV never fails, but may diverge.

Example

```
> SIMP_CONV arith_ss [] ``(\x. x + 3) 4``;
val it = ⊢ (λx. x + 3) 4 = 7: thm
```

SIMP_CONV is a powerful way of manipulating terms. Other functions in the simplification library provide the same facilities when in the contexts of goals and tactics (SIMP_TAC, ASM_SIMP_TAC etc.), and theorems (SIMP_RULE), but SIMP_CONV provides the underlying functionality, and is useful in its own right, just as conversions are generally.

See also

bossLib.++, bossLib.ASM_SIMP_TAC, bossLib.FULL_SIMP_TAC, simpLib.mk_simpset, bossLib.rewrites, bossLib.SIMP_RULE, bossLib.SIMP_TAC, simpLib.SSFRAG, bossLib.EVAL

SIMP_CONV (simpLib)

```
simpLib.SIMP_CONV : simpset -> thm list -> conv
```

Re-exported from `bossLib.SIMP_CONV`. See that entry for full documentation.

SIMP_PROVE (simpLib)

```
simpLib.SIMP_PROVE : simpset -> thm list -> term -> thm
```

Like `SIMP_CONV`, but converts boolean terms to theorem with same conclusion.

`SIMP_PROVE ss thml` is equivalent to `EQT_ELIM o SIMP_CONV ss thml`.

Failure

Fails if the term can not be shown to be equivalent to true. May diverge.

Example

Applying the tactic

```
ASSUME_TAC (SIMP_PROVE arith_ss [] ``x < y ==> x < y + 6``)
```

to the goal `?- x + y = 10` yields the new goal

```
x < y ==> x < y + 6 ?- x + y = 10
```

Using `SIMP_PROVE` here allows `ASSUME_TAC` to add a new fact, where the equality with truth that `SIMP_CONV` would produce would be less useful.

`SIMP_PROVE` is useful when constructing theorems to be passed to other tools, where those other tools would prefer not to have theorems of the form `|- P = T`.

See also

simpLib.SIMP_CONV, simpLib.SIMP_RULE, simpLib.SIMP_TAC

SIMP_RULE (bossLib)

```
bossLib.SIMP_RULE : simpset -> thm list -> thm -> thm
```

Also exported as ***simpLib.SIMP_RULE****.*

Simplifies the conclusion of a theorem according to the given simpset and theorem rewrites.

SIMP_RULE simplifies the conclusion of a theorem, adding the given theorems to the simpset parameter as rewrites. The way in which terms are transformed as a part of simplification is described in the entry for SIMP_CONV.

Failure

Never fails, but may diverge.

Example

The following also demonstrates the higher order rewriting possible with simplification (FORALL_AND_THM states `|- (!x. P x /\ Q x) = (!x. P x) /\ (!x. Q x)`):

```
> SIMP_RULE bool_ss [boolTheory.FORALL_AND_THM]
            (ASSUME (Term`!x. P (x + 1) /\ R x /\ x < y`));
val it =  [.] ⊢ (∀x. P (x + 1)) ∧ (∀x. R x) ∧ ∀x. x < y: thm
```

Comments

SIMP_RULE ss thmlist is equivalent to CONV_RULE (SIMP_CONV ss thmlist).

See also

simpLib.ASM_SIMP_RULE, bossLib.SIMP_CONV, bossLib.SIMP_TAC, bossLib.bool_ss

SIMP_RULE (simpLib)

```
simpLib.SIMP_RULE : simpset -> thm list -> thm -> thm
```

Re-exported from `bossLib.SIMP_RULE`. See that entry for full documentation.

SIMP_TAC (bossLib)

```
bossLib.SIMP_TAC : simpset -> thm list -> tactic
```

Also exported as `simpLib.SIMP_TAC`.

Simplifies the goal, using the given simpset and the additional theorems listed.

`SIMP_TAC` adds the theorems of the second argument to the simpset argument as rewrites and then applies the resulting simpset to the conclusion of the goal. The exact behaviour of a simpset when applied to a term is described further in the entry for `SIMP_CONV`.

With simple simpsets, `SIMP_TAC` is similar in effect to `REWRITE_TAC`; it transforms the conclusion of a goal by using the (equational) theorems given and those already in the simpset as rewrite rules over the structure of the conclusion of the goal.

Just as `ASM_REWRITE_TAC` includes the assumptions of a goal in the rewrite rules that `REWRITE_TAC` uses, `ASM_SIMP_TAC` adds the assumptions of a goal to the rewrites and then performs simplification.

Failure

`SIMP_TAC` never fails, though it may diverge.

Example

`SIMP_TAC` and the `arith_ss` simpset combine to prove quite difficult seeming goals:

```
- val (_, p) = SIMP_TAC arith_ss []
               ([], Term`P x /\ (x = y + 3) ==> P x /\ y < x`);
```

```
> val p = fn : thm list -> thm

- p [];
> val it = |- P x /\ (x = y + 3) ==> P x /\ y < x : thm
```

SIMP_TAC is similar to REWRITE_TAC if used with just the bool_ss simpset. Here it is used in conjunction with the arithmetic theorem GREATER_DEF, |- !m n. m > n = n < m, to advance a goal:

```
- SIMP_TAC bool_ss [GREATER_DEF]  ([], Term`T /\ 5 > 4 \/ F`);
> val it = ([([], `4 < 5`)], fn) : subgoals
```

Comments

The simplification library is described further in other documentation, but its full capabilities are still rather opaque.

Simplification is one of the most powerful tactics available to the HOL user. It can be used both to solve goals entirely or to make progress with them. However, poor simpsets or a poor choice of rewrites can still result in divergence, or poor performance.

See also

bossLib.++, bossLib.ASM_SIMP_TAC, bossLib.std_ss, bossLib.bool_ss, bossLib.arith_ss, bossLib.list_ss, bossLib.FULL_SIMP_TAC, simpLib.mk_simpset, Rewrite.REWRITE_TAC, bossLib.SIMP_CONV, simpLib.SIMP_PROVE, bossLib.SIMP_RULE

SIMP_TAC (simpLib)

```
simpLib.SIMP_TAC : simpset -> thm list -> tactic
```

Re-exported from bossLib.SIMP_TAC. See that entry for full documentation.

SIMPLE_EXISTS (Drule)

```
Drule.SIMPLE_EXISTS : term -> thm -> thm
```

Introduces existential quantification using as witness a given free variable.

When applied to a free variable term and a theorem, SIMPLE_EXISTS gives the theorem made by existentially quantifying the conclusion of the given theorem over the given free variable.

```
    A |- p
-------------  SIMPLE_EXISTS ``x``
    A |- ?x. p
```

Failure

Fails if the term argument is not a free variable.

Comments

The free variable need not appear in the conclusion of the theorem, and may appear in the hypotheses.

Example

```
- SIMPLE_EXISTS (Term `x`) (REFL (Term `x`));
> val it = |- ?x. x = x : thm

- SIMPLE_EXISTS (Term `x`) (REFL T);
> val it = |- ?x. T = T : thm
```

See also

Thm.EXISTS, Thm.CHOOSE, Tactic.EXISTS_TAC

SIMPLE_QUANT_INST_ss (quantHeuristicsLib)

`quantHeuristicsLib.SIMPLE_QUANT_INST_ss : simpLib.ssfrag`

A simpset fragment corresponding to `quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV`.

See also

`quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV`, `bossLib.SQI_ss`

SIMPLE_QUANT_INSTANTIATE_CONV(quantHeuristicsLib)

`quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV : conv`

A conversion for instantiating quantifiers. In contrast to `quantHeuristicsLib.QUANT_INSTANTIATE_CONV` it only searches for gap guesses without free variables. As a result, it is much less powerful, but also much faster than `quantHeuristicsLib.QUANT_INSTANTIATE_CONV`.

Failure

If no instantiation could be found.

Example

```
> quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV ``?x. P x /\ (x = 5)``
val it = ⊢ (∃x. P x ∧ x = 5) ⇔ P 5 ∧ 5 = 5: thm

> quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV ``!x. (x = 5) ==> P x``
val it = ⊢ (∀x. x = 5 ⇒ P x) ⇔ 5 = 5 ⇒ P 5: thm
```

```
> quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV ``!x. Q x ==> !z. Z z /\ (x = 5)
val it =
   ⊢ (∀x. Q x ⇒ ∀z. Z z ∧ x = 5 ⇒ P x z) ⇔ Q 5 ⇒ ∀z. Z z ∧ 5 = 5 ⇒ P 5 z: thm

> quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV ``!x. ((3, x, y) = zxy) ==> P x`
val it =
   ⊢ (∀x. (3,x,y) = zxy ⇒ P x) ⇔
     (3,FST (SND zxy),y) = zxy ⇒ P (FST (SND zxy)): thm

> quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV ``some x. (x = 2) /\ P x``
val it = ⊢ (some x. x = 2 ∧ P x) = if 2 = 2 ∧ P 2 then SOME 2 else NONE: thm

> quantHeuristicsLib.SIMPLE_QUANT_INSTANTIATE_CONV ``?x1 x2 x3. P x1 x2 /\ (x2::x1:
val it =
   ⊢ (∃x1 x2 x3. P x1 x2 ∧ x2::x1::l = 3::f x3::l') ⇔
     ∃x2 x3. P (f x3) x2 ∧ x2::f x3::l = 3::f x3::l': thm
```

See also

quantHeuristicsLib.QUANT_INSTANTIATE_CONV, unwind.UNWIND_EXISTS_CONV, unwind.UNWIND_-FORALL_CONV, quantHeuristicsLib.SIMPLE_QUANT_INST_ss, bossLib.SQI_ss

singleton_of_list (Lib)

```
Lib.singleton_of_list : 'a list -> 'a
```

Turns a single-element list into a singleton.

`singleton_of_list [x]` returns `x`.

Failure

Fails if applied to a list that is not of length 1.

See also

Lib.pair_of_list, Lib.triple_of_list, Lib.quadruple_of_list

SIZES_CONV (wordsLib)

```
wordsLib.SIZES_CONV : conv
```

Evaluates `dimindex`, `dimword` and `INT_MIN`.

Example

```
> wordsLib.SIZES_CONV “dimword(:32)”
val it = ⊢ dimword (:32) = 4294967296: thm
```

Comments

Evaluations are stored and so will be slightly faster when repeated.

See also

`wordsLib.SIZES_ss`

SIZES_ss (wordsLib)

```
wordsLib.SIZES_ss : ssfrag
```

Simplification fragment for words.

The fragment `SIZES_ss` evaluates terms “`dimindex(:'a)`”, “`dimword(:'a)`”, “`INT_MIN(:'a)`”, and “`FINITE (UNIV : 'a set)`” for numeric types.

Example

```
> SIMP_CONV (pure_ss++wordsLib.SIZES_ss) [] “dimindex(:32) + INT_MIN(:32) + dimword(:32)
```

```
val it =
   ⊢ dimindex (:32) + INT_MIN (:32) + dimword (:32) =
     32 + 2147483648 + 4294967296: thm
```

See also

wordsLib.SIZES_CONV, wordsLib.WORD_CONV, fcpLib.FCP_ss, wordsLib.BIT_ss, wordsLib.WORD_ARITH_ss, wordsLib.WORD_LOGIC_ss, wordsLib.WORD_SHIFT_ss, wordsLib.WORD_ARITH_EQ_ss, wordsLib.WORD_BIT_EQ_ss, wordsLib.WORD_EXTRACT_ss, wordsLib.WORD_MUL_LSL_ss, wordsLib.WORD_ss

SKOLEM_CONV (Conv)

```
Conv.SKOLEM_CONV : conv
```

Proves the existence of a Skolem function.

When applied to an argument of the form `!x1...xn. ?y. P`, the conversion `SKOLEM_CONV` returns the theorem:

```
   |- (!x1...xn. ?y. P) = (?y'. !x1...xn. P[y' x1 ... xn/y])
```

where `y'` is a primed variant of `y` not free in the input term.

Failure

`SKOLEM_CONV tm` fails if `tm` is not a term of the form `!x1...xn. ?y. P`.

See also

Conv.X_SKOLEM_CONV

snd (Lib)

```
Lib.snd : ('a * 'b) -> 'b
```

Extracts the second component of a pair.

`snd (x,y)` returns `y`.

Failure

Never fails. However, notice that `snd (x,y,z)` fails to typecheck, since `(x,y,z)` is not a pair.

Example

```
- snd (1, "foo");
> val it = "foo" : string

- snd (1, "foo", []);
! Toplevel input:
! snd (1, "foo", []);
!     ^^^^^^^^^^^^^
! Type clash: expression of type
!   'g * 'h * 'i
! cannot have type
!   'j * 'k
! because the tuple has the wrong number of components

- snd (1, ("foo", ()));
> val it = ("foo", ()) : string * unit
```

See also

`Lib`, `Lib.fst`

SNOC_CONV (listLib)

`listLib.SNOC_CONV : conv`

Computes by inference the result of adding an element to the tail end of a list.

SNOC_CONV takes a term tm in the following form:

```
SNOC x [x0;...xn]
```

It returns the theorem

```
|- SNOC x [x0;...xn] = [x0;...xn;x]
```

where the right-hand side is the list in the canonical form, i.e., constructed with only the constructor CONS.

Failure

SNOC_CONV tm fails if tm is not of the form described above.

Example

Evaluating

```
SNOC_CONV “SNOC 5[0;1;2;3;4]”;
```

returns the following theorem:

```
|- SNOC 5[0;1;2;3;4] = [0;1;2;3;4;5]
```

See also

listLib.FOLDL_CONV, listLib.FOLDR_CONV, listLib.list_FOLD_CONV

SNOC_INDUCT_TAC (listLib)

listLib.SNOC_INDUCT_TAC : tactic

Performs tactical proof by structural induction on lists.

SNOC_INDUCT_TAC reduces a goal `!l.P[l]`, where `l` ranges over lists, to two subgoals corresponding to the base and step cases in a proof by structural induction on `l` from the tail end. The induction hypothesis appears among the assumptions of the subgoal for the step case. The specification of SNOC_INDUCT_TAC is:

```
                    A ?- !l. P
   ================================================  SNOC_INDUCT_TAC
    A |- P[NIL/l]   A u {{P[l'/l]}} ?- !x. P[SNOC x l'/l]
```

where `l'` is a primed variant of `l` that does not appear free in the assumptions `A` (usually, `l'` is just `l`). When SNOC_INDUCT_TAC is applied to a goal of the form `!l.P`, where `l` does not appear free in `P`, the subgoals are just `A ?- P` and `A u {{P}} ?- !h.P`.

Failure

SNOC_INDUCT_TAC `g` fails unless the conclusion of the goal `g` has the form `!l.t`, where the variable `l` has type `(ty)list` for some type `ty`.

See also

listLib.EQ_LENGTH_INDUCT_TAC, listLib.EQ_LENGTH_SNOC_INDUCT_TAC, listLib.LIST_-INDUCT_TAC

SOME_EL_CONV (listLib)

```
listLib.SOME_EL_CONV : conv -> conv
```

Computes by inference the result of applying a predicate to the elements of a list.

SOME_EL_CONV takes a conversion `conv` and a term `tm` of the following form:

```
SOME_EL P [x0;...xn]
```

It returns the theorem

```
|- SOME_EL P [x0;...xn] = F
```

if for every xi occurred in the list, conv "P xi" returns a theorem |- P xi = F, otherwise, if for at least one xi, evaluating conv "P xi" returns the theorem |- P xi = T, then it returns the theorem

```
|- SOME_EL P [x0;...xn] = T
```

Failure

SOME_EL_CONV conv tm fails if tm is not of the form described above, or failure occurs when evaluating conv "P xi" for some xi.

Example

Evaluating

```
SOME_EL_CONV bool_EQ_CONV "SOME_EL ($= T) [T;F;T]";
```

returns the following theorem:

```
|- SOME_EL($= T)[T;F;T] = T
```

In general, if the predicate P is an explicit lambda abstraction (\x. P x), the conversion should be in the form

```
(BETA_CONV THENC conv')
```

See also

listLib.ALL_EL_CONV, listLib.IS_EL_CONV, listLib.FOLDL_CONV, listLib.FOLDR_CONV, listLib.list_FOLD_CONV

sort (Lib)

```
Lib.sort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Sorts a list using a given transitive 'ordering' relation.

The call `sort opr list` where `opr` is a curried transitive relation on the elements of `list`, will sort the list, i.e., will permute `list` such that if `x opr y` but not `y opr x` then `x` will occur to the left of `y` in the sorted list. In particular if `opr` is a total order, the result list will be sorted in the usual sense of the word.

Failure

Never fails.

Example

A simple example is:

```
- sort (curry (op<)) [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5, 8, 9, 7, 9];
> val it = [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 7, 8, 9, 9, 9] : int list
```

The following example is a little more complicated. Note that the 'ordering' is not antisymmetric.

```
- sort (curry (op< o (fst ## fst)))
       [(1,3), (7,11), (3,2), (3,4), (7,2), (5,1)];
> val it = [(1,3), (3,4), (3,2), (5,1), (7,2), (7,11)] : (int * int) list
```

Comments

The Standard ML Basis Library also provides implementations of sorting.

See also

`Lib.int_sort`, `Lib.topsort`

sort_vars (HolKernel)

```
HolKernel.sort_vars : string list -> term list -> term list
```

Sorts a list of variables according to first argument.

A call to `sort_vars [s1,s2,..sn] vs` will return a permutation of `vs` such that variables with the name `s1` will appears first, followed by those with the name `s2` etc.

Failure

Never fails.

Example

```
> sort_vars ["a", "b", "d"] [``b:bool``, ``c:num``, ``d:bool``, ``a:'a``];
val it = [“a”, “b”, “d”, “c”]: term list
```

See also

`Conv.RESORT_EXISTS_CONV`, `Conv.RESORT_FORALL_CONV`

SPEC (Thm)

```
Thm.SPEC : term -> thm -> thm
```

Specializes the conclusion of a theorem.

When applied to a term `u` and a theorem `A |- !x. t`, then `SPEC` returns the theorem `A |- t[u/x]`. If necessary, variables will be renamed prior to the specialization to ensure that `u` is free for `x` in `t`, that is, no variables free in `u` become bound after substitution.

```
    A |- !x. t
--------------  SPEC u
 A |- t[u/x]
```

Failure

Fails if the theorem’s conclusion is not universally quantified, or if `x` and `u` have different types.

Example

The following example shows how SPEC renames bound variables if necessary, prior to substitution: a straightforward substitution would result in the clearly invalid theorem |- ~y ==> (!y. y ==> ~y).

```
- let val xv = Term `x:bool`
      and yv = Term `y:bool`
  in
    (GEN xv o DISCH xv o GEN yv o DISCH yv) (ASSUME xv)
  end;
> val it = |- !x. x ==> !y. y ==> x : thm

- SPEC (Term `~y`) it;
> val it = |- ~y ==> !y'. y' ==> ~y : thm
```

See also

Drule.ISPEC, Drule.SPECL, Drule.SPEC_ALL, Drule.SPEC_VAR, Thm.GEN, Thm.GENL, Drule.GEN_ALL

SPEC_ALL (Drule)

```
Drule.SPEC_ALL : thm -> thm
```

Specializes the conclusion of a theorem with its own quantified variables.

When applied to a theorem A |- !x1...xn. t, the inference rule SPEC_ALL returns the theorem A |- t[x1'/x1]...[xn'/xn] where the xi' are distinct variants of the corresponding xi, chosen to avoid clashes with any variables free in the assumption list and with the names of constants. Normally xi' is just xi, in which case SPEC_ALL simply removes all universal quantifiers.

```
     A |- !x1...xn. t
--------------------------  SPEC_ALL
 A |- t[x1'/x1]...[xn'/xn]
```

Failure

Never fails.

Example

```
> SPEC_ALL CONJ_ASSOC;
val it = ⊢ t1 ∧ t2 ∧ t3 ⇔ (t1 ∧ t2) ∧ t3: thm
```

See also

Thm.GEN, Thm.GENL, Drule.GEN_ALL, Tactic.GEN_TAC, Thm.SPEC, Drule.SPECL, Tactic.SPEC_TAC

SPEC_ALL_TAC (ConseqConv)

```
ConseqConv.SPEC_ALL_TAC : tactic
```

Generalizes a goal.

When applied to a goal `A ?- t`, the tactic SPEC_ALL_TAC generalizes all variables that are free in `t`, but not in `A`. This results in a goal of the form `A ?- !x1 ... xn. t`.

```
          A ?- t
====================  SPEC_ALL_TAC
 A ?- !x1 ... xn. t
```

Example

```
- val _ = set_goal ([``(P x):bool``], ``Q x /\ Z y``)
> Initial goal:

  Q x /\ Z y
  ------------------------------------
    P x
```

```
- e(SPEC_ALL_TAC)
>
  !Q Z y. Q x /\ Z y
  ------------------------------------
    P x
```

Failure

`SPEC_ALL_TAC` never fails. However, maybe no variable is generalized.

See also

`Tactic.SPEC_TAC`

SPEC_TAC (Tactic)

```
Tactic.SPEC_TAC : term * term -> tactic
```

Generalizes a goal.

When applied to a pair of terms `(u,x)`, where `x` is just a variable, and a goal `A ?- t`, the tactic `SPEC_TAC` generalizes the goal to `A ?- !x. t[x/u]`, that is, all instances of `u` are turned into `x`.

```
      A ?- t
================  SPEC_TAC (u,x)
 A ?- !x. t[x/u]
```

Failure

Fails unless `x` is a variable with the same type as `u`.

Removing unnecessary speciality in a goal, particularly as a prelude to an inductive proof.

See also

Thm.GEN, Thm.GENL, Drule.GEN_ALL, Tactic.GEN_TAC, Thm.SPEC, Drule.SPECL, Drule.SPEC_ALL, Tactic.STRIP_TAC

SPEC_UNDISCH_EXL (Drule)

```
Drule.SPEC_UNDISCH_EXL : thm -> thm
```

Strips universal quantifiers and antecedents of implications (splitting conjunctive antecedents), then where possible replaces the formerly quantified variables as existentials in the new hypotheses.

In this example, assume that a1 and a3 (only) involve the free variable x.

```
|- !x. a1 ==> !y. a2 ==> !z. a3 ==> !u. t
---------------------------------------- SPEC_UNDISCH_EXL
    ?x. a1 /\ a3, a2 |- t
```

Failure

No failure. Can leave the supplied theorem unchanged.

Comments

See EXISTS_LEFT for more on the existential quantification aspect. Note that the existential quantification happens only for variables which were universally quantified in the supplied theorem (to get around this limitation, first apply GEN_ALL to the supplied theorem).

Example

Where trans_thm is `[] |- !x y z. (x = y) ==> (y = z) ==> (x = z)`

SPEC_UNDISCH_EXL trans_thm is `[?y. (x = y) /\ (y = z)] |- x = z`

See also

`Drule.EXISTS_LEFT`, `Tactic.irule`

SPEC_VAR (Drule)

```
Drule.SPEC_VAR : thm -> term * thm
```

Specializes the conclusion of a theorem, returning the chosen variant.

When applied to a theorem `A |- !x. t`, the inference rule `SPEC_VAR` returns the term `x'` and the theorem `A |- t[x'/x]`, where `x'` is a variant of `x` chosen to avoid free variable capture.

```
    A |- !x. t
--------------  SPEC_VAR
 A |- t[x'/x]
```

Failure

Fails unless the theorem's conclusion is universally quantified.

Comments

This rule is very similar to plain `SPEC`, except that it returns the variant chosen, which may be useful information under some circumstances.

See also

`Thm.GEN`, `Thm.GENL`, `Drule.GEN_ALL`, `Tactic.GEN_TAC`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`

Specialize (Thm)

```
Thm.Specialize : term -> thm -> thm
```

Specializes the conclusion of a universal theorem.

When applied to a term `u` and a theorem `A |- !x. t`, `Specialize` returns the theorem `A |- t[u/x]`. Care is taken to ensure that no variables free in `u` become bound after this operation.

```
    A |- !x. t
--------------  Specialize u
 A |- t[u/x]
```

Failure

Fails if the theorem's conclusion is not universally quantified, or if `x` and `u` have different types.

Comments

`Specialize` behaves identically to `SPEC`, but is faster.

See also

`Thm.SPEC`, `Drule.ISPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`, `Drule.SPEC_VAR`, `Thm.GEN`, `Thm.GENL`, `Drule.GEN_ALL`

SPECL (Drule)

```
Drule.SPECL : term list -> thm -> thm
```

Specializes zero or more variables in the conclusion of a theorem.

When applied to a term list `[u1;...;un]` and a theorem `A |- !x1...xn. t`, the inference rule `SPECL` returns the theorem `A |- t[u1/x1]...[un/xn]`, where the substitutions are made sequentially left-to-right in the same way as for `SPEC`, with the same sort of alpha-conversions applied to `t` if necessary to ensure that no variables which are free in `ui` become bound after substitution.

```
      A |- !x1...xn. t
--------------------------  SPECL [u1,...,un]
  A |- t[u1/x1]...[un/xn]
```

It is permissible for the term-list to be empty, in which case the application of SPECL has no effect.

Failure

Fails unless each of the terms is of the same type as that of the appropriate quantified variable in the original theorem.

Example

The following is a specialization of a theorem from theory `arithmetic`.

```
- arithmeticTheory.LESS_EQ_LESS_EQ_MONO;
> val it = |- !m n p q. m <= p /\ n <= q ==> m + n <= p + q : thm

- SPECL (map Term [`1`, `2`, `3`, `4`]) it;
> val it = |- 1 <= 3 /\ 2 <= 4 ==> 1 + 2 <= 3 + 4 : thm
```

See also

`Thm.GEN`, `Thm.GENL`, `Drule.GEN_ALL`, `Tactic.GEN_TAC`, `Thm.SPEC`, `Drule.SPEC_ALL`, `Tactic.SPEC_TAC`

spine_pair (pairSyntax)

```
pairSyntax.spine_pair : term -> term list
```

Breaks a paired structure into its constituent pieces.

Example

```
> pairSyntax.spine_pair (Term `((1,2),(3,4))`);
val it = [“(1,2)”, “3”, “4”]: term list
```

Comments

Note that `spine_pair` is similar, but not identical, to `strip_pair` which works recursively.

Failure

Never fails.

See also

`pairSyntax.strip_pair`

SPLICE_CONJ_CONV (Conv)

```
Conv.SPLICE_CONJ_CONV : conv
```

Partially normalize a conjunction.

Normalize to right associativity a conjunction without recursing in the right conjunct.

Failure

Fails if the user-provided term is not a conjunction.

Example

```
> SPLICE_CONJ_CONV ``(a1 /\ a2 /\ a3) /\ b /\ c``;
val it = ⊢ (a1 ∧ a2 ∧ a3) ∧ b ∧ c ⇔ a1 ∧ a2 ∧ a3 ∧ b ∧ c: thm

> SPLICE_CONJ_CONV ``(a1 /\ a2) /\ (b1 /\ b2) /\ c``;
val it = ⊢ (a1 ∧ a2) ∧ (b1 ∧ b2) ∧ c ⇔ a1 ∧ a2 ∧ (b1 ∧ b2) ∧ c: thm
```

split (Lib)

```
Lib.split : ('a * 'b) list -> ('a list * 'b list)
```

Converts a list of pairs into a pair of lists.

`split [(x1,y1),...,(xn,yn)]` returns `([x1,...,xn],[y1,...,yn])`.

Failure

Never fails.

Comments

Identical to the Basis function `ListPair.unzip` and the function `Lib.unzip`.

See also

`Lib.unzip`, `Lib.zip`, `Lib.combine`

split_after (Lib)

```
Lib.split_after : int -> 'a list -> 'a list * 'a list
```

Breaks a list in two at a specified index.

An invocation `split_after k [x1,...,xk,...xn]` returns the pair `([x1,...,xk], [xk+1,...,xn])`. If `k` is `0`, then `split_after k l` returns `([],l)`. Similarly, `split_after (length l) l` returns `(l,[])`.

Failure

If `k` is negative, or longer than the length of the list.

Example

```
> split_after 2 [1,2,3,4,5]
val it = ([1, 2], [3, 4, 5]): int list * int list

> split_after 0 [1,2,3,4,5];
val it = ([], [1, 2, 3, 4, 5]): int list * int list

> split_after 5 [1,2,3,4,5];
val it = ([1, 2, 3, 4, 5], []): int list * int list

> split_after 6 [1,2,3,4,5];
Exception- HOL_ERR (at Lib.split_after: index too big) raised

> split_after 0 ([]:int list);
val it = ([], []): int list * int list
```

See also

`Lib.partition`, `Lib.pluck`

SPLIT_LT (Tactical)

```
Tactical.SPLIT_LT : int -> list_tactic * list_tactic -> list_tactic
```

Splits a list of goals into two and applies a list-tactic to each part

For list-tactics `ltac1` and `ltac2`, integer `n` and goal list `gl`, the application `SPLIT_LT n (ltac1, ltac2) gl` applies `ltac1` to the first `n` goals in `gl`, and `ltac2` to the remainder. If `n` is negative, `ltac1` is applied to the goals before the last `-n`, and `ltac2` to the last `-n` goals.

Failure

The application `SPLIT_LT n (ltac1, ltac2)` never fails, but when applied to a goal list, it fails if the index `n` is (in absolute value) larger then the length of the list, or if either of the list-tactics `ltac1` and `ltac2` fails.

Example

To apply tactic `tac1` to a goal, and then to apply `tac2` to all resulting subgoals except the first, use

```
tac1 THEN_LT SPLIT_LT 1 (ALL_LT, ALLGOALS tac2)
```

See also

`Tactical.THEN_LT`, `Tactical.ALL_LT`, `Tactical.ALLGOALS`

split_pair_case_tac (bossLib)

`bossLib.split_pair_case_tac : tactic`

Splits pairs that are arguments to a pair-case expression.

A call to `split_pair_case_tac` will search the goal (starting with the conclusion and moving onto each assumption in turn), for a sub-term of the form `case p of (x,y,...) => e`, where the variables appearing in `p` are free in the goal. The tactic will then introduce a new assumption in the goal of the form

```
p = (x,y,...)
```

where the variables `x`, `y` etc., are chosen to be as close as possible to the names in the case expression. In other words, they will vary only if those names are already free in the goal.

Failure

Fails if there is no such sub-term.

Example

```
> split_pair_case_tac ([], ``(case p of (x,y,z) => x + y * z) > 10``);
val it = ([([“p = (x,y,z)”], “(case p of (x,y,z) => x + y * z) > 10”)], fn):
   goal list * validation
```

See also

bossLib.pairarg_tac, pairLib.Pair_Cases_on

SPOSE_NOT_THEN (bossLib)

```
bossLib.SPOSE_NOT_THEN : (thm -> tactic) -> tactic
```

Initiate proof by contradiction.

SPOSE_NOT_THEN provides a flexible way to start a proof by contradiction. Simple tactics for contradiction proofs often simply negate the goal and place it on the assumption list. However, if the goal is quantified, as is often the case, then more processing is required in order to get it into a suitable form for subsequent work. SPOSE_NOT_THEN ttac negates the current goal, pushes the negation inwards, and applies ttac to it.

Failure

Never fails, unless ttac fails.

Example

Suppose we want to prove Euclid's theorem.

```
!m. ?n. prime n /\ m < n
```

The classic proof is by contradiction. However, if we start such a proof with CCONTR_TAC, we get the goal

```
{ ~!m. ?n. prime n /\ m < n } ?- F
```

and one would immediately want to simplify the assumption, which is a bit awkward. Instead, an invocation SPOSE_NOT_THEN ASSUME_TAC yields

```
{ ?m. !n. ~prime n \/ ~(m < n) } ?- F
```

and SPOSE_NOT_THEN STRIP_ASSUME_TAC results in

```
{ !n. ~prime n \/ ~(m < n) } ?- F
```

See also

Tactic.CCONTR_TAC, Tactic.CONTR_TAC, Tactic.ASSUME_TAC, Tactic.STRIP_ASSUME_TAC

SQI_ss (bossLib)

```
bossLib.SQI_ss : simpLib.ssfrag
```

A synonym for quantHeuristicsLib.SIMPLE_QUANT_INST_ss.

See also

quantHeuristicsLib.SIMPLE_QUANT_INST_ss

SRULE (bossLib)

```
bossLib.SRULE : thm list -> thm -> thm
```

Simplification with standard simpset as a derived rule

A call to SRULE ths th simplifies the theorem th using the standard simpset (accessible through a call to srw_ss()) and the theorems ths, returning the simplified theorem.

The implementation of SRULE is

```
fun SRULE ths th = SIMP_RULE (srw_ss()) ths th
```

The fact that this definition is not eta-reduced means that partial applications of SRULE will continue to pick up the current value of srw_ss() when they are eventually fully applied, rather than bake in the value from the time of the partial application.

Failure

Should never fail.

See also

Conv.CONV_RULE, simpLib.SIMP_RULE, BasicProvers.srw_ss

srw_ss (BasicProvers)

```
BasicProvers.srw_ss : unit -> simpset
```

Re-exported from bossLib.srw_ss. See that entry for full documentation.

srw_ss (bossLib)

```
bossLib.srw_ss : unit -> simpset
```

Also exported as BasicProvers.srw_ss.

Returns the "stateful rewriting" system's underlying simpset.

A call to srw_ss() returns a simpset value that is internally maintained and updated by the system. Its value changes as new types enter the TypeBase, and as theories are loaded. For this reason, it can't be accessed as a simple value, but is instead hidden behind a function.

The value behind srw_ss() can change in three ways. Firstly, whenever a type enters the TypeBase, the type's associated simplification theorems (accessible directly using the function TypeBase.simpls_of) are all added to the simpset. This ensures that the "obvious" rewrite theorems for a type (such as the disjointness of constructors) need never be explicitly specified.

Secondly, users can interactively add simpset fragments to the srw_ss() value by using the function augment_srw_ss. This function might be used after a definition is made to ensure that a particular constant always has its definition expanded. (Whether or not a constant warrants this is something that needs to be determined on a case-by-case basis.)

Thirdly, theories can augment the srw_ss() value as they load. This is set up in a theory's script file with the function export_rewrites. This causes a list of appropriate theorems to be added when the theory loads. It is up to the author of the theory to ensure that the theorems added to the simpset are sensible.

Failure

Never fails.

See also

`bossLib.augment_srw_ss`, `BasicProvers.export_rewrites`, `bossLib.SRW_TAC`

SRW_TAC (BasicProvers)

```
BasicProvers.SRW_TAC : ssfrag list -> thm list -> tactic
```

Re-exported from `bossLib.SRW_TAC`. See that entry for full documentation.

SRW_TAC (bossLib)

```
bossLib.SRW_TAC : ssfrag list -> thm list -> tactic
```

Also exported as `BasicProvers.SRW_TAC`.

A version of `RW_TAC` with an implicit `simpset`.

A call to `SRW_TAC [d1,...,dn] thlist` produces the same result as

```
RW_TAC (srw_ss() ++ d1 ++ ... ++ dn) thlist
```

Failure

When applied to a goal, the tactic resulting from an application of `SRW_TAC` may diverge.

Comments

There are two reasons why one might prefer SRW_TAC to RW_TAC. Firstly, when a large number of datatypes are present in the TypeBase, the implementation of RW_TAC has to merge the attendant simplifications for each type onto its simpset argument each time it is called. This can be rather time-consuming. Secondly, the simpset returned by srw_ss() can be augmented with fragments from other sources than the TypeBase, using the functions augment_srw_ss and export_rewrites. This can make for a tool that is simple to use, and powerful because of all its accumulated simpset fragments.

Naturally, the latter advantage can also be a disadvantage: if SRW_TAC does too much because there is too much in the simpset underneath srw_ss(), then there is no way to get around this using SRW_TAC.

Typical invocations of SRW_TAC will be of the form

```
SRW_TAC [][th1, th2,.. ]
```

The first argument, for lists of simpset fragments is for the inclusion of fragments that are not always appropriate. An example of such a fragment is numSimps.ARITH_ss, which embodies an arithmetic decision procedure for the natural numbers.

See also

bossLib.srw_ss, bossLib.augment_srw_ss, BasicProvers.export_rewrites, simpLib.SSFRAG

SSFRAG (simpLib)

```
simpLib.SSFRAG : { ac : (thm * thm) list,
           congs  : thm list,
           convs  : {conv  : (term list -> conv) -> term list -> conv,
                     key   : (term list * term) option,
                     name  : string,
                     trace : int} list,
           dprocs : Traverse.reducer list,
           filter : (controlled_thm -> controlled_thm list) option,
           name   : string option,
           rewrs  : thm list } -> ssfrag
```

Constructs `ssfrag` values.

The `ssfrag` type is the way in which simplification components are packaged up and made available to the simplifier (though `ssfrag` values must first be turned into simpsets, either by addition to an existing simpset, or with the `mk_simpset` function).

The big record type passed to `SSFRAG` as an argument has seven fields. Here we describe each in turn.

The `ac` field is a list of "AC theorem" pairs. Each such pair is the pair of theorems stating that a given binary function is associative and commutative. The theorems can be given in either order, can present the associativity theorem in either direction, and can be generalised to any extent.

The `congs` field is a list of congruence theorems justifying the addition of theorems to simplification contexts. For example, the congruence theorem for implication is

```
|- (P = P') ==> (P' ==> (Q = Q')) ==> (P ==> Q = P' ==> Q')
```

This theorem encodes a rewriting strategy. The consequent of the chain of implications is the form of term in question, where the appropriate components have been rewritten. Then, in left-to-right order, the various antecedents of the implication specify the rewriting strategy which gives rise to the consequent. In this example, `P` is first simplified to `P'` without any additional context, then, using `P'` as additional context, simplification of `Q` proceeds, producing `Q'`. Another example is a rule for conjunction:

```
|- (P ==> (Q = Q')) ==> (Q' ==> (P = P')) ==> ((P /\ Q) = (P' /\ Q'))
```

Here `P` is assumed while `Q` is simplified to `Q'`. Then, `Q'` is assumed while `P` is simplified to `P'`. If an antecedent doesn't involve the relation in question (here, equality) then it is treated as a side-condition, and the simplifier will be recursively invoked to try and solve it.

Higher-order congruence rules are also possible. These provide a method for dealing with bound variables. The following is a rule for the restricted universal quantifier, for example:

```
|- (P = Q) ==> (!v. v IN Q ==> (f v = g v)) ==>
   (RES_FORALL P f = RES_FORALL Q g)
```

(If `f` is an abstraction, `\x. t`, then `RES_FORALL P f` is pretty-printed as `!x::P. t`) Terms in the conclusions of higher-order congruence rules that might be abstractions (such as `f` above) should be kept as variables, rather than written out as abstractions. In other words, the conclusion of the congruence rule above should not be written as

```
RES_FORALL P (\v. f v) = RES_FORALL Q (\v. g v)
```

The convs field is a list of conversions that the simplifier will apply. Each conversion added to an ssfrag value is done so in a record consisting of four fields.

The conv field of this subsidiary record type includes the value of the conversion itself. When the simplifier applies the conversion it is actually passed two extra arguments (as the type indicates). The first is a solver function that can be used to recursively do side-condition solving, and the second is a stack of side-conditions that have been accumulated to date. Many conversions will typically ignore these arguments (as in the example below).

The key field of the subsidiary record type is an optional pattern, specifying the places where the conversion should be applied. If the value is NONE, then the conversion will be applied to all sub-terms. If the value is SOME(lcs, t), then the term t is used as a pattern specifying those terms to which the conversion should be applied. Further, the list lcs (which must be a list of variables), specifies those variables in t which shouldn't be generalised against; they are effectively local constants. Note, however, that the types in the pattern term t will not be used to eliminate possible matches, so that if a match is desired with a particular type instantiation of a term, then the conversion will need to reject the input itself. The name and trace fields are only relevant to the debugging facilities of the simplifier.

The dprocs field of the record passed to SSFRAG is where decision procedures can be specified. Documentation describing the construction and use of values of type reducer is available in the DESCRIPTION.

The filter field of the record is an optional function, which, if present, is composed with the standard simplifier's function for generating rewrites from theorems, and replaces that function. The version of this present in bool_ss and its descendents will, for example, turn |- P /\ Q into |- P and |- Q, and |- ~(t1 = t2) into |- (t1 = t2) = F and |- (t2 = t1) = F.

The controlled_thm type is defined in the module BoundedRewrites, and pairs a theorem with a bound, which is either the value UNBOUNDED or the constructor BOUNDED applied to an integer reference. The reference is used to limit the number of times a rewrite may be applied. The filter gets information as to whether and how a rewrite is bounded as this can affect its decision on whether or not to include a rewrite at all (if it looks as if it will loop, and the bound is UNBOUNDED, it should be dropped, but it may choose to keep it if there is bound present).

The rewrs field of the record is a list of rewrite theorems that are to be applied.

The name field of the record is an optional name for the simpset fragment that is used when it, and simpsets that it becomes part of are pretty-printed.

Failure

Never fails. Failure to provide theorems of just the right form may cause later application of simplification functions to fail, documentation to the contrary notwithstanding.

Example

Given a conversion `MUL_CONV` to calculate multiplications, the following illustrates how this can be added to a simpset:

```
- val ssd = SSFRAG {ac = [], congs = [],
                    convs = [{conv = K (K MUL_CONV),
                              key= SOME ([], Term`x * y`),
                              name = "MUL_CONV",
                              trace = 2}],
                    dprocs = [], filter = NONE, rewrs = []};
> val ssd =
    SSFRAG{ac = [], congs = [],
           convs =
             [{conv = fn, key = SOME([], `x * y`), name = "MUL_CONV",
               trace = 2}], dprocs = [], filter = NONE, rewrs = []}
    : ssfrag
- SIMP_CONV bool_ss [] (Term`3 * 4`);
> val it = |- 3 * 4 = 3 * 4 : thm
- SIMP_CONV (bool_ss ++ ssd) [] (Term`3 * 4`);
> val it = |- 3 * 4 = 12 : thm
```

Given the theorems `ADD_SYM` and `ADD_ASSOC` from `arithmeticTheory`, we can construct a normaliser for additive terms.

```
- val ssd2 = SSFRAG {ac = [(SPEC_ALL ADD_ASSOC, SPEC_ALL ADD_SYM)],
                     congs = [], convs = [], dprocs = [],
                     filter = NONE, rewrs = []};
> val ssd2 =
    SSFRAG{ac = [(|- m + n + p = (m + n) + p, |- m + n = n + m)],
           congs = [], convs = [], dprocs = [], filter = NONE,
           rewrs = []}
    : ssfrag
- SIMP_CONV (bool_ss ++ ssd2) [] (Term`(y + 3) + x + 4`);
  (* note that the printing of + in this example is that of a
```

```
        right associative operator.*)
   > val it = |- (y + 3) + x + 4 = 3 + 4 + x + y : thm
```

See also

simpLib.++, boolSimps.bool_ss, simpLib.Cong, simpLib.mk_simpset, simpLib.rewrites, simpLib.SIMP_CONV

start_time (Lib)

```
Lib.start_time : unit -> Timer.cpu_timer
```

Set a timer running.

An application `start_time ()` creates a timer and starts it. A later invocation `end_time t`, where `t` is a timer, will need to be called to get the elapsed time between the two function calls.

Failure

Never fails.

Example

```
> val clock = start_time ();
val clock = ?: Timer.cpu_timer
```

Comments

Multiple timers may be started without any interfering with the others.

Further operations associated with the type `cpu_timer` may be found in the Standard ML Basis Library structures `Timer` and `Time`.

See also

Lib.end_time, Lib.time

state (Lib)

```
Lib.state : ('a,'b) istream -> 'b
```

Project the state of an istream.

An application `state istrm` yields the value of the current state of `istrm`.

Failure

If the projection function supplied when building the stream fails on the current element of the state.

Example

```
> val istrm = mk_istream (fn x => x+1) 0 (strcat "gsym" o int_to_string);
val istrm = ?: (int, string) istream

> state istrm;
val it = "gsym0": string

> next (next istrm);
val it = ?: (int, string) istream

> state istrm;
val it = "gsym2": string
```

See also

`Lib.mk_istream`, `Lib.next`, `Lib.reset`

std_ss (bossLib)

```
bossLib.std_ss : simpset
```

Basic simplification set.

The simplification set `std_ss` extends `bool_ss` with a useful set of rewrite rules for terms involving options, pairs, and sums. It also performs beta and eta reduction. It applies some standard rewrites to evaluate expressions involving only numerals.

The following rewrites from `pairTheory` are included in `std_ss`:

```
|- !x. (FST x,SND x) = x
|- !x y. FST (x,y) = x
|- !x y. SND (x,y) = y
|- !x y a b. ((x,y) = (a,b)) = (x = a) /\ (y = b)
|- !f. CURRY (UNCURRY f) = f
|- !f. UNCURRY (CURRY f) = f
|- (CURRY f = CURRY g) = (f = g)
|- (UNCURRY f = UNCURRY g) = (f = g)
|- !f x y. CURRY f x y = f (x,y)
|- !f x y. UNCURRY f (x,y) = f x y
|- !f g x y. (f ## g) (x,y) = (f x,g y)
```

The following rewrites from `sumTheory` are included in `std_ss`:

```
|- !x. ISL x ==> (INL (OUTL x) = x)
|- !x. ISR x ==> (INR (OUTR x) = x)
|- (!x. ISL (INL x)) /\ !y. ~ISL (INR y)
|- (!x. ISR (INR x)) /\ !y. ~ISR (INL y)
|- !x. OUTL (INL x) = x
|- !x. OUTR (INR x) = x
|- !x y. ~(INL x = INR y)
|- !x y. ~(INR y = INL x)
|- (!y x. (INL x = INL y) = (x = y)) /\
   (!y x. (INR x = INR y) = (x = y))
|- (!f g x. case f g (INL x) = f x) /\
   (!f g y. case f g (INR y) = g y)
```

The following rewrites from `optionTheory` are included in `std_ss`:

```
|- (!x y. (SOME x = SOME y) = (x = y))
|- (!x. ~(NONE = SOME x))
|- (!x. ~(SOME x = NONE))
|- (!x. THE (SOME x) = x)
```

```
|- (!x. IS_SOME (SOME x) = T)
|- (IS_SOME NONE = F)
|- (!x. IS_NONE x = (x = NONE))
|- (!x. ~IS_SOME x = (x = NONE))
|- (!x. IS_SOME x ==> (SOME (THE x) = x))
|- (!x. case NONE SOME x = x)
|- (!x. case x SOME x = x)
|- (!x. IS_NONE x ==> (case e f x = e))
|- (!x. IS_SOME x ==> (case e f x = f (THE x)))
|- (!x. IS_SOME x ==> (case e SOME x = x))
|- (!u f. case u f NONE = u)
|- (!u f x. case u f (SOME x) = f x)
|- (!f x. OPTION_MAP f (SOME x) = SOME (f x))
|- (!f. OPTION_MAP f NONE = NONE)
|- (OPTION_JOIN NONE = NONE)
|- (!x. OPTION_JOIN (SOME x) = x)
|- !f x y. (OPTION_MAP f x = SOME y) = ?z. (x = SOME z) /\ (y = f z)
|- !f x. (OPTION_MAP f x = NONE) = (x = NONE)
```

For performing obvious simplification steps on terms, formulas, and goals. Also, sometimes simplification with more powerful simpsets, like `arith_ss`, becomes too slow, in which case one can use `std_ss` supplemented with whatever theorems are needed.

Comments

The simplification sets provided in `BasicProvers` and `bossLib` (currently `bool_ss`, `std_ss`, `arith_ss`, and `list_ss`) do not include useful rewrites stemming from HOL datatype declarations, such as injectivity and distinctness of constructors. However, the simplification routines `RW_TAC` and `SRW_TAC` automatically load these rewrites.

See also

`BasicProvers.RW_TAC`, `BasicProvers.SRW_TAC`, `simpLib.SIMP_TAC`, `simpLib.SIMP_CONV`, `simpLib.SIMP_RULE`, `BasicProvers.bool_ss`, `bossLib.arith_ss`, `bossLib.list_ss`

store_thm (Tactical)

```
Tactical.store_thm : string * term * tactic -> thm
```

Proves and then stores a theorem in the current theory segment.

The call `store_thm(name, t, tac)` is equivalent to `save_thm(name, prove(t, tac))`.

Failure

Whenever `prove` fails to prove the given term.

Saving theorems for retrieval in later sessions. Binding the result of `store_thm` to an ML variable makes it easy to access the theorem in the current terminal session.

See also

`Tactical.prove`, `Theory.save_thm`

strcat (Lib)

```
Lib.strcat : string -> string -> string
```

Concatenates two ML strings.

Failure

Never fails.

Example

```
> strcat "1" "";
val it = "1": string
```

```
> strcat "hello" "world";
val it = "helloworld": string

> strcat "hello" (strcat " " "world");
val it = "hello world": string
```

STRENGTHEN_CONSEQ_CONV_RULE (ConseqConv)

```
ConseqConv.STRENGTHEN_CONSEQ_CONV_RULE : directed_conseq_conv -> thm -> thm
```

Tries to strengthen the antecedent of a theorem consisting of an implication.

Given a theorem of the form `|- A ==> C` and a directed consequence conversion `c` a call of `STRENGTHEN_CONSEQ_CONV_RULE c thm` tries to strengthen `A` to a predicate `sA` using `c`. If it succeeds it returns the theorem `|- sA ==> C`.

See also

`ConseqConv.WEAKEN_CONSEQ_CONV_RULE`

string_to_int (Lib)

```
Lib.string_to_int : string -> int
```

Translates from a string to an integer.

An application `string_to_int s` returns the integer denoted by `s`, if such exists.

Failure

If the string cannot be translated to an integer.

Example

```
> string_to_int "123";
val it = 123: int

> string_to_int "~123";
val it = ~123: int

> string_to_int "foo";
Exception- HOL_ERR (at Lib.string_to_int: not convertable) raised
```

Comments

Similar functionality can be obtained from the Standard ML Basis Library function `Int.fromString`.

See also

`Lib.int_to_string`

strip_abs (Term)

```
Term.strip_abs : term -> term list * term
```

Also exported as ***boolSyntax.strip_abs****.*

Break apart consecutive lambda abstractions.

If M is a term of the form `\v1...vn.N`, where `N` is not a lambda abstraction, then `strip_abs M` equals `([v1,...,vn],N)`. Otherwise, the result is `([],M)`.

Failure

Never fails.

Example

```
> strip_abs (Term `\x y z. x ==> y ==> z`);
val it = (["x", "y", "z"], "x ⇒ y ⇒ z"): term list * term

> strip_abs T;
val it = ([], "T"): term list * term
```

Comments

In the current implementation of HOL, `strip_abs` is far faster than iterating `dest_abs` for terms with many consecutive binders.

See also

`Term.strip_binder`, `Term.dest_abs`, `boolSyntax.strip_forall`, `boolSyntax.strip_exists`

strip_abs (boolSyntax)

```
boolSyntax.strip_abs : term -> term list * term
```

Re-exported from `Term.strip_abs`. See that entry for full documentation.

STRIP_ALL_THEN (Thm_cont)

```
Thm_cont.STRIP_ALL_THEN : thm_tactical
```

Splits a theorem into a list of theorems and then calls the resulting theorem tactic on it.

Given a theorem `th` `STRIP_ALL_THEN ttac th` splits `th` into a list of theorems and then applies the `ttac` on the resulting theorems. This is done by recursively applying `STRIP_THM_THEN` and then calling the `ttac` if the theorem can't be split anymore.

Failure

`STRIP_ALL_THEN ttac th` fails if the any application of `ttac` fails, which is applied with the stripped theorems from `th`.

Comments

`STRIP_ALL_THEN` behaves exactly like `REPEAT_TCL STRIP_THM_THEN` but is faster.

See also

`Thm_cont.REPEAT_TCL`, `Thm_cont.STRIP_ALL_THEN`, `Tactic.STRIP_ASSUME_TAC`

strip_anylet (pairSyntax)

```
pairSyntax.strip_anylet : term -> (term * term) list list * term
```

Repeatedly destructs arbitrary `let` terms.

The invocation `strip_anylet M` where `M` has the form of a let-abstraction, i.e., `LET P Q`, returns a pair `([[(a1,b1),...,(an,bn)], ... [(u1,v1),...,(uk,vk)]],body)`, where the first element of the pair is a list of lists of bindings, and the second is the body of the let. The binding lists are required since let terms can, in general, be of the form (using surface syntax) `let a1 = b1 and ... and an = bn in body`.

Failure

Never fails.

Example

```
> pairSyntax.strip_anylet ``let g x = A in
                let v = g x y in
                let f x y (a,b) = g a
                and foo = M
```

```
                in
                 f x foo v``;
val it =
   ([[(“g x”, “A”)], [(“v”, “g x y”)],
     [(“f x y (a,b)”, “g a”), (“foo”, “M”)]], “f x foo v”):
   (term * term) list list * term
```

Programming that involves manipulation of term syntax.

See also

`boolSyntax.dest_let`, `pairSyntax.mk_anylet`, `pairSyntax.list_mk_anylet`, `pairSyntax.dest_anylet`

STRIP_ASSUME_TAC (Tactic)

```
Tactic.STRIP_ASSUME_TAC : thm_tactic
```

Splits a theorem into a list of theorems and then adds them to the assumptions.

Given a theorem `th` and a goal `(A,t)`, `STRIP_ASSUME_TAC th` splits `th` into a list of theorems. This is done by recursively breaking conjunctions into separate conjuncts, cases-splitting disjunctions, and eliminating existential quantifiers by choosing arbitrary variables. Schematically, the following rules are applied:

```
         A ?- t
   ======================  STRIP_ASSUME_TAC (A' |- v1 /\ ... /\ vn)
    A u {v1,...,vn} ?- t

              A ?- t
   ================================  STRIP_ASSUME_TAC (A' |- v1 \/ ... \/ vn)
    A u {v1} ?- t ... A u {vn} ?- t

        A ?- t
   ====================  STRIP_ASSUME_TAC (A' |- ?x.v)
    A u {v[x'/x]} ?- t
```

where `x'` is a variant of `x`.

If the conclusion of `th` is not a conjunction, a disjunction or an existentially quantified term, the whole theorem `th` is added to the assumptions.

As assumptions are generated, they are examined to see if they solve the goal (either by being alpha-equivalent to the conclusion of the goal or by deriving a contradiction).

The assumptions of the theorem being split are not added to the assumptions of the goal(s), but they are recorded in the proof. This means that if `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), `STRIP_ASSUME_TAC (A'|-v)` results in an invalid tactic.

Failure

Never fails.

Example

When solving the goal

```
?- m = 0 + m
```

assuming the clauses for addition with `STRIP_ASSUME_TAC ADD_CLAUSES` results in the goal

```
{m + (SUC n) = SUC(m + n), (SUC m) + n = SUC(m + n),
 m + 0 = m, 0 + m = m, m = 0 + m} ?- m = 0 + m
```

while the same tactic directly solves the goal

```
?- 0 + m = m
```

`STRIP_ASSUME_TAC` is used when applying a previously proved theorem to solve a goal, or when enriching its assumptions so that resolution, rewriting with assumptions and other operations involving assumptions have more to work with.

See also

`Tactic.ASSUME_TAC`, `Tactic.CHOOSE_TAC`, `Thm_cont.CHOOSE_THEN`, `Thm_cont.CONJUNCTS_THEN`, `Tactic.DISJ_CASES_TAC`, `Thm_cont.DISJ_CASES_THEN`

strip_binder (Term)

```
Term.strip_binder : term option -> term -> term list * term
```

Break apart consecutive binders.

An application `strip_binder (SOME c) (c(\v1. ... (c(\vn.M))...))` returns `([v1,...,vn],M)`. The constant `c` should represent a term binding operation.

An application `strip_binder NONE (\v1...vn. M)` returns `([v1,...,vn],M)`.

Failure

Never fails.

Example

`strip_abs` could be defined as follows.

```
- val strip_abs = strip_binder NONE;
> val strip_abs = fn : term -> term list * term

- strip_abs (Term `\x y z. x /\ y ==> z`);
> val it = ([`x`, `y`, `z`], `x /\ y ==> z`) : term list * term
```

Defining `strip_forall` is similar.

```
strip_binder (SOME boolSyntax.universal)
```

Comments

Terms with many consecutive binders should be taken apart using `strip_binder` and its instantiations `strip_abs`, `strip_forall`, and `strip_exists`. In the current implementation of HOL, iterating `dest_abs`, `dest_forall`, or `dest_exists` is far slower for terms with many consecutive binders.

See also

Term.list_mk_binder, Term.strip_abs, boolSyntax.strip_forall, boolSyntax.strip_exists

STRIP_BINDER_CONV (Conv)

```
Conv.STRIP_BINDER_CONV : term option -> conv -> conv
```

Applies a conversion underneath a binder prefix.

If the application of conv to M yields |- M = N, then STRIP_BINDER_CONV (SOME c) conv (c(\v1. ... (c(\vn.M))...)) returns |- c(\v1. ... (c(\vn.M))...) = c(\v1. ... (c(\vn.N))...) and STRIP_BINDER_CONV NONE conv (\v1 ... vn.M) returns |- (\v1 ... vn.M) = (\v1 ... vn.N).

Failure

If conv M fails. Also fails if some of [v1,...,vn] are free in the hypotheses of conv M.

Example

```
> STRIP_BINDER_CONV NONE BETA_CONV (Term `\u v w. (\a. a + v * w) u`);
val it = ⊢ (λu v w. (λa. a + v * w) u) = (λu v w. u + v * w): thm

> STRIP_BINDER_CONV (SOME existential) SYM_CONV
                    (Term `?u v w x y. u + v = w + x + y`);
val it = ⊢ (∃u v w x y. u + v = w + x + y) ⇔ ∃u v w x y. w + x + y = u + v:
   thm
```

Comments

STRIP_BINDER_CONV is more efficient than iterated application of BINDER_CONV or ABS_CONV or QUANT_CONV.

See also

Conv.BINDER_CONV, Conv.ABS_CONV, Conv.QUANT_CONV, Conv.STRIP_BINDER_CONV, Conv.STRIP_-QUANT_CONV

strip_comb (boolSyntax)

```
boolSyntax.strip_comb : term -> term * term list
```

Iteratively breaks apart combinations (function applications).

If `M` has the form `t t1 ... tn` then `strip_comb M` returns `(t,[t1,...,tn])`. Note that

```
strip_comb(list_mk_comb(t,[t1,...,tn]))
```

will not be `(t,[t1,...,tn])` if `t` is a combination.

Failure

Never fails.

Example

```
> strip_comb (Term `x /\ y`);
val it = ("$/\", ["x", "y"]): term * term list

> strip_comb T;
val it = ("T", []): term * term list
```

See also

Term.list_mk_comb, Term.dest_comb

strip_conj (boolSyntax)

```
boolSyntax.strip_conj : term -> term list
```

Recursively breaks apart conjunctions.

If `M` is of the form `t1 /\ ... /\ tn`, where no `ti` is a conjunction, then `strip_conj M` returns `[t1,...,tn]`. Any `ti` that is a conjunction is broken down by `strip_conj`, hence

```
strip_conj(list_mk_conj [t1,...,tn])
```

will not return `[t1,...,tn]` if any `ti` is a conjunction.

Failure

Never fails.

Example

```
> strip_conj (Term `(a /\ b) /\ c /\ d`);
val it = [“a”, “b”, “c”, “d”]: term list
```

See also

`boolSyntax.dest_conj`, `boolSyntax.mk_conj`, `boolSyntax.list_mk_conj`

strip_disj (boolSyntax)

```
boolSyntax.strip_disj : term -> term list
```

Recursively breaks apart disjunctions.

If `M` is of the form `t1 \/ ... \/ tn`, where no `ti` is a disjunction, then `strip_disj M` returns `[t1,...,tn]`. Any `ti` that is a disjunction is broken down by `strip_disj`, hence

```
strip_disj(list_mk_disj [t1,...,tn])
```

will not return `[t1,...,tn]` if any `ti` is a disjunction.

Failure

Never fails.

Example

```
> strip_disj (Term `(a \/ b) \/ c \/ d`);
val it = [“a”, “b”, “c”, “d”]: term list
```

See also

`boolSyntax.dest_disj`, `boolSyntax.mk_disj`, `boolSyntax.list_mk_disj`

strip_exists (boolSyntax)

```
boolSyntax.strip_exists : term -> term list * term
```

Iteratively breaks apart existential quantifications.

If `M` has the structure `?x1 ... xn. t` then `strip_exists M` returns `([x1,...,xn],t)`. Note that

```
strip_exists(list_mk_exists(["x1";...;"xn"],"t"))
```

will not return `([x1,...,xn],t)` if `t` is an existential quantification.

Failure

Never fails.

See also

`boolSyntax.list_mk_exists`, `boolSyntax.dest_exists`

strip_forall (boolSyntax)

```
boolSyntax.strip_forall : term -> term list * term
```

Iteratively breaks apart universal quantifications.

If `M` has the form `!x1 ... xn. t` then `strip_forall M` returns `([x1,...,xn],t)`. Note that

```
strip_forall(list_mk_forall([x1,...,xn],t,))
```

will not return `([x1,...,xn],t)` if `t` is a universal quantification.

Failure

Never fails.

See also

`boolSyntax.list_mk_forall`, `boolSyntax.dest_forall`

strip_fun (boolSyntax)

```
boolSyntax.strip_fun : hol_type -> hol_type list * hol_type
```

Iteratively breaks apart function types.

If `fty` is of the form `ty1 -> (... (tyn -> ty) ...)`, then `strip_fun fty` returns `([ty1,...,tyn],ty)`. Note that

```
strip_fun(list_mk_fun([ty1,...,tyn],ty))
```

will not return `([ty1,...,tyn],ty)` if `ty` is a function type.

Failure

Never fails.

Example

```
> strip_fun (Type `:(a -> 'bool) -> ('b -> 'c)`);
Exception- HOL_ERR
  (at Parse.type parser: at line 1, character 19:
      a not a known type operator) raised
```

See also

`boolSyntax.list_mk_fun`, `Type.dom_rng`, `Type.dest_type`

STRIP_GOAL_THEN (Tactic)

```
Tactic.STRIP_GOAL_THEN : thm_tactic -> tactic
```

Splits a goal by eliminating one outermost connective, applying the given theorem-tactic to the antecedents of implications.

Given a theorem-tactic `ttac` and a goal `(A,t)`, `STRIP_GOAL_THEN` removes one outermost occurrence of one of the connectives `!`, `==>`, `~` or `/\` from the conclusion of the goal `t`. If `t` is a universally quantified term, then `STRIP_GOAL_THEN` strips off the quantifier:

```
    A ?- !x.u
==============  STRIP_GOAL_THEN ttac
  A ?- u[x'/x]
```

where `x'` is a primed variant that does not appear free in the assumptions `A`. If `t` is a conjunction, then `STRIP_GOAL_THEN` simply splits the conjunction into two subgoals:

```
    A ?- v /\ w
================  STRIP_GOAL_THEN ttac
 A ?- v   A ?- w
```

If `t` is an implication `u ==> v` and if:

```
     A ?- v
===============  ttac (u |- u)
   A' ?- v'
```

then:

```
     A ?- u ==> v
====================  STRIP_GOAL_THEN ttac
      A' ?- v'
```

Finally, a negation `~t` is treated as the implication `t ==> F`.

Failure

`STRIP_GOAL_THEN ttac (A,t)` fails if `t` is not a universally quantified term, an implication, a negation or a conjunction. Failure also occurs if the application of `ttac` fails, after stripping the goal.

Example

When solving the goal

```
?- (n = 1) ==> (n * n = n)
```

a possible initial step is to apply

```
STRIP_GOAL_THEN SUBST1_TAC
```

thus obtaining the goal

```
?- 1 * 1 = 1
```

`STRIP_GOAL_THEN` is used when manipulating intermediate results (obtained by stripping outer connectives from a goal) directly, rather than as assumptions.

See also

`Tactic.CONJ_TAC`, `Thm_cont.DISCH_THEN`, `Tactic.FILTER_STRIP_THEN`, `Tactic.GEN_-TAC`, `Tactic.STRIP_ASSUME_TAC`, `Tactic.STRIP_TAC`

strip_imp (boolSyntax)

```
boolSyntax.strip_imp : term -> term list * term
```

Iteratively breaks apart implications.

If `M` is of the form `t1 ==> (... (tn ==> t) ...)`, then `strip_imp M` returns `([t1,...,tn],t)`. Note that

```
strip_imp(list_mk_imp([t1,...,tn],t))
```

will not return `([t1,...,tn],t)` if `t` is an implication.

Failure

Never fails.

Example

```
> strip_imp "(T ==> F) ==> (T ==> F)";;
Exception- Type error in function application.
   Function: strip_imp : term -> term list * term
   Argument: "(T ==> F) ==> (T ==> F)" : string
   Reason:
      Can't unify term (*Created from opaque signature*) with
         string (*In Basis*) (Different type constructors)
Fail "Static Errors" raised

> strip_imp (Term `t1 ==> t2 ==> t3 ==> ~t`);
val it = ([“t1”, “t2”, “t3”, “t”], “F”): term list * term
```

See also

boolSyntax.list_mk_imp, boolSyntax.dest_imp

strip_imp_only (boolSyntax)

```
boolSyntax.strip_imp_only : term -> term list * term
```

Iteratively breaks apart implications.

If `M` is of the form `t1 ==> (... (tn ==> t) ...)`, then `strip_imp_only M` returns `([t1,...,tn],t)`. Note that

```
strip_imp_only(list_mk_imp([t1,...,tn],t))
```

will not return `([t1,...,tn],t)` if `t` is an implication.

Failure

Never fails.

Example

```
> strip_imp_only (Term `(T ==> F) ==> (T ==> F)`);
val it = ([“T ⇒ F”, “T”], “F”): term list * term

> strip_imp_only (Term `t1 ==> t2 ==> t3 ==> ~t`);
val it = ([“t1”, “t2”, “t3”], “¬t”): term list * term
```

See also

boolSyntax.list_mk_imp, boolSyntax.dest_imp

strip_neg (boolSyntax)

```
boolSyntax.strip_neg : term -> term * int
```

Breaks iterated negations down to an unnegated core.

If `M` is of the form `~...~t`, then `strip_neg M` returns `(t,n)`, where `n` is the number of consecutive negations being applied to `t`.

Failure

Never fails.

Example

```
> strip_neg (Term `~~~~t`);
val it = (“t”, 4): term * int

> strip_neg (Term `x`);
val it = (“x”, 0): term * int
```

Comments

There is no corresponding entrypoint for building iterated negations. If such functionality is desired, `funpow` may be used:

```
- funpow 3 mk_neg T;
> val it = `~~~T` : term
```

See also

`boolSyntax.dest_neg`, `boolSyntax.mk_neg`, `Lib.funpow`

strip_pabs (pairSyntax)

```
pairSyntax.strip_pabs : term -> term list * term
```

Iteratively breaks apart paired abstractions.

`strip_pabs "\p1 ... pn. t"` returns `([p1,...,pn],t)`. Note that

```
strip_pabs(list_mk_abs([p1,...,pn],t))
```

will not return `([p1,...,pn],t)` if `t` is a paired abstraction.

Failure

Never fails.

See also

`boolSyntax.strip_abs`, `pairSyntax.list_mk_pabs`, `pairSyntax.dest_pabs`

strip_pair (pairSyntax)

```
pairSyntax.strip_pair : term -> term list
```

Recursively breaks a paired structure into its constituent pieces.

Example

```
> pairSyntax.strip_pair (Term `((1,2),(3,4))`);
val it = [“1”, “2”, “3”, “4”]: term list
```

Comments

Note that `strip_pair` is similar, but not identical, to `spine_pair` which does not work recursively.

Failure

Never fails.

See also

`pairSyntax.spine_pair`

strip_pexists (pairSyntax)

```
pairSyntax.strip_pexists : term -> term list * term
```

Iteratively breaks apart paired existential quantifications.
`strip_pexists "?p1 ... pn. t"` returns `([p1,...,pn],t)`. Note that

```
strip_pexists(list_mk_pexists([[p1,...,pn],t))
```

will not return `([p1,...,pn],t)` if `t` is a paired existential quantification.

Failure

Never fails.

See also

`boolSyntax.strip_exists`, `pairSyntax.dest_pexists`

strip_pforall (pairSyntax)

```
pairSyntax.strip_pforall : term -> term list * term
```

Iteratively breaks apart paired universal quantifications.
`strip_pforall "!p1 ... pn. t"` returns `([p1,...,pn],t)`. Note that

```
strip_pforall(list_mk_pforall([p1,...,pn],t))
```

will not return `([p1,...,pn],t)` if `t` is a paired universal quantification.

Failure

Never fails.

See also

boolSyntax.strip_forall, pairSyntax.dest_pforall

STRIP_QUANT_CONV (Conv)

```
Conv.STRIP_QUANT_CONV : conv -> conv
```

Applies a conversion underneath a quantifier prefix.

If tm has the form Q(\v1. ... (Q(\vn.M))...) and the application of conv to M yields |- M = N, then STRIP_QUANT_CONV conv tm returns |- Q(\v1. ... (Q(\vn.M))...) = Q(\v1. ... (Q(\vn.N))...), provided Q is Hilbert's choice operator or a universal, existential, or unique-existence quantifer.

Otherwise, STRIP_QUANT_CONV conv tm returns conv tm.

Failure

If conv M fails. Or if conv tm fails when tm is not a quantified term. Also fails if some of [v1,...,vn] are free in the hypotheses of conv M.

Example

```
> STRIP_QUANT_CONV (STRIP_QUANT_CONV SYM_CONV)
   (Term `!x y z. ?!p q r. x + y*z = p*q + r`);
val it =
   ⊢ (∀x y z. ∃!p q r. x + y * z = p * q + r) ⇔
     ∀x y z. ∃!p q r. p * q + r = x + y * z: thm
```

Comments

To deal with binders not in the above list, e.g., newly introduced ones, use STRIP_BINDER_CONV.

For deeply nested quantifiers, STRIP_QUANT_CONV and STRIP_BINDER_CONV are more efficient than iterated application of QUANT_CONV, BINDER_CONV, or ABS_CONV.

See also

`Conv.STRIP_BINDER_CONV`, `Conv.QUANT_CONV`, `Conv.BINDER_CONV`, `Conv.ABS_CONV`

strip_res_exists (res_quanLib)

```
res_quanLib.strip_res_exists : (term -> ((term # term) list # term))
```

Iteratively breaks apart a restricted existentially quantified term.

`strip_res_exists` is an iterative term destructor for restricted existential quantifications. It iteratively breaks apart a restricted existentially quantified term into a list of pairs which are the restricted quantified variables and predicates and the body.

```
   strip_res_exists "?x1::P1. ... ?xn::Pn. t"
```

returns `([("x1","P1");...;("xn","Pn")],"t")`.

Failure

Never fails.

See also

`res_quanLib.list_mk_res_exists`, `res_quanLib.is_res_exists`, `res_quanLib.dest_res_exists`

strip_res_forall (res_quanLib)

```
res_quanLib.strip_res_forall : term -> ((term # term) list # term)
```

Iteratively breaks apart a restricted universally quantified term.

`strip_res_forall` is an iterative term destructor for restricted universal quantifications. It iteratively breaks apart a restricted universally quantified term into a list of pairs which are the restricted quantified variables and predicates and the body.

```
strip_res_forall "!x1::P1. ... !xn::Pn. t"
```

returns `([("x1","P1");...;("xn","Pn")],"t")`.

Failure

Never fails.

See also

`res_quanLib.list_mk_res_forall`, `res_quanLib.is_res_forall`, `res_quanLib.dest_res_forall`

STRIP_TAC (Tactic)

```
Tactic.STRIP_TAC : tactic
```

Splits a goal by eliminating one outermost connective.

Given a goal `(A,t)`, `STRIP_TAC` removes one outermost occurrence of one of the connectives `!`, `==>`, `~` or `/\` from the conclusion of the goal `t`. If `t` is a universally quantified term, then `STRIP_TAC` strips off the quantifier:

```
    A ?- !x.u
==============  STRIP_TAC
  A ?- u[x'/x]
```

where `x'` is a primed variant that does not appear free in the assumptions `A`. If `t` is a conjunction, then `STRIP_TAC` simply splits the conjunction into two subgoals:

```
    A ?- v /\ w
================  STRIP_TAC
 A ?- v   A ?- w
```

If `t` is an implication, `STRIP_TAC` moves the antecedent into the assumptions, stripping conjunctions, disjunctions and existential quantifiers according to the following rules:

```
 A ?- v1 /\ ... /\ vn ==> v
============================
    A u {v1,...,vn} ?- v
```

```
   A ?- v1 \/ ... \/ vn ==> v
================================
 A u {v1} ?- v ... A u {vn} ?- v
```

```
  A ?- ?x.w ==> v
====================
 A u {w[x'/x]} ?- v
```

where `x'` is a primed variant of `x` that does not appear free in `A`. Finally, a negation `~t` is treated as the implication `t ==> F`.

Failure

`STRIP_TAC (A,t)` fails if `t` is not a universally quantified term, an implication, a negation or a conjunction.

Example

Applying `STRIP_TAC` twice to the goal:

```
?- !n. m <= n /\ n <= m ==> (m = n)
```

results in the subgoal:

```
{n <= m, m <= n} ?- m = n
```

When trying to solve a goal, often the best thing to do first is `REPEAT STRIP_TAC` to split the goal up into manageable pieces.

See also

`Tactic.CONJ_TAC`, `Tactic.DISCH_TAC`, `Thm_cont.DISCH_THEN`, `Tactic.GEN_TAC`, `Tactic.STRIP_ASSUME_TAC`, `Tactic.STRIP_GOAL_THEN`

STRIP_THM_THEN (Thm_cont)

```
Thm_cont.STRIP_THM_THEN : thm_tactical
```

STRIP_THM_THEN applies the given theorem-tactic using the result of stripping off one outer connective from the given theorem.

Given a theorem-tactic ttac, a theorem th whose conclusion is a conjunction, a disjunction or an existentially quantified term, and a goal (A,t), STRIP_THM_THEN ttac th first strips apart the conclusion of th, next applies ttac to the theorem(s) resulting from the stripping and then applies the resulting tactic to the goal.

In particular, when stripping a conjunctive theorem A'|- u /\ v, the tactic

```
ttac(u|-u) THEN ttac(v|-v)
```

resulting from applying ttac to the conjuncts, is applied to the goal. When stripping a disjunctive theorem A'|- u \/ v, the tactics resulting from applying ttac to the disjuncts, are applied to split the goal into two cases. That is, if

```
 A ?- t                          A ?- t
=========  ttac (u|-u)    and   =========  ttac (v|-v)
 A ?- t1                         A ?- t2
```

then:

```
      A ?- t
==================  STRIP_THM_THEN ttac (A'|- u \/ v)
 A ?- t1  A ?- t2
```

When stripping an existentially quantified theorem A'|- ?x.u, the tactic ttac(u|-u), resulting from applying ttac to the body of the existential quantification, is applied to the goal. That is, if:

```
 A ?- t
=========  ttac (u|-u)
 A ?- t1
```

then:

```
   A ?- t
=============  STRIP_THM_THEN ttac (A'|- ?x. u)
   A ?- t1
```

The assumptions of the theorem being split are not added to the assumptions of the goal(s) but are recorded in the proof. If A' is not a subset of the assumptions A of the goal (up to alpha-conversion), STRIP_THM_THEN ttac th results in an invalid tactic.

Failure

`STRIP_THM_THEN ttac th` fails if the conclusion of `th` is not a conjunction, a disjunction or an existentially quantified term. Failure also occurs if the application of `ttac` fails, after stripping the outer connective from the conclusion of `th`.

`STRIP_THM_THEN` is used enrich the assumptions of a goal with a stripped version of a previously-proved theorem.

See also

`Thm_cont.CHOOSE_THEN`, `Thm_cont.CONJUNCTS_THEN`, `Thm_cont.DISJ_CASES_THEN`, `Tactic.STRIP_ASSUME_TAC`

STRUCT_CASES_TAC (Tactic)

```
Tactic.STRUCT_CASES_TAC : thm_tactic
```

Performs very general structural case analysis.

When it is applied to a theorem of the form:

```
th = A' |- ?y11...y1m. (x=t1) /\ (B11 /\ ... /\ B1k) \/ ... \/
             ?yn1...ynp. (x=tn) /\ (Bn1 /\ ... /\ Bnp)
```

in which there may be no existential quantifiers where a 'vector' of them is shown above, `STRUCT_CASES_TAC th` splits a goal `A ?- s` into `n` subgoals as follows:

```
                              A ?- s
==============================================================
 A u {B11,...,B1k} ?- s[t1/x]  ...  A u {Bn1,...,Bnp} ?- s[tn/x]
```

that is, performs a case split over the possible constructions (the `ti`) of a term, providing as assumptions the given constraints, having split conjoined constraints into separate assumptions. Note that unless `A'` is a subset of `A`, this is an invalid tactic.

Failure

Fails unless the theorem has the above form, namely a conjunction of (possibly multiply existentially quantified) terms which assert the equality of the same variable `x` and the given terms.

Example

Suppose we have the goal:

```
?- ~(l:(*)list = []) ==> (LENGTH l) > 0
```

then we can get rid of the universal quantifier from the inbuilt list theorem `list_CASES`:

```
list_CASES = !l. (l = []) \/ (?t h. l = CONS h t)
```

and then use `STRUCT_CASES_TAC`. This amounts to applying the following tactic:

```
STRUCT_CASES_TAC (SPEC_ALL list_CASES)
```

which results in the following two subgoals:

```
?- ~(CONS h t = []) ==> (LENGTH(CONS h t)) > 0
```

```
?- ~([] = []) ==> (LENGTH[]) > 0
```

Note that this is a rather simple case, since there are no constraints, and therefore the resulting subgoals have no assumptions.

Generating a case split from the axioms specifying a structure.

See also

`Tactic.ASM_CASES_TAC`, `Tactic.BOOL_CASES_TAC`, `Tactic.COND_CASES_TAC`, `Tactic.DISJ_-CASES_TAC`

SUB_AND_COND_ELIM_CONV (Arith)

```
Arith.SUB_AND_COND_ELIM_CONV : conv
```

Eliminates natural number subtraction, PRE, and conditional statements from a formula.

This function eliminates natural number subtraction and the predecessor function, `PRE`, from a formula, but in doing so may generate conditional statements, so these are eliminated too. The conditional statements are moved up through the term and if at any point the branches of the conditional become Boolean-valued the conditional is eliminated. Subtraction operators are moved up until a relation (such as less-than) is reached. The subtraction can then be transformed into an addition. Provided the argument term is a formula, only an abstraction can prevent a conditional being moved up far enough to be eliminated. If the term is not a formula it may not be possible to eliminate the subtraction. The function is also incapable of eliminating subtractions that appear in arguments to functions other than the standard operators of arithmetic.

The function is not as delicate as it could be; it tries to eliminate all conditionals in a formula when it need only eliminate those that have to be removed in order to eliminate subtraction.

Failure

Never fails.

Example

```
#SUB_AND_COND_ELIM_CONV
# "((p + 3) <= n) ==> (!m. ((m = 0) => (n - 1) | (n - 2)) > p)";;
|- (p + 3) <= n ==> (!m. ((m = 0) => n - 1 | n - 2) > p) =
   (p + 3) <= n ==>
   (!m. (~(m = 0) \/ n > (1 + p)) /\ ((m = 0) \/ n > (2 + p)))

#SUB_AND_COND_ELIM_CONV
# "!f n. f ((SUC n = 0) => 0 | (SUC n - 1)) < (f n) + 1";;
|- (!f n. (f((SUC n = 0) => 0 | (SUC n) - 1)) < ((f n) + 1)) =
   (!f n.
     (~(SUC n = 0) \/ (f 0) < ((f n) + 1)) /\
```

```
     ((SUC n = 0) \/ (f((SUC n) - 1)) < ((f n) + 1)))

#SUB_AND_COND_ELIM_CONV
# "!f n. (\m. f ((m = 0) => 0 | (m - 1))) (SUC n) < (f n) + 1";;
|- (!f n. ((\m. f((m = 0) => 0 | m - 1))(SUC n)) < ((f n) + 1)) =
   (!f n. ((\m. ((m = 0) => f 0 | f(m - 1)))(SUC n)) < ((f n) + 1))
```

Useful as a preprocessor to decision procedures which do not allow natural number subtraction in their argument formula.

See also

`Arith.COND_ELIM_CONV`

SUB_CONV (Conv)

```
Conv.SUB_CONV : conv -> conv
```

Applies a conversion to the top-level subterms of a term.

For any conversion `c`, the function returned by `SUB_CONV c` is a conversion that applies `c` to all the top-level subterms of a term. Its implementation is

```
  fun SUB_CONV c = TRY_CONV (COMB_CONV c ORELSEC ABS_CONV c)
```

Example

If the conversion `c` maps `t` to `|- t = t'`, then `SUB_CONV c` maps an abstraction ``` ``\x.t`` ``` to the theorem:

```
   |- (\x.t) = (\x.t')
```

That is, `SUB_CONV c` ``` ``\x.t`` ``` applies `c` to the body of the abstraction ``` ``\x.t`` ```. If `c` is a conversion that maps ``` ``t1`` ``` to the theorem `|- t1 = t1'` and ``` ``t2`` ``` to the theorem `|- t2 = t2'`, then the conversion `SUB_CONV c` maps an application ``` ``t1 t2`` ``` to the theorem:

```
|- (t1 t2) = (t1' t2')
```

That is, `SUB_CONV c ``t1 t2`` applies `c` to the both the operator `t1` and the operand `t2` of the application `` ``t1 t2`` ``. Finally, for any conversion `c`, the function returned by `SUB_CONV c` acts as the identity conversion on variables and constants. That is, if `` ``t`` `` is a variable or constant, then `SUB_CONV c ``t`` ` raises the `UNCHANGED` exception.

Failure

`SUB_CONV c tm` fails if `tm` is an abstraction `` ``\x.t`` `` and the conversion `c` fails when applied to `t`, or if `tm` is an application `` ``t1 t2`` `` and the conversion `c` fails when applied to either `t1` or `t2`. The function returned by `SUB_CONV c` may also fail if the ML function `c:term->thm` is not, in fact, a conversion (i.e. a function that maps a term `t` to a theorem `|- t = t'`).

See also

`Conv.ABS_CONV`, `Conv.COMB_CONV`, `Conv.RAND_CONV`, `Conv.RATOR_CONV`

subgoal (bossLib)

```
bossLib.subgoal : term quotation -> tactic
```

Produces a subgoal.

A call to `subgoal q` is equivalent (by definition) to a call to `Q.SUBGOAL_THEN q STRIP_ASSUME_TAC`.

Failure

Fails if the provided quotation does not parse to a term of boolean type in the context of the current goal.

Comments

The `subgoal` tactic is also available via the name `sg`.

See also

bossLib.by, Q.SUBGOAL_THEN

SUBGOAL_THEN (Tactical)

```
Tactical.SUBGOAL_THEN : term -> thm_tactic -> tactic
```

Allows the user to introduce a lemma.

The user proposes a lemma and is then invited to prove it under the current assumptions. The lemma is then used with the thm_tactic to simplify the goal. That is, if

```
 A1 ?- t1
==========  f (u |- u)
 A2 ?- t2
```

then

```
      A1 ?- t1
====================  SUBGOAL_THEN u f
 A1 ?- u   A2 ?- t2
```

Typically f (u |- u) will be an invalid tactic because it would return a validation function which generated the theorem A1,u |- t1 from the theorem A2 |- t2. Nonetheless, the tactic SUBGOAL_THEN u f is valid because of the extra sub-goal where u must be proved.

Failure

SUBGOAL_THEN will fail if an attempt is made to use a nonboolean term as a lemma.

When combined with rotate, SUBGOAL_THEN allows the user to defer some part of a proof and to continue with another part. SUBGOAL_THEN is most convenient when the tactic solves the original goal, leaving only the subgoal. For example, suppose the user wishes to prove the goal

```
{n = SUC m} ?- (0 = n) ==> t
```

Using SUBGOAL_THEN to focus on the case in which ~(n = 0), rewriting establishes it truth, leaving only the proof that ~(n = 0). That is,

```
SUBGOAL_THEN (Term `~(0 = n)`) (fn th => REWRITE_TAC [th])
```

generates the following subgoals:

```
{n = SUC m} ?-  ~(0 = n)
?- T
```

Comments

Some users may expect the generated tactic to be f (A1 |- u), rather than f (u |- u).

SUBS (Drule)

```
Drule.SUBS : (thm list -> thm -> thm)
```

Makes simple term substitutions in a theorem using a given list of theorems.

Term substitution in HOL is performed by replacing free subterms according to the transformations specified by a list of equational theorems. Given a list of theorems A1|-t1=v1,...,An|-tn=vn and a theorem A|-t, SUBS simultaneously replaces each free occurrence of ti in t with vi:

```
        A1|-t1=v1 ... An|-tn=vn    A|-t
-------------------------------------------  SUBS[A1|-t1=v1;...;An|-tn=vn]
 A1 u ... u An u A |- t[v1,...,vn/t1,...,tn]       (A|-t)
```

No matching is involved; the occurrence of each ti being substituted for must be a free in t (see SUBST_MATCH). An occurrence which is not free can be substituted by using rewriting rules such as REWRITE_RULE, PURE_REWRITE_RULE and ONCE_REWRITE_RULE.

Failure

SUBS [th1,...,thn] (A|-t) fails if the conclusion of each theorem in the list is not an equation. No change is made to the theorem A |- t if no occurrence of any left-hand side of the supplied equations appears in t.

Example

Substitutions are made with the theorems

```
- val thm1 = SPECL [Term`m:num`, Term`n:num`] arithmeticTheory.ADD_SYM
  val thm2 = CONJUNCT1 arithmeticTheory.ADD_CLAUSES;
> val thm1 = |- m + n = n + m : thm
  val thm2 = |- 0 + m = m : thm
```

depending on the occurrence of free subterms

```
- SUBS [thm1, thm2] (ASSUME (Term `(n + 0) + (0 + m) = m + n`));
> val it =  [.] |- n + 0 + m = n + m : thm

- SUBS [thm1, thm2] (ASSUME (Term `!n. (n + 0) + (0 + m) = m + n`));
> val it =  [.] |- !n. n + 0 + m = m + n : thm
```

SUBS can sometimes be used when rewriting (for example, with REWRITE_RULE) would diverge and term instantiation is not needed. Moreover, applying the substitution rules is often much faster than using the rewriting rules.

See also

Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.REWRITE_RULE, Thm.SUBST, Rewrite.SUBST_MATCH, Drule.SUBS_OCCS

SUBS_OCCS (Drule)

```
Drule.SUBS_OCCS : (int list * thm) list -> thm -> thm
```

Makes substitutions in a theorem at specific occurrences of a term, using a list of equational theorems.

Given a list `(l1,A1|-t1=v1),...,(ln,An|-tn=vn)` and a theorem `(A|-t)`, SUBS_OCCS simultaneously replaces each `ti` in `t` with `vi`, at the occurrences specified by the integers in the list `li = [o1,...,ok]` for each theorem `Ai|-ti=vi`.

```
  (l1,A1|-t1=v1) ... (ln,An|-tn=vn)  A|-t
------------------------------------------  SUBS_OCCS[(l1,A1|-t1=v1),...,
 A1 u ... An u A |- t[v1,...,vn/t1,...,tn]            (ln,An|-tn=vn)] (A|-t)
```

Failure

`SUBS_OCCS [(l1,th1),...,(ln,thn)] (A|-t)` fails if the conclusion of any theorem in the list is not an equation. No change is made to the theorem if the supplied occurrences `li` of the left-hand side of the conclusion of `thi` do not appear in `t`.

Example

The commutative law for addition

```
- val thm = SPECL [Term `m:num`, Term`n:num`] arithmeticTheory.ADD_SYM;
> val thm = |- m + n = n + m : thm
```

can be used for substituting only the second occurrence of the subterm `m + n`

```
- SUBS_OCCS [([2],thm)]
            (ASSUME (Term `(n + m) + (m + n) = (m + n) + (m + n)`));
> val it =  [.] |- n + m + (m + n) = n + m + (m + n) : thm
```

`SUBS_OCCS` is used when rewriting at specific occurrences of a term, and rules such as `REWRITE_RULE`, `PURE_REWRITE_RULE`, `ONCE_REWRITE_RULE`, and `SUBS` are too extensive or would diverge.

See also

`Rewrite.ONCE_REWRITE_RULE`, `Rewrite.PURE_REWRITE_RULE`, `Rewrite.REWRITE_RULE`, `Drule.SUBS`, `Thm.SUBST`, `Rewrite.SUBST_MATCH`

subst (Lib)

```
Lib.type ('a,'b) subst
```

Type abbreviation for substitutions.

The type `('a,'b) subst` abbreviates the type `{redex,residue} list`, in which `redex` has type `'a` and `residue` has type `'b`. Usually, a `{redex,residue}` pair in a substition is interpreted as 'replace occurrences of `redex` by `residue`'.

Comments

The different types of `redex` and `residue` components allows flexibility, as in the rule of inference `SUBST`, which takes a `(term,thm) subst` argument.

See also

`Lib.|->`, `Term.subst`, `Term.inst`, `Thm.SUBST`

subst (Term)

```
Term.subst : (term,term) subst -> term -> term
```

Substitutes terms in a term.

Given a "(term,term) subst" (a list of `{redex, residue}` records) and a term `tm`, `subst` attempts to replace each free occurrence of a `redex` in `tm` by its associated `residue`. The substitution is done in parallel, i.e., once a redex has been replaced by its residue, at some place in the term, that residue at that place will not itself be replaced in the current call. When necessary, renaming of bound variables in `tm` is done to avoid capturing the free variables of an incoming residue.

Failure

Failure occurs if there exists a `{redex, residue}` record in the substitution such that the types of the `redex` and `residue` are not equal.

Example

```
> load "arithmeticTheory";
val it = (): unit

> subst [Term`SUC 0` |-> Term`1`]
        (Term`SUC(SUC 0)`);
val it = "SUC 1": term
```

```
> subst [Term`SUC 0` |-> Term`1`,
         Term`SUC 1` |-> Term`2`]
        (Term`SUC(SUC 0)`);
val it = “SUC 1”: term

> subst [Term`SUC 0` |-> Term`1`,
         Term`SUC 1` |-> Term`2`]
        (Term`SUC(SUC 0) = SUC 1`);
val it = “SUC 1 = 2”: term

> subst [Term`b:num` |-> Term`a:num`]
        (Term`\a:num. b:num`);
val it = “λa'. a”: term

> subst [Term`flip:'a` |-> Term`foo:'a`]
       (Term`waddle:'a`);
Exception- HOL_ERR
  (at Preterm.type-analysis: on line 1, characters 12-15:

Type constraint failure:
  Term: flip :(α -> β -> γ) -> β -> α -> γ
  Constraint: :α
) raised
```

See also

`Term.inst`, `Thm.SUBST`, `Drule.SUBS`, `Lib.|->`

SUBST (Thm)

```
Thm.SUBST : (term,thm) subst -> term -> thm -> thm
```

Makes a set of parallel substitutions in a theorem.

Implements the following rule of simultaneous substitution

```
    A1 |- t1 = u1 ,  ... , An |- tn = un ,    A |- t[t1,...,tn]
```

```
   ------------------------------------------------------------
        A u A1 u ... u An |- t[u1,...,un]
```

Evaluating

```
SUBST [x1 |-> (A1 |- t1=u1) ,..., xn |-> (An |- tn=un)]
      t[x1,...,xn]
      (A |- t[t1,...,tn])
```

returns the theorem `A u A1 u ... u An |- t[u1,...,un]` (perhaps with fewer assumptions, see paragraph below). The term argument `t[x1,...,xn]` is a template which should match the conclusion of the theorem being substituted into, with the variables `x1`, `...` , `xn` marking those places where occurrences of `t1`, `...` , `tn` are to be replaced by the terms `u1`, `...` , `un`, respectively. The occurrence of `ti` at the places marked by `xi` must be free (i.e. `ti` must not contain any bound variables). `SUBST` automatically renames bound variables to prevent free variables in `ui` becoming bound after substitution.

The assumptions of the returned theorem may not contain all the assumptions `A1 u ... u An` if some of them are not required. In particular, if the theorem `Ak |- tk = uk` is unnecessary because `xk` does not appear in the template, then `Ak` is not be added to the assumptions of the returned theorem.

Failure

If the template does not match the conclusion of the hypothesis, or the terms in the conclusion marked by the variables `x1`, `...` , `xn` in the template are not identical to the left hand sides of the supplied equations (i.e. the terms `t1`, `...` , `tn`).

Example

```
> val x = “x:num”
  and y = “y:num”
  and th0 = SPEC “0” arithmeticTheory.ADD1
  and th1 = SPEC “1” arithmeticTheory.ADD1;
val th0 = ⊢ SUC 0 = 0 + 1: thm
val th1 = ⊢ SUC 1 = 1 + 1: thm
val x = “x”: term
val y = “y”: term

> SUBST [x |-> th0, y |-> th1]
```

```
       “(x+y) > SUC 0”
       (ASSUME “(SUC 0 + SUC 1) > SUC 0”);
val it =  [.] ⊢ 0 + 1 + (1 + 1) > SUC 0: thm

> SUBST [x |-> th0, y |-> th1]
       “(SUC 0 + y) > SUC 0”
       (ASSUME “(SUC 0 + SUC 1) > SUC 0”);
val it =  [.] ⊢ SUC 0 + (1 + 1) > SUC 0: thm

> SUBST [x |-> th0, y |-> th1]
       “(x+y) > x”
       (ASSUME “(SUC 0 + SUC 1) > SUC 0”);
val it =  [.] ⊢ 0 + 1 + (1 + 1) > 0 + 1: thm
```

Comments

SUBST is perhaps overly complex for a primitive rule of inference.

For substituting at selected occurrences. Often useful for writing special purpose derived inference rules.

See also

Thm.INST, Drule.SUBS, Drule.SUBST_CONV, Lib.|->

SUBST1_TAC (Tactic)

```
Tactic.SUBST1_TAC : thm_tactic
```

Makes a simple term substitution in a goal using a single equational theorem.

Given a theorem A'|-u=v and a goal (A,t), the tactic SUBST1_TAC (A'|-u=v) rewrites the term t into t[v/u], by substituting v for each free occurrence of u in t:

```
      A ?- t
   =============  SUBST1_TAC (A'|-u=v)
    A ?- t[v/u]
```

The assumptions of the theorem used to substitute with are not added to the assumptions of the goal but are recorded in the proof. If `A'` is not a subset of the assumptions `A` of the goal (up to alpha-conversion), then `SUBST1_TAC (A'|-u=v)` results in an invalid tactic.

`SUBST1_TAC` automatically renames bound variables to prevent free variables in `v` becoming bound after substitution.

Failure

`SUBST1_TAC th (A,t)` fails if the conclusion of `th` is not an equation. No change is made to the goal if no free occurrence of the left-hand side of `th` appears in `t`.

Example

When trying to solve the goal

```
?- m * n = (n * (m - 1)) + n
```

substituting with the commutative law for multiplication

```
SUBST1_TAC (SPECL ["m:num"; "n:num"] MULT_SYM)
```

results in the goal

```
?- n * m = (n * (m - 1)) + n
```

`SUBST1_TAC` is used when rewriting with a single theorem using tactics such as `REWRITE_TAC` is too expensive or would diverge. Applying `SUBST1_TAC` is also much faster than using rewriting tactics.

See also

`Rewrite.ONCE_REWRITE_TAC`, `Rewrite.PURE_REWRITE_TAC`, `Rewrite.REWRITE_TAC`, `Tactic.SUBST_ALL_TAC`, `Tactic.SUBST_TAC`

SUBST_ALL_TAC (Tactic)

`Tactic.SUBST_ALL_TAC : thm_tactic`

Substitutes using a single equation in both the assumptions and conclusion of a goal.

SUBST_ALL_TAC breaches the style of natural deduction, where the assumptions are kept fixed. Given a theorem A|-u=v and a goal ([t1;...;tn], t), SUBST_ALL_TAC (A|-u=v) transforms the assumptions t1,...,tn and the term t into t1[v/u],...,tn[v/u] and t[v/u] respectively, by substituting v for each free occurrence of u in both the assumptions and the conclusion of the goal.

```
        {t1,...,tn} ?- t
================================  SUBST_ALL_TAC (A|-u=v)
 {t1[v/u],...,tn[v/u]} ?- t[v/u]
```

The assumptions of the theorem used to substitute with are not added to the assumptions of the goal, but they are recorded in the proof. If A is not a subset of the assumptions of the goal (up to alpha-conversion), then SUBST_ALL_TAC (A|-u=v) results in an invalid tactic.

SUBST_ALL_TAC automatically renames bound variables to prevent free variables in v becoming bound after substitution.

Failure

SUBST_ALL_TAC th (A,t) fails if the conclusion of th is not an equation. No change is made to the goal if no occurrence of the left-hand side of th appears free in (A,t).

Example

Simplifying both the assumption and the term in the goal

```
{0 + m = n} ?- 0 + (0 + m) = n
```

by substituting with the theorem |- 0 + m = m for addition

```
SUBST_ALL_TAC (CONJUNCT1 ADD_CLAUSES)
```

results in the goal

```
{m = n} ?- 0 + m = n
```

See also

Rewrite.ONCE_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST1_TAC, Tactic.SUBST_TAC

subst_assoc (Lib)

```
Lib.subst_assoc : ('a -> bool) -> ('a,'b)subst -> 'b option
```

Treats a substitution as an association list.

An application `subst_assoc P [{redex_1,residue_1},...,{redex_n,residue_n}]` returns `SOME residue_i` if `P` holds of `redex_i`, and did not hold (or fail) for `{redex_j | 1 <= j < i}`. If `P` holds for none of the redexes in the substitution, `NONE` is returned.

Failure

If `P redex_i` fails for some `redex` encountered in the left-to-right traversal of the substitution.

Example

```
> subst_assoc is_abs [T |-> F, Term `\x.x` |-> Term `combin$I`];
val it = SOME (“I”): term option
```

See also

Lib.assoc, Lib.rev_assoc, Lib.assoc1, Lib.assoc2, Lib.|->

SUBST_CONV (Drule)

```
Drule.SUBST_CONV : {redex :term, residue :thm} list -> term -> conv
```

Makes substitutions in a term at selected occurrences of subterms, using a list of theorems.

SUBST_CONV implements the following rule of simultaneous substitution

```
                A1 |- t1 = v1 ... An |- tn = vn
-------------------------------------------------------------------
 A1 u ... u An |- t[t1,...,tn/x1,...,xn] = t[v1,...,vn/x1,...,xn]
```

The first argument to SUBST_CONV is a list `[{redex=x1, residue = A1|-t1=v1},...,{redex = xn, residue = An|-tn=vn}]`. The second argument is a template term `t[x1,...,xn]`, in which the variables `x1,...,xn` are used to mark those places where occurrences of `t1,...,tn` are to be replaced with the terms `v1,...,vn`, respectively. Thus, evaluating

```
SUBST_CONV [{redex = x1, residue = A1|-t1=v1},...,
            {redex = xn, residue = An|-tn=vn}]
           t[x1,...,xn]
           t[t1,...,tn/x1,...,xn]
```

returns the theorem

```
A1 u ... u An |- t[t1,...,tn/x1,...,xn] = t[v1,...,vn/x1,...,xn]
```

The occurrence of `ti` at the places marked by the variable `xi` must be free (i.e. `ti` must not contain any bound variables). SUBST_CONV automatically renames bound variables to prevent free variables in `vi` becoming bound after substitution.

Failure

`SUBST_CONV [{redex=x1,residue=th1},...,{redex=xn,residue=thn}] t[x1,...,xn] t'` fails if the conclusion of any theorem `thi` in the list is not an equation; or if the template `t[x1,...,xn]` does not match the term `t'`; or if and term `ti` in `t'` marked by the variable `xi` in the template, is not identical to the left-hand side of the conclusion of the theorem `thi`.

Example

The values

```
> val ADD1 = arithmeticTheory.ADD1
val ADD1 = ⊢ ∀m. SUC m = m + 1: thm
```

```
> val thm0 = SPEC (Term`0`) ADD1
  and thm1 = SPEC (Term`1`) ADD1
  and x = Term`x:num` and y = Term`y:num`;
val thm0 = ⊢ SUC 0 = 0 + 1: thm
val thm1 = ⊢ SUC 1 = 1 + 1: thm
val x = “x”: term
val y = “y”: term
```

can be used to substitute selected occurrences of the terms `SUC 0` and `SUC 1`

```
> SUBST_CONV [{redex=x, residue=thm0},{redex=y,residue=thm1}]
             “x + y > SUC 1”
             “SUC 0 + SUC 1 > SUC 1”;
val it = ⊢ SUC 0 + SUC 1 > SUC 1 ⇔ 0 + 1 + (1 + 1) > SUC 1: thm
```

The `|->` syntax can also be used:

```
> SUBST_CONV [x |-> thm0, y |-> thm1]
             “x + y > SUC 1”
             “SUC 0 + SUC 1 > SUC 1”;
val it = ⊢ SUC 0 + SUC 1 > SUC 1 ⇔ 0 + 1 + (1 + 1) > SUC 1: thm
```

`SUBST_CONV` is used when substituting at selected occurrences of terms and using rewriting rules/conversions is too extensive.

See also

`Conv.REWR_CONV`, `Drule.SUBS`, `Thm.SUBST`, `Drule.SUBS_OCCS`, `Lib.|->`

SUBST_MATCH (Rewrite)

```
Rewrite.SUBST_MATCH : (thm -> thm -> thm)
```

Substitutes in one theorem using another, equational, theorem.

Given the theorems `A|-u=v` and `A'|-t`, `SUBST_MATCH (A|-u=v) (A'|-t)` searches for one free instance of `u` in `t`, according to a top-down left-to-right search strategy, and then substitutes the corresponding instance of `v`.

```
 A |- u=v   A' |- t
--------------------  SUBST_MATCH (A|-u=v) (A'|-t)
  A u A' |- t[v/u]
```

SUBST_MATCH allows only a free instance of u to be substituted for in t. An instance which contain bound variables can be substituted for by using rewriting rules such as REWRITE_RULE, PURE_REWRITE_RULE and ONCE_REWRITE_RULE.

Failure

SUBST_MATCH th1 th2 fails if the conclusion of the theorem th1 is not an equation. Moreover, SUBST_MATCH (A|-u=v) (A'|-t) fails if no instance of u occurs in t, since the matching algorithm fails. No change is made to the theorem (A'|-t) if instances of u occur in t, but they are not free (see SUBS).

Example

The commutative law for addition

```
- val thm1 = SPECL [Term `m:num`, Term `n:num`] arithmeticTheory.ADD_SYM;
> val thm1 = |- m + n = n + m : thm
```

is used to apply substitutions, depending on the occurrence of free instances

```
- SUBST_MATCH thm1 (ASSUME (Term `(n + 1) + (m - 1) = m + n`));
> val it =  [.] |- m - 1 + (n + 1) = m + n : thm

- SUBST_MATCH thm1 (ASSUME (Term `!n. (n + 1) + (m - 1) = m + n`));
> val it =  [.] |- !n. n + 1 + (m - 1) = m + n : thm
```

SUBST_MATCH is used when rewriting with the rules such as REWRITE_RULE, using a single theorem is too extensive or would diverge. Moreover, applying SUBST_MATCH can be much faster than using the rewriting rules.

See also

Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Rewrite.REWRITE_RULE, Drule.SUBS, Thm.SUBST

subst_occs (HolKernel)

```
HolKernel.subst_occs : int list list -> (term,term) subst -> term -> term
```

Substitutes for particular occurrences of subterms of a given term.

For each `{redex,residue}` in the second argument, there should be a corresponding integer list `l_i` in the first argument that specifies which free occurrences of `redex_i` in the third argument should be substituted by `residue_i`.

Failure

Failure occurs if any substitution fails, or if the length of the first argument is not equal to the length of the substitution. In other words, every substitution pair should be accompanied by a list specifying when the substitution is applicable.

Example

```
> subst_occs [[1,3]] [Term `SUC 0` |-> Term `1`]
            (Term `SUC 0 + SUC 0 = SUC(SUC 0)`);
val it = “1 + SUC 0 = SUC 1”: term

> subst_occs [[1],[1]] [Term `SUC 0` |-> Term `1`,
                        Term `SUC 1` |-> Term `2`]
            (Term `SUC(SUC 0) = SUC 1`);
val it = “SUC 1 = 2”: term

> subst_occs [[1],[1]] [Term`SUC(SUC 0)` |-> Term `2`,
                        Term`SUC 0`      |-> Term `1`]
            (Term`SUC(SUC 0) = SUC 0`);
val it = “2 = 1”: term
```

See also

`Term.subst`, `Lib.|->`

SUBST_OCCS_TAC (Tactic)

```
Tactic.SUBST_OCCS_TAC : (int list * thm) list -> tactic
```

Makes substitutions in a goal at specific occurrences of a term, using a list of theorems.

Given a list `(l1,A1|-t1=u1),...,(ln,An|-tn=un)` and a goal `(A,t)`, `SUBST_OCCS_TAC` replaces each `ti` in `t` with `ui`, simultaneously, at the occurrences specified by the integers in the list `li = [o1,...,ok]` for each theorem `Ai|-ti=ui`.

```
            A ?- t
============================  SUBST_OCCS_TAC [(l1,A1|-t1=u1),...,
 A ?- t[u1,...,un/t1,...,tn]                  (ln,An|-tn=un)]
```

The assumptions of the theorems used to substitute with are not added to the assumptions `A` of the goal, but they are recorded in the proof. If any `Ai` is not a subset of `A` (up to alpha-conversion), `SUBST_OCCS_TAC [(l1,A1|-t1=u1),...,(ln,An|-tn=un)]` results in an invalid tactic.

`SUBST_OCCS_TAC` automatically renames bound variables to prevent free variables in `ui` becoming bound after substitution.

Failure

`SUBST_OCCS_TAC [(l1,th1),...,(ln,thn)] (A,t)` fails if the conclusion of any theorem in the list is not an equation. No change is made to the goal if the supplied occurrences `li` of the left-hand side of the conclusion of `thi` do not appear in `t`.

Example

When trying to solve the goal

```
?- (m + n) + (n + m) = (m + n) + (m + n)
```

applying the commutative law for addition on the third occurrence of the subterm `m + n`

```
SUBST_OCCS_TAC [([3], SPECL [Term `m:num`, Term `n:num`]
                            arithmeticTheory.ADD_SYM)]
```

results in the goal

```
?- (m + n) + (n + m) = (m + n) + (n + m)
```

SUBST_OCCS_TAC is used when rewriting a goal at specific occurrences of a term, and when rewriting tactics such as REWRITE_TAC, PURE_REWRITE_TAC, ONCE_REWRITE_TAC, SUBST_TAC, etc. are too extensive or would diverge.

See also

Rewrite.ONCE_REWRITE_TAC, Rewrite.PURE_REWRITE_TAC, Rewrite.REWRITE_TAC, Tactic.SUBST1_TAC, Tactic.SUBST_TAC

SUBST_TAC (Tactic)

```
Tactic.SUBST_TAC : (thm list -> tactic)
```

Makes term substitutions in a goal using a list of theorems.

Given a list of theorems A1|-u1=v1,...,An|-un=vn and a goal (A,t), SUBST_TAC rewrites the term t into the term t[v1,...,vn/u1,...,un] by simultaneously substituting vi for each occurrence of ui in t with vi:

```
            A ?- t
============================  SUBST_TAC [A1|-u1=v1,...,An|-un=vn]
 A ?- t[v1,...,vn/u1,...,un]
```

The assumptions of the theorems used to substitute with are not added to the assumptions A of the goal, while they are recorded in the proof. If any Ai is not a subset of A (up to alpha-conversion), then SUBST_TAC [A1|-u1=v1,...,An|-un=vn] results in an invalid tactic.

SUBST_TAC automatically renames bound variables to prevent free variables in vi becoming bound after substitution.

Failure

SUBST_TAC [th1,...,thn] (A,t) fails if the conclusion of any theorem in the list is not an equation. No change is made to the goal if no occurrence of the left-hand side of the conclusion of thi appears in t.

Example

When trying to solve the goal

```
?- (n + 0) + (0 + m) = m + n
```

by substituting with the theorems

```
- val thm1 = SPEC_ALL arithmeticTheory.ADD_SYM
  val thm2 = CONJUNCT1 arithmeticTheory.ADD_CLAUSES;
thm1 = |- m + n = n + m
thm2 = |- 0 + m = m
```

applying `SUBST_TAC [thm1, thm2]` results in the goal

```
?- (n + 0) + m = n + m
```

`SUBST_TAC` is used when rewriting (for example, with `REWRITE_TAC`) is extensive or would diverge. Substituting is also much faster than rewriting.

See also

`Rewrite.ONCE_REWRITE_TAC`, `Rewrite.PURE_REWRITE_TAC`, `Rewrite.REWRITE_TAC`, `Tactic.SUBST1_TAC`, `Tactic.SUBST_ALL_TAC`

subtract (Lib)

```
Lib.subtract : ''a list -> ''a list -> ''a list
```

Computes the set-theoretic difference of two ‘sets’.

Behaves exactly like `set_diff`.

See also

`Lib.set_diff`

SUC_CONV (reduceLib)

```
reduceLib.SUC_CONV : conv
```

Calculates by inference the successor of a numeral.

If `n` is a numeral (e.g. `0`, `1`, `2`, `3`,...), then `SUC_CONV "SUC n"` returns the theorem:

```
|- SUC n = s
```

where `s` is the numeral that denotes the successor of the natural number denoted by `n`.

Failure

`SUC_CONV tm` fails unless `tm` is of the form `"SUC n"`, where `n` is a numeral.

Example

```
#SUC_CONV "SUC 33";;
|- SUC 33 = 34
```

SUC_TO_NUMERAL_DEFN_CONV (numLib)

```
numLib.SUC_TO_NUMERAL_DEFN_CONV : conv
```

Translates equations using `SUC n` to use numeral constructors instead.

This conversion modifies conjunctions of universally quantified equations so that any argument terms of the form `SUC x` on the LHS of the equations (with `x` one of the equation's universally quantified variables), are translated to a form appropriate for rewriting when the argument term is a numeral.

This procedure uses the following theorem:

```
  |- !f g. (!n. f (SUC n) = g n (SUC n)) =
          (!n. f (NUMERAL (BIT1 n)) =
               g (NUMERAL (BIT1 n)) (NUMERAL (BIT1 n) - 1)) /\
          (!n. f (NUMERAL (BIT2 n)) =
               g (NUMERAL (BIT2 n)) (NUMERAL (BIT1 n)))
```

Example

```
> CONV_RULE numLib.SUC_TO_NUMERAL_DEFN_CONV arithmeticTheory.FACT;
val it =
   ⊢ FACT 0 = 1 ∧
     (∀n. FACT (NUMERAL (BIT1 n)) =
          NUMERAL (BIT1 n) * FACT (NUMERAL (BIT1 n) − 1)) ∧
     ∀n. FACT (NUMERAL (BIT2 n)) = NUMERAL (BIT2 n) * FACT (NUMERAL (BIT1 n)):
   thm
```

Failure

Fails if the input term is not the conjunction of universally quantified equations, where there may be just one conjunct, and where equations may have no quantification at all. Those conjuncts which don't involve terms of the form `SUC x` are returned unchanged.

Comments

Useful for translating definitions over numbers (which often involve `SUC` terms), into a form that can be used to work with numerals easily.

See also

`numLib.num_CONV`

SUFF_TAC (Tactic)

```
Tactic.SUFF_TAC : term -> tactic
```

Introduces an implicational subgoal.

A call to SUFF_TAC t on the goal asl ?- g introduces two subgoals: asl ?- t ==> g and asl ?- t. At a high-level, the user's claim is that term t suffices (hence the name) to prove the goal. The first new goal to be discharged is the check for this; the second is to actually show t.

Failure

Will fail if t is not of type :bool.

See also

Tactic.SUBGOAL_THEN, bossLib.suffices_by

suffices_by (bossLib)

```
op bossLib.suffices_by : term quotation * tactic -> tactic
```

Replace the goal's conclusion with a sufficient alternative.

A call to the tactic q suffices_by tac will first attempt to parse the quotation q in the context of the current goal. Assuming this generates a term qt of boolean type, it will then generate two sub-goals. Assuming the current goal is asl ?- g, the first new sub-goal will be that qt implies g, thus asl ?- qt ==> g. The second goal will be asl ?- qt.

The system next applies tac to the first sub-goal (the implication). If tac solves the goal (the common or at least, desired, case), the user will then be presented with one goal, where the original g has been replaced with qt. In this way, the user has adjusted the goal, replacing the old g with a qt that is sufficient to prove it.

Failure

A call to q suffices_by tac will fail if the quotation q does not parse to a term of boolean type. This parsing is done in the context of the whole goal (asl,g), using the parse_in_context function. The call will also fail if tac does not solve the newly generated subgoal.

Example

If the current goal is

```
f n m = f m n
------------------------------------
  0.  m <= n
  1.  n <= m
```

then the tactic `` `m = n` suffices_by SIMP_TAC bool_ss [] `` will result in the goal

```
m = n
------------------------------------
  0.  m <= n
  1.  n <= m
```

where the call to `SIMP_TAC` has successfully proved the theorem

```
|- (m = n) ==> (f m n = f n m)
```

eliminating the first of the two sub-goals that was generated.

Comments

The tactic `suffices_by` is designed to support a backwards style of reasoning. Superficially, it appears to be dual to the tactic `by`, which provides a forward-reasoning facility. In fact, both are implementing a backwards application of the sequent calculus's "cut" rule; the difference is which of the two premises to the rule is worked on by the provided tactics.

See also

`bossLib.by`, `Parse.parse_in_context`, `Tactic.SUFF_TAC`

SUM_CONV (listLib)

`listLib.SUM_CONV : conv`

Computes by inference the result of summing the elements of a list.

For any object language list of the form "`[x1;x2;...;xn]`", where `x1`, `x2`, `...`, `xn` are numeral constants, the result of evaluating

```
SUM_CONV "SUM [x1;x2;...;xn]"
```

is the theorem

```
|- SUM [x1;x2;...;xn] = n
```

where `n` is the numeral constant that denotes the sum of the elements of the list.

Example

Evaluating `SUM_CONV "SUM [0;1;2;3]"` will return the following theorem:

```
|- SUM [0;1;2;3] = 6
```

Failure

`SUM_CONV tm` fails if `tm` is not of the form described above.

See also

`listLib.FOLDL_CONV`, `listLib.FOLDR_CONV`, `listLib.list_FOLD_CONV`

suspend (bossLib)

```
bossLib.suspend : string -> tactic
```

Suspends a goal (proving it), so that it can be resumed later.

Given any goal `g`, a call to `suspend nm` proves that goal by assuming an encoding of the goal as a hypothesis. This encoding includes the name `nm`, so that the resulting theorem is effectively

```
[e(nm,g)] |- g
```

Of course, if the goal is a subgoal in a larger proof, this new hypothesis will propagate out to the final theorem proved, recording the point at which the wider goal was suspended. If `suspend` is called multiple times within the wider goal, multiple such hypotheses will be created.

Failure

Never fails.

Comment

The `VALID` tactical considers `suspend`-encoded assumptions legitimate, not flagging them as invalid and causing a tactic that uses them to fail.

See also

`bossLib.cheat`, `Tactical.VALID`.

swap (Lib)

```
Lib.swap : 'a * 'b -> 'b * 'a
```

Swaps the two components of a pair.

`swap (x,y)` returns `(y,x)`.

Failure

Never fails.

See also

`Lib.fst`, `Lib.snd`, `Lib.pair`, `Lib.rpair`

SWAP_EXISTS_CONV (Conv)

```
Conv.SWAP_EXISTS_CONV : conv
```

Interchanges the order of two existentially quantified variables.

When applied to a term argument of the form `?x y. P`, the conversion `SWAP_EXISTS_CONV` returns the theorem:

```
|- (?x y. P) = (?y x. P)
```

Failure

`SWAP_EXISTS_CONV` fails if applied to a term that is not of the form `?x y. P`.

SWAP_PEXISTS_CONV (PairRules)

```
PairRules.SWAP_PEXISTS_CONV : conv
```

Interchanges the order of two existentially quantified pairs.

When applied to a term argument of the form `?p q. t`, the conversion `SWAP_PEXISTS_CONV` returns the theorem:

```
|- (?p q. t) = (?q t. t)
```

Failure

`SWAP_PEXISTS_CONV` fails if applied to a term that is not of the form `?p q. t`.

See also

`Conv.SWAP_EXISTS_CONV`, `PairRules.SWAP_PFORALL_CONV`

SWAP_PFORALL_CONV (PairRules)

```
PairRules.SWAP_PFORALL_CONV : conv
```

Interchanges the order of two universally quantified pairs.

When applied to a term argument of the form `!p q. t`, the conversion `SWAP_PFORALL_CONV` returns the theorem:

```
|- (!p q. t) = (!q p. t)
```

Failure

`SWAP_PFORALL_CONV` fails if applied to a term that is not of the form `!p q. t`.

See also

`PairRules.SWAP_PEXISTS_CONV`

SYM (Thm)

```
Thm.SYM : thm -> thm
```

Swaps left-hand and right-hand sides of an equation.

When applied to a theorem `A |- t1 = t2`, the inference rule `SYM` returns `A |- t2 = t1`.

```
 A |- t1 = t2
--------------  SYM
 A |- t2 = t1
```

Failure

Fails unless the theorem is equational.

See also

Conv.GSYM, Drule.NOT_EQ_SYM, Thm.REFL, Tacic.SYM_TAC

SYM_CONV (Conv)

```
Conv.SYM_CONV : conv
```

Interchanges the left and right-hand sides of an equation.

When applied to an equational term `t1 = t2`, the conversion SYM_CONV returns the theorem:

```
|- (t1 = t2) = (t2 = t1)
```

Failure

Fails if applied to a term that is not an equation.

See also

Thm.SYM

SYM_TAC (Tactic)

```
Tactic.SYM_TAC : tactic
```

Flips an equality at the top-level of a goal

An application of SYM_TAC behaves as follows:

```
    G  ?-   x = y
================   SYM_TAC
    G  ?-   y = x
```

Failure

Fails if the goal is not an equality.

Comments

Also available as the alias `sym_tac`.

See also

`Tactic.REFL_TAC`, `Thm.SYM`

syntax_fns (HolKernel)

```
HolKernel.syntax_fns : {dest: term -> exn -> term -> 'a, make: term -> 'b -> term, n: in
             term * ('b -> term) * (term -> 'a) * (term -> bool)
```

Helps in providing syntax support for theory constants.

`syntax_fns syntax thy name` returns a 4-tuple, consisting of: a term, a term destructor function, a term constructor function and a term recogniser function. These provide syntax support for operation `name` from theory `thy`. The record argument `syntax` consists of `dest: term -> exn -> term -> 'a` and `make: term -> 'b -> term` functions, together with an "expected arity" value `n`. Through a sequence of instantiations, the `syntax_fns` function can be used to quickly and reliably write a `thySyntax.sml` file.

Example

To provide syntax support for unary operations from the theory `num`, one can use the following function:

```
> val num1 = HolKernel.syntax_fns {n = 1, make = HolKernel.mk_monop, dest = HolKernel.de
val num1 = fn:
   string -> term * (term -> term) * (term -> term) * (term -> bool)
```

The following call then provides support for the `SUC` constant:

```
> val (suc_tm, mk_suc, dest_suc, is_suc) = num1 "SUC";
val dest_suc = fn: term -> term
val is_suc = fn: term -> bool
val mk_suc = fn: term -> term
val suc_tm = “SUC”: term
```

A SUC term can be constructed with

```
> val tm = mk_suc ``4n``;
val tm = “SUC 4”: term
```

The resulting term tm can be identified and destructed as follows:

```
> is_suc tm;
val it = true: bool
> val v = dest_suc tm;
val v = “4”: term
```

A standard error message is raised when dest_suc fails, e.g.

```
> is_suc ``SUC``;
val it = false: bool
> val v = dest_suc ``SUC``;
Exception- HOL_ERR at numSyntax.dest_SUC: raised
```

The value n in the call to syntax_fns acts purely as a sanity check. For example, the following fails because SUC is not a binary operation:

```
> HolKernel.syntax_fns {n = 2, make = HolKernel.mk_binop, dest = HolKernel.dest_bin
Exception- HOL_ERR (at numSyntax.syntax_fns: bad number of arguments) raised
```

Typically, the dest and make functions will be complementary (with type variables 'a and 'b being the same), however this need not be the case. Advanced uses of syntax_fns can take types into consideration. For example, the constant bitstring$v2w with type bitstring->'a word is supported as follows:

```
> val tm = bitstringSyntax.mk_v2w (``l:bitstring``, ``:32``);
val tm = “v2w l”: term
```

```
> type_of tm;
val it = ":word32": hol_type
> bitstringSyntax.dest_v2w tm;
val it = ("1", ":32"): term * hol_type
```

This is implemented as follows:

```
val (v2w_tm, mk_v2w, dest_v2w, is_v2w) =
   HolKernel.syntax_fns
     {n = 1,
      dest = fn tm1 => fn e => fn w => (HolKernel.dest_monop tm1 e w, wordsSyntax.dim_of
      make = fn tm => fn (v, ty) => Term.mk_comb (Term.inst [Type.alpha |-> ty] tm, v)}
     "bitstring" "v2w"
```

T (boolSyntax)

```
boolSyntax.T : term
```

Constant denoting truth.

The ML variable `boolSyntax.T` is bound to the term `bool$T`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.negation`, `boolSyntax.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`

TAC_PROOF (Tactical)

```
Tactical.TAC_PROOF : goal * tactic -> thm
```

Attempts to prove a goal using a given tactic.

When applied to a goal-tactic pair `(A ?- t,tac)`, the `TAC_PROOF` function attempts to prove the goal `A ?- t`, using the tactic `tac`. If it succeeds, it returns the theorem `A' |- t` corresponding to the goal, where the assumption list `A'` may be a proper superset of `A` unless the tactic is valid; there is no inbuilt validity checking.

Failure

Fails unless the goal has hypotheses and conclusions all of type `bool`, and the tactic can solve the goal.

See also

`BasicProvers.PROVE`, `Tactical.prove`, `Tactical.VALID`

TACS_TO_LT (Tactical)

```
Tactical.TACS_TO_LT : tactic list -> list_tactic
```

The list-tactic which applies a list of tactics to the corresponding members of a list of goals.

If `T1,...,Tn` are tactics, `TACS_TO_LT [T1,...,Tn]` is a list-tactic which applies the tactics `T1,...,Tn` to the corresponding goals.

Failure

The application of `TACS_TO_LT` to a tactic list never fails. The resulting list-tactic fails if length of the goal list is not the same as that of the tactic list, or finally if `Ti` fails when applied to the `i`'th member of the goal list.

Applying different tactics to different subgoals.

Example

Where `tac1` is a tactic and `tacs2` is a list of tactics, `tac1 THEN_LT TACS_TO_LT tacs2` is equivalent to `tac1 THENL tacs2`

See also

`Tactical.THEN_LT`, `Tactical.THENL`

tactictoe (tacticToe)

```
tacticToe.tactictoe : term -> thm
```

A call to the rule `tacticToe.tactictoe` on a term `tm` is equivalent to a call to the tactic `tacticToe.ttt` on the goal `([],tm)`.

See also

`tacticToe.ttt`

tag (Tag)

```
Tag.type tag
```

Abstract type of oracle tags.

The type `tag` is used to track the use of oracles in HOL. An 'oracle' is a source of theorems that are not proved, but just asserted. In HOL, such unproven 'theorems' are used to incorporate the results of external proof tools. Each theorem coming from an oracle has a tag attached to it. This tag gets copied to any theorems hereditarily generated from an oracular theorem by inference.

See also

`Tag.read`, `Thm.mk_oracle_thm`

tag (Thm)

```
Thm.tag : thm -> tag
```

Extract the tag from a theorem.

An invocation `tag th`, where `th` has type `thm`, returns the tag of the theorem. If derivation of the theorem has appealed at some point to an oracle, the tag of that oracle will be embedded in the result. Otherwise, an empty tag is returned.

Failure

Never fails.

Example

```
> Thm.tag (mk_thm([],F));
val it = [oracles: #] [axioms: ]: tag

> Thm.tag NOT_FORALL_THM;
val it = [oracles: #] [axioms: ]: tag
```

See also

`Thm.mk_oracle_thm`, `Tag.read`, `Tag.merge`, `Tag.pp_tag`

TAUT_CONV (tautLib)

```
tautLib.TAUT_CONV : conv
```

Tautology checker. Proves instances of propositional formulae.

Given an instance `t` of a valid propositional formula, `TAUT_CONV` proves the theorem `|- t = T`. A propositional formula is a term containing only Boolean constants, Boolean-valued

variables, Boolean equalities, implications, conjunctions, disjunctions, negations and Boolean-valued conditionals. An instance of a formula is the formula with one or more of the variables replaced by terms of the same type. The conversion accepts terms with or without universal quantifiers for the variables.

Failure

Fails if the term is not an instance of a propositional formula or if the instance is not a valid formula.

Example

```
#TAUT_CONV
# ``!x n y. ((((n = 1) \/ ~x) ==> y) /\ (y ==> ~(n < 0)) /\ (n < 0)) ==> x``;
|- (!x n y. ((n = 1) \/ ~x ==> y) /\ (y ==> ~n < 0) /\ n < 0 ==> x) = T

#TAUT_CONV ``((((n = 1) \/ ~x) ==> y) /\ (y ==> ~(n < 0)) /\ (n < 0)) ==> x``;
|- ((n = 1) \/ ~x ==> y) /\ (y ==> ~n < 0) /\ n < 0 ==> x = T

#TAUT_CONV ``!n. (n < 0) \/ (n = 0)``;
Uncaught exception:
HOL_ERR
```

See also

`tautLib.TAUT_PROVE`, `tautLib.TAUT_TAC`, `tautLib.PTAUT_CONV`

TAUT_PROVE (tautLib)

```
tautLib.TAUT_PROVE : term -> thm
```

Tautology checker. Proves propositional formulae (and instances of them).

Given an instance of a valid propositional formula, `TAUT_PROVE` returns the instance of the formula as a theorem. A propositional formula is a term containing only Boolean constants, Boolean-valued variables, Boolean equalities, implications, conjunctions,

disjunctions, negations and Boolean-valued conditionals. An instance of a formula is the formula with one or more of the variables replaced by terms of the same type. The conversion accepts terms with or without universal quantifiers for the variables.

Failure

Fails if the term is not an instance of a propositional formula or if the instance is not a valid formula.

Example

```
#TAUT_PROVE
# ``!x n y. ((((n = 1) \/ ~x) ==> y) /\ (y ==> ~(n < 0)) /\ (n < 0)) ==> x``;
|- !x n y. ((n = 1) \/ ~x ==> y) /\ (y ==> ~n < 0) /\ n < 0 ==> x

#TAUT_PROVE ``((((n = 1) \/ ~x) ==> y) /\ (y ==> ~(n < 0)) /\ (n < 0)) ==> x``;
|- ((n = 1) \/ ~x ==> y) /\ (y ==> ~n < 0) /\ n < 0 ==> x

#TAUT_PROVE ``!n. (n < 0) \/ (n = 0)``;
Uncaught exception:
HOL_ERR
```

See also

`tautLib.TAUT_CONV`, `tautLib.TAUT_TAC`, `tautLib.PTAUT_PROVE`

TAUT_TAC (tautLib)

```
tautLib.TAUT_TAC : tactic
```

Tautology checker. Proves propositional goals (and instances of them).

Given a goal that is an instance of a propositional formula, this tactic will prove the goal provided it is valid. A propositional formula is a term containing only Boolean constants, Boolean-valued variables, Boolean equalities, implications, conjunctions, disjunctions, negations and Boolean-valued conditionals. An instance of a formula is the formula with

one or more of the variables replaced by terms of the same type. The tactic accepts goals with or without universal quantifiers for the variables.

Failure

Fails if the conclusion of the goal is not an instance of a propositional formula or if the instance is not a valid formula.

See also

`tautLib.TAUT_CONV`, `tautLib.TAUT_PROVE`, `tautLib.PTAUT_TAC`

tDefine (bossLib)

```
bossLib.tDefine : string -> term quotation -> tactic -> thm
```

General-purpose function definition facility.

`tDefine` is a definition package similar to `Define` except that it has a tactic parameter which is used to perform the termination proof for the specified function. `tDefine` accepts the same syntax used by `Define` for specifying functions.

If the specification is a simple abbreviation, or is primitive recursive (i.e., it exactly follows the recursion pattern of a previously declared HOL datatype) then the invocation of `tDefine` succeeds and stores the derived equations in the current theory segment. Otherwise, the function is not an instance of primitive recursion, and the termination prover may succeed or fail.

When processing the specification of a recursive function, `tDefine` must perform a termination proof. It automatically constructs termination conditions for the function, and invokes the supplied tactic in an attempt to prove the termination conditions. If that attempt fails, then `tDefine` fails.

If it succeeds, then `tDefine` stores the specified equations in the current theory segment, using the string argument as a stem for the name. An induction theorem customized for the defined function is also stored in the current segment. Note, however, that an induction theorem is not stored for primitive recursive functions, since that theorem would be identical to the induction theorem resulting from the declaration of the datatype.

If the tactic application fails, then tDefine fails.

Failure

tDefine fails if its input fails to parse and typecheck.

tDefine fails if it cannot prove the termination of the specified recursive function. In that case, one has to embark on the following multi-step process: (1) construct the function and synthesize its termination conditions with Hol_defn; (2) set up a goal to prove the termination conditions with tgoal; (3) interactively prove the termination conditions, usually by starting with an invocation of WF_REL_TAC; and (4) package everything up with an invocation of tDefine.

Example

The following attempt to invoke Define fails because the current default termination prover for Define is too weak:

```
Hol_datatype`foo = c1 | c2 | c3`;

Define `(f c1 x = x) /\
        (f c2 x = x + 3) /\
        (f c3 x = f c2 (x + 6))`;
```

The following invocation of tDefine uses the supplied tactic to prove termination.

```
tDefine "f"
   `(f c1 x = x) /\
    (f c2 x = x + 3) /\
    (f c3 x = f c2 (x + 6))`
 (WF_REL_TAC `measure (\p. case FST p of c3 -> 1 || _ -> 0)`);

Equations stored under "f_def".
Induction stored under "f_ind".
> val it = |- (f c1 x = x) /\ (f c2 x = x + 3) /\ (f c3 x = f c2 (x + 6)) : thm
```

Comments

tDefine automatically adds the definition it makes into the hidden ‘compset’ accessed by EVAL and EVAL_TAC.

See also

`bossLib.Define`, `bossLib.xDefine`, `TotalDefn.DefineSchema`, `bossLib.Hol_defn`, `Defn.tgoal`, `Defn.tprove`, `bossLib.WF_REL_TAC`, `bossLib.recInduct`, `bossLib.EVAL`, `bossLib.EVAL_TAC`

temp_set_grammars (Parse)

```
Parse.temp_set_grammars : type_grammar.grammar * term_grammar.grammar -> unit
```

Sets the global type and term grammars.

HOL uses two global grammars to control the parsing and printing of term and type values. These can be adjusted in a controlled way with functions such as `add_rule` and `overload_on`. By using just these standard functions, the system is able to export theories in such a way that changes to grammars persist from session to session.

Nonetheless it is occasionally useful to set grammar values directly. This change can't be made to persist, but will affect the current session.

Failure

Never fails.

See also

`Parse.current_grammars`, `Parse.add_rule`, `Parse.overload_on`, `Parse.parse_from_grammars`, `Parse.print_from_grammars`, `Parse.Term`

Term (Parse)

```
Parse.Term : term quotation -> term
```

Parses a quotation into a term value.

The parsing process for terms divides into four distinct phases.

The first phase converts the quotation argument into abstract syntax, a relatively simple parse tree datatype, with the following datatype definition (from `Absyn`):

```
datatype vstruct
    = VAQ    of term
    | VIDENT of string
    | VPAIR  of vstruct * vstruct
    | VTYPED of vstruct * pretype
datatype absyn
    = AQ    of term
    | IDENT of string
    | APP   of absyn * absyn
    | LAM   of vstruct * absyn
    | TYPED of absyn * pretype
```

This phase of parsing is concerned with the treatment of the rawest concrete syntax. It has no notion of whether or not a term corresponds to a constant or a variable, so all `preterm` leaves are ultimately either `IDENT`s or `AQ`s (anti-quotations).

This first phase converts infixes, mixfixes and all the other categories of syntactic rule from the global grammar into simple structures built up using `APP`. For example, `` `x op y` `` (where `op` is an infix) will turn into

```
APP(APP(IDENT "op", IDENT "x"), IDENT "y")
```

and `` `tok1 x tok2 y` `` (where `tok1 _ tok2` has been declared as a `Prefix` form for the term `f`) will turn into

```
APP(APP(IDENT "f", IDENT "x"), IDENT "y")
```

The special syntaxes for “let” and record expressions are also handled at this stage. For more details on how this is done see the reference entry for `Absyn`, which function can be used in isolation to see what is done at this phase.

The second phase of parsing consists of the resolution of names, identifying what were just `VAR`s as constants or genuine variables (whether free or bound). This phase also annotates all leaves of the data structure (given in the entry for `Preterm`) with type information.

The third phase of parsing works over the `Preterm` datatype and does type-checking, though ignoring overloaded values. The datatype being operated over uses reference variables to allow for efficiency, and the type-checking is done "in place". If type-checking is successful, the resulting value has consistent type annotations.

The final phase of parsing resolves overloaded constants. The type-checking done to this point may completely determine which choice of overloaded constant is appropriate, but if not, the choice may still be completely determined by the interaction of the possible types for the overloaded possibilities.

Finally, depending on the value of the global flags `guessing_tyvars` and `guessing_overloads`, the parser will make choices about how to resolve any remaining ambiguities.

The parsing process is entirely driven by the global grammar. This value can be inspected with the `term_grammar` function.

Failure

All over place, and for all sorts of reasons.

Turns strings into terms.

See also

`Parse.Absyn`, `Parse.overload_on`, `Parse.term_grammar`

term (Term)

```
Term.eqtype term
```

ML datatype of HOL terms.

The ML abstract type `term` represents the set of HOL terms, which is essentially the simply typed lambda calculus of Church. A term may be a variable, a constant, an application of one term to another, or a lambda abstraction.

Comments

Since `term` is an ML eqtype, any two `terms` `tm1` and `tm2` can be tested for equality by `tm1 = tm2`. However, the fundamental notion of equality for terms is implemented by `aconv`.

Since term is an abstract type, access to its representation is mediated by the interface presented by the `Term` structure.

See also

`Type.hol_type`

term_grammar (Parse)

```
Parse.term_grammar : unit -> term_grammar.grammar
```

Returns the current global term grammar.

Failure

Never fails.

Comments

There is a pretty-printer installed in the interactive system so that term grammar values are presented nicely. The global term grammar is passed as a parameter to the `Term` parsing function in the `Parse` structure, and also drives the installed term and theorem pretty-printers.

See also

`Parse.parse_from_grammars`, `Parse.print_from_grammars`, `Parse.temp_set_grammars`, `Parse.Term`

term_to_string (Parse)

```
Parse.term_to_string : term -> string
```

Converts a term to a string.

Uses the global term grammar and pretty-printing flags to turn a term into a string. It assumes that the string should be broken up as if for display on a screen that is as wide as the value stored in the `Globals.linewidth` variable.

Failure

Should never fail.

See also

`Parse.print_term`

tex_theory (EmitTeX)

```
EmitTeX.tex_theory : string -> unit
```

Emits theory as LaTeX commands and creates a document template.

An invocation of `tex_theory thy` will export the named theory `thy` as a collection of LaTeX commands and it will also generate a document "thy.tex" that presents the theory. The string `"-"` may be used to denote the current theory segment. The theory is exported with `print_theory_as_tex`.

Failure

Will fail if there is a system error when trying to write the files. It will not overwite the file `name`, however, "HOLname.tex" may be overwritten.

Example

The invocation

```
- EmitTeX.tex_theory "list";
> val it = () : unit
```

will generate two files "HOLlist.tex" and "list.tex".

See also

EmitTeX.print_term_as_tex, EmitTeX.print_type_as_tex, EmitTeX.print_theorem_as_tex, EmitTeX.print_theory_as_tex, EmitTeX.print_theories_as_tex_doc

tgoal (Defn)

```
Defn.tgoal : defn -> proofs
```

Set up a termination proof.

`tgoal defn` sets up a termination proof for the function represented by `defn`. It creates a new goalstack and makes it the focus of subsequent goalstack operations.

Failure

`tgoal defn` fails if `defn` represents a non-recursive or primitive recursive function.

Example

```
> val qsort_defn =
   Hol_defn "qsort"
      `(qsort ___ [] = []) /\
       (qsort ord (x::rst) =
           APPEND (qsort ord (FILTER ($~ o ord x) rst))
             (x :: qsort ord (FILTER (ord x) rst)))`;
<<HOL message: inventing new type variable names: 'a>>
val qsort_defn =
   HOL function definition (recursive)

   Equation(s) :
    [...] ⊢ qsort v0 [] = []
    [...]
   ⊢ qsort ord (x::rst) =
     qsort ord (FILTER ($¬ o ord x) rst) ++ x::qsort ord (FILTER (ord x) rst)
```

```
   Induction :
    [...]
   ⊢ ∀P. (∀v0. P v0 []) ∧
         (∀ord x rst.
            P ord (FILTER (ord x) rst) ∧ P ord (FILTER ($¬ ∘ ord x) rst) ⇒
            P ord (x::rst)) ⇒
         ∀v v1. P v v1

   Termination conditions :
     0. ∀rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)
     1. ∀rst x ord. R (ord,FILTER ($¬ ∘ ord x) rst) (ord,x::rst)
     2. WF R: DefnBase.defn

> Defn.tgoal qsort_defn;
val it =
   Proof manager status: 1 proof.
   1. Incomplete goalstack:
        Initial goal:
        ∃R. WF R ∧ (∀rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)) ∧
            ∀rst x ord. R (ord,FILTER ($¬ ∘ ord x) rst) (ord,x::rst)
```

See also

`TotalDefn.WF_REL_TAC`, `Defn.tprove`, `Defn.Hol_defn`

THEN (Tactical)

```
op Tactical.THEN : tactic * tactic -> tactic}
op THEN : list_tactic * tactic -> list_tactic
```

Applies a tactic to all subgoals produced by a tactic or list-tactic.

If `T1` and `T2` are tactics, `T1 THEN T2` is a tactic which applies `T1` to a goal, then applies the tactic `T2` to all the subgoals generated. If `T1` solves the goal then `T2` is never applied.

Alternatively, `T1` may be a list-tactic which is applied to an initial list of goals.

Failure

The application of THEN to a pair of tactics never fails. The resulting tactic fails if T1 fails when applied to the goal, or if T2 does when applied to any of the resulting subgoals.

Comments

Although normally used to sequence tactics which generate a single subgoal, it is worth remembering that it is sometimes useful to apply the same tactic to multiple subgoals; sequences like the following:

```
EQ_TAC THENL [ASM_REWRITE_TAC[], ASM_REWRITE_TAC[]]
```

can be replaced by the briefer:

```
EQ_TAC THEN ASM_REWRITE_TAC[]
```

See also

Tactical.EVERY, Tactical.ORELSE, Tactical.THENL, Tactical.THEN_LT

THEN1 (Tactical)

```
op Tactical.THEN1 : tactic * tactic -> tactic
```

A tactical like THEN that applies the second tactic only to the first subgoal.

If T1 and T2 are tactics, T1 THEN1 T2 is a tactic which applies T1 to a goal, then applies the tactic T2 to the first subgoal generated. T1 must produce at least one subgoal, and T2 must completely solve the first subgoal of T1.

Failure

The application of THEN1 to a pair of tactics never fails. The resulting tactic fails if T1 fails when applied to the goal, if T1 does not produce at least one subgoal (i.e., T1 completely solves the goal), or if T2 does not completely solve the first subgoal generated by T1.

Comments

THEN1 can be applied to make the proof more linear, avoiding unnecessary THENLs. It is especially useful when used with REVERSE.

Example

For example, given the goal

```
simple_goal /\ complicated_goal
```

the tactic

```
(CONJ_TAC THEN1 T0)
THEN T1
THEN T2
THEN ...
THEN Tn
```

avoids the extra indentation of

```
CONJ_TAC THENL
[T0,
 T1
 THEN T2
 THEN ...
 THEN Tn]
```

See also

Tactical.EVERY, Tactical.ORELSE, Tactical.REVERSE, Tactical.THEN, Tactical.THENL

THEN_CONSEQ_CONV (ConseqConv)

```
ConseqConv.THEN_CONSEQ_CONV : (conseq_conv -> conseq_conv -> conseq_conv)
```

Applies two consequence conversions in sequence.

`THEN_CONSEQ_CONV cc1 cc2` corresponds to `c1 THENC c2` for classical conversions. Thus, if `cc1` returns `|- t' ==> t` when applied to `t`, and `cc2` returns `|- t'' ==> t'` when applied to `t'`, then `(THEN_CONSEQ_CONV cc1 cc2) t` returns `|- t'' ==> t`. `THEN_CONSEQ_CONV` can handle weakening as well: If `cc1` returns `|- t ==> t'` when applied to `t`, and `cc2` returns `|- t' ==> t''` when applied to `t'`, then `(THEN_CONSEQ_CONV cc1 cc2) t` returns `|- t ==> t''`. Finally, if `cc1` returns `|- t = t'` when applied to `t`, and `cc2` returns `|- t' = t''` when applied to `t'`, then `(THEN_CONSEQ_CONV cc1 cc2) t` returns `|- t = t''`. If one of the conversions returns an equation, while the other returns an implication, the needed implication is automatically deduced.

See also

`Conv.THENC`, `ConseqConv.EVERY_CONSEQ_CONV`

THEN_LT (Tactical)

```
op Tactical.THEN_LT : tactic * list_tactic -> tactic
op THEN_LT : list_tactic * list_tactic -> list_tactic
```

Applies a list-tactic to the corresponding subgoals generated by a tactic or by a previous list-tactic.

If `tac` is a tactic and `ltac` is a list-tactic, then `tac THEN_LT ltac` is a tactic which applies `tac` to a goal, and if it does not fail, applies the list-tactic `ltac` to the resulting list of subgoals.

If `ltac1` and `ltac2` are list-tactics, then `ltac1 THEN_LT ltac2` is a list-tactic which applies `ltac1` to a goal list, and if it does not fail, applies `ltac2` to the resulting list of goals.

Failure

The application of `THEN_LT` to a tactic or list-tactic and a list-tactic never fails.

The tactic `tac THEN_LT ltac` fails if `tac` fails when applied to the goal, or if `ltac` fails when applied to the resulting subgoal list.

The list-tactic `ltac1 THEN_LT ltac2` fails if `ltac1` fails when applied to the goal list, or if `ltac2` fails when applied to the goal list result of `ltac1`.

Applying a combination of tactics to a list of subgoals, or otherwise manipulating a list of subgoals.

Example

Where `tac1` and `tac2` are tactics, `tac1 THEN_LT ALLGOALS tac2` is equivalent to `tac1 THEN tac2`

Where `tac1` is a tactic and `tacs2` is a list of tactics, `tac1 THEN_LT NULL_OK_LT (TACS_TO_LT tacs2)` is equivalent to `tac1 THENL tacs2`

Where `tac` is a tactic, `tac THEN_LT REVERSE_LT` is equivalent to `REVERSE tac`

See also

`Tactical.ALLGOALS`, `Tactical.THEN`, `Tactical.TACS_TO_LT`, `Tactical.THENL`, `Tactical.NULL_OK_LT`, `Tactical.NTH_GOAL`, `Tactical.REVERSE_LT`, `Tactical.REVERSE`

THEN_TCL (Thm_cont)

```
Thm_cont.THEN_TCL : (thm_tactical * thm_tactical -> thm_tactical)
```

Composes two theorem-tacticals.

If `ttl1` and `ttl2` are two theorem-tacticals, `ttl1 THEN_TCL ttl2` is a theorem-tactical which composes their effect; that is, if:

```
ttl1 ttac th1 = ttac th2
```

and

```
ttl2 ttac th2 = ttac th3
```

then

```
(ttl1 THEN_TCL ttl2) ttac th1 = ttac th3
```

Failure

The application of THEN_TCL to a pair of theorem-tacticals never fails.

See also

Thm_cont.EVERY_TCL, Thm_cont.FIRST_TCL, Thm_cont.ORELSE_TCL

THENC (Conv)

```
op Conv.THENC : (conv -> conv -> conv)
```

Applies two conversions in sequence.

If the conversion c1 returns |- t = t' when applied to a term ``t``, and c2 returns |- t' = t'' when applied to ``t'``, then the composite conversion (c1 THENC c2) ``t`` returns |- t = t''. That is, (c1 THENC c2) ``t`` has the effect of transforming the term ``t`` first with the conversion c1 and then with the conversion c2.

THENC also handles the possibility that either of its arguments might return the UNCHANGED exception. If the first conversion returns UNCHANGED when applied to its argument, THENC just returns the result of the second conversion applied to the same initial term. If the second conversion raises UNCHANGED (and the first did not), then the result will be the theorem returned by the first conversion. In this way, unnecessary calls to TRANS can be avoided.

Failure

(This refers to failure other than by raising UNCHANGED). (c1 THENC c2) ``t`` fails if either the conversion c1 fails when applied to ``t``, or if c1 ``t`` succeeds and returns |- t = t' but c2 fails when applied to ``t'``. (c1 THENC c2) ``t`` may also fail if either of c1 or c2 is not, in fact, a conversion (i.e. a function that maps a term t to a theorem |- t = t').

See also

Conv.UNCHANGED, Conv.EVERY_CONV

THENL (Tactical)

```
op Tactical.THENL : tactic * tactic list -> tactic
op THENL : list_tactic * tactic list -> list_tactic
```

Applies a list of tactics to the corresponding subgoals generated by a tactic or a list-tactic.

If `T` is a tactic or list-tactic and `T1`, . . . , `Tn` are tactics, `T THENL [T1, . . . ,Tn]` is a tactic or list-tactic which applies `T` to a goal or goal list, and if it does not fail, applies the tactics `T1`, . . . , `Tn` to the corresponding subgoals, unless `T` completely solves the goal(s).

Failure

The application of `THENL` to a (list-)tactic and tactic list never fails. The resulting tactic fails if `T` fails when applied to the goal(s), or if the goal list is not empty and its length is not the same as that of the tactic list, or finally if `Ti` fails when applied to the `i`'th subgoal generated by `T`.

Applying different tactics to different subgoals.

See also

`Tactical.EVERY`, `Tactical.ORELSE`, `Tactical.THEN`, `Tactical.THEN_LT`

theorems (DB)

```
DB.theorems : string -> (string * thm) list
```

All the theorems stored in the named theory.

An invocation `theorems thy`, where `thy` is the name of a currently loaded theory segment, will return a list of the theorems stored in that theory. Axioms and definitions are excluded. Each theorem is paired with its name in the result. The string `"-"` may be used to denote the current theory segment.

Failure

Never fails. If thy is not the name of a currently loaded theory segment, the empty list is returned.

Example

```
> theorems "combin";
val it =
   [("W_THM", ⊢ ∀f x. W f x = f x x),
    ("UPDATE_EQ", ⊢ ∀f a b c. f⦇a ↦ c; a ↦ b⦈ = f⦇a ↦ c⦈),
    ("UPDATE_COMMUTES",
     ⊢ ∀f a b c d. a ≠ b ⇒ f⦇a ↦ c; b ↦ d⦈ = f⦇b ↦ d; a ↦ c⦈),
    ("UPDATE_APPLY_IMP_ID", ⊢ ∀f b a. f a = b ⇒ f⦇a ↦ b⦈ = f),
    ("UPDATE_APPLY_ID_RWT",
     ⊢ (∀f a b. f⦇a ↦ b⦈ = f ⇔ f a = b) ∧ ∀f a b. f = f⦇a ↦ b⦈ ⇔ f a = b),
    ("UPDATE_APPLY_ID", ⊢ ∀f a b. f a = b ⇔ f⦇a ↦ b⦈ = f),
    ("UPDATE_APPLY1", ⊢ ∀a x f. f⦇a ↦ x⦈ a = x),
    ("UPDATE_APPLY",
     ⊢ (∀a x f. f⦇a ↦ x⦈ a = x) ∧ ∀a b x f. a ≠ b ⇒ f⦇a ↦ x⦈ b = f b),
    ("UPD_SAME_KEY_UNWIND",
     ⊢ ∀f1 f2 a b c.
         f1⦇a ↦ b⦈ = f2⦇a ↦ c⦈ ⇒ b = c ∧ ∀v. f1⦇a ↦ v⦈ = f2⦇a ↦ v⦈),
[...Output elided...]
```

See also

DB.thy, DB.fetch, DB.thms, DB.definitions, DB.axioms, DB.listDB

thm (Thm)

```
Thm.type thm
```

Type of theorems of the HOL logic.

The abstract type thm represents the theorems derivable by inference in the HOL logic. The type of theorems can be viewed as the inductive closure of the axioms of the HOL

logic by the primitive inference rules of HOL. Robin Milner had the brilliant insight to implement this view by encapsulating the primitive rules of inference for a logic as the constructors for an abstract type of theorems. This implementation technique is adopted in HOL.

See also

`Thm.dest_thm`, `Thm.hyp`, `Thm.concl`, `Thm.tag`, `Thm.ASSUME`, `Thm.REFL`, `Thm.BETA_CONV`, `Thm.ABS`, `Thm.DISCH`, `Thm.MP`, `Thm.SUBST`, `Thm.INST_TYPE`

thm_count (Count)

```
Count.thm_count :
     unit ->
     {ASSUME : int, REFL : int, BETA_CONV : int, SUBST : int,
      ABS : int, DISCH : int, MP : int, INST_TYPE : int,
      MK_COMB : int, AP_TERM : int, AP_THM : int, ALPHA : int,
      ETA_CONV : int, SYM : int, TRANS : int, EQ_MP : int,
      EQ_IMP_RULE : int, INST : int, SPEC : int, GEN : int,
      EXISTS : int, CHOOSE : int, CONJ : int, CONJUNCT1 : int,
      CONJUNCT2 : int, DISJ1 : int, DISJ2 : int, DISJ_CASES : int,
      NOT_INTRO : int, NOT_ELIM : int, CCONTR : int, GEN_ABS : int,
      definition : int, axiom : int, from_disk : int, oracle :int,
      total :int }
```

Returns the current value of the theorem counter.

If enabled, HOL maintains a counter which is incremented every time a primitive inference is performed (or an axiom or definition set up). A call to `thm_count()` returns the current value of this counter. Inference counting needs to be enabled with the call `Count.counting_thms true`. Counting can be turned off by calling `counting_thms false`.

The default is for inference counting not to be enabled.

Failure

Never fails.

See also

Count.apply

thms (DB)

```
DB.thms : string -> (string * thm) list
```

All the theorems, definitions, and axioms stored in the named theory.

An invocation `thms thy`, where `thy` is the name of a currently loaded theory segment, will return a list of the theorems, definitions, and axioms stored in that theory. Each theorem is paired with its name in the result. The string `"-"` may be used to denote the current theory segment.

Failure

Never fails. If `thy` is not the name of a currently loaded theory segment, the empty list is returned.

Example

```
> thms "combin";
val it =
   [("W_THM", ⊢ ∀f x. W f x = f x x), ("W_DEF", ⊢ W = (λf x. f x x)),
    ("UPDATE_EQ", ⊢ ∀f a b c. f⦇a ↦ c; a ↦ b⦈ = f⦇a ↦ c⦈),
    ("UPDATE_def", ⊢ ∀a b. (a =+ b) = (λf c. if a = c then b else f c)),
    ("UPDATE_COMMUTES",
     ⊢ ∀f a b c d. a ≠ b ⇒ f⦇a ↦ c; b ↦ d⦈ = f⦇b ↦ d; a ↦ c⦈),
    ("UPDATE_APPLY_IMP_ID", ⊢ ∀f b a. f a = b ⇒ f⦇a ↦ b⦈ = f),
    ("UPDATE_APPLY_ID_RWT",
     ⊢ (∀f a b. f⦇a ↦ b⦈ = f ⇔ f a = b) ∧ ∀f a b. f = f⦇a ↦ b⦈ ⇔ f a = b),
    ("UPDATE_APPLY_ID", ⊢ ∀f a b. f a = b ⇔ f⦇a ↦ b⦈ = f),
    ("UPDATE_APPLY1", ⊢ ∀a x f. f⦇a ↦ x⦈ a = x),
    ("UPDATE_APPLY",
     ⊢ (∀a x f. f⦇a ↦ x⦈ a = x) ∧ ∀a b x f. a ≠ b ⇒ f⦇a ↦ x⦈ b = f b),
    ("UPD_SAME_KEY_UNWIND",
```

```
     ⊢ ∀f1 f2 a b c.
[...Output elided...]
```

See also

DB.thy, DB.theorems, DB.axioms, DB.definitions, DB.fetch, DB.listDB

thy (DB)

```
DB.thy : string -> data list
```

Return the contents of a theory.

An invocation `DB.thy s` returns the contents of the specified theory segment `s` in a list of `(thy,name),(thm,class)` tuples. In a tuple, `(thy,name)` designate the theory and the name given to the object in the theory. The `thm` element is the named object, and `class` its classification (one of `Thm` (theorem), `Axm` (axiom), or `Def` (definition)).

Case distinctions are ignored when determining the segment. The current segment may be specified, either by the distinguished literal `"-"`, or by the name given when creating the segment with `new_theory`.

Failure

Never fails, but will return an empty list when `s` does not designate a currently loaded theory segment.

Example

```
> List.take (DB.thy "pair", 3);
val it =
   [(("pair", "WF_RPROD"),
     (⊢ ∀R Q. WF R ∧ WF Q ⇒ WF (R ### Q),
      {class = Thm, loc =
       Located
        {exact = true, linenum = 851, scriptpath =
         "$(HOLDIR)/src/coretypes/pairScript.sml"}, private = false})),
```

```
  (("pair", "WF_LEX"),
   (⊢ ∀R Q. WF R ∧ WF Q ⇒ WF (R LEX Q),
    {class = Thm, loc =
     Located
      {exact = true, linenum = 808, scriptpath =
       "$(HOLDIR)/src/coretypes/pairScript.sml"}, private = false})),
  (("pair", "UNCURRY_VAR"),
   (⊢ ∀f v. UNCURRY f v = f (FST v) (SND v),
    {class = Thm, loc =
     Located
      {exact = true, linenum = 216, scriptpath =
       "$(HOLDIR)/src/coretypes/pairScript.sml"}, private = false}))]:
data list
```

See also

DB.class, DB.data, DB.listDB, DB.theorems, DB.match, Theory.new_theory

thy_addon (Theory)

Theory.type thy_addon

Type of theory additions.

The type abbreviation thy_addon, declared as

```
type thy_addon = {sig_ps : unit PP.pprinter option,
                  struct_ps : unit PP.pprinter option}
```

packages up the arguments to adjoin_to_theory. The sig_ps argument is an optional prettyprinter, which will be invoked when the theory signature file is written. The struct_ps argument is an optional prettyprinter invoked when the theory structure file is written.

The unit type parameter in both cases simply means that the pretty-printers won't be passed any extra information when invoked.

See also

`Theory.adjoin_to_theory`

thy_ssfrag (BasicProvers)

```
BasicProvers.thy_ssfrag : string -> simpLib.ssfrag
```

Returns simplifier fragment for a theory

Returns the simpset fragment recorded for the given theory. This consists of the rewrites passed to `export_rewrites`.

Failure

Fails if the theory was not found, or did not export any theorems.

See also

`BasicProvers.export_rewrites`

thytype_abbrev (Parse)

```
Parse.thytype_abbrev : {Name:string,Thy:string} * hol_type * bool -> unit
```

Abbreviates a type to a specific theory-qualified name.

A call to `thytype_abbrev({Name=n,Thy=t}, ty, prp)` establishes the "kernel" name `t$n` as an abbreviation for the type `ty`, as happens with `type_abbrev`. The boolean flag `prp` indicates whether or not this abbreviation will also be used when the printer comes to print the given type. In other words, after the call it becomes possible to write "`:args t$n`" to stand for type `ty`. If there are type variables in `ty` they become the parameters to the new abbreviated type operator. These parameters need to be filled in in the `args`

position above. If there are no type variables in `ty` then abbreviation is of a whole type, and `args` must be blank.

If there was an existing abbreviation for `t$n`, then this will be replaced by the call.

In addition, after the given call, this abbreviation will become the preferred binding for the bare name `n`. Other abbreviations in different theories will need to use the form with fully-qualified names (`thy1$n`, `thy2$n` etc).

If the boolean flag is false, this invocation is comparable to the behaviour after `intputonly_type_abbrev`: the abbreviation can be used to input types of the desired pattern, but such types will print as they did previously.

Failure

Fails if `ty` is a variable type.

Comments

As with other parsing and pretty-printing functions, there is a companion function, `temp_thytype_abbrev`, which has the same effect on the global grammar but does not cause the change to persist when the theory is exported.

It is legitimate to use a string for the theory component of the record that does not correspond to the current theory. Indeed, it is perfectly reasonable to do this, if one wants to give priority to a particular ancestral abbreviation.

See also

`Parse.type_abbrev`

time (Lib)

```
Lib.time : ('a -> 'b) -> 'a -> 'b
```

Measure how long a function application takes.

An application `time f x` starts a clock, applies `f` to `x`, and then checks the clock to see how long that took. It prints out the elapsed runtime, garbage collection time, and system time before returning the value of `f x`.

Failure

If `f x` raises `e`, then `time f x` raises `e`, but still reports elapsed time.

Example

```
> time (int_sort) (for 0 999 I);
runtime: 0.00299s,    gctime: 0.00000s,     systime: 0.00000s.
val it =
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
    21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38,
    39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56,
    57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
    75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92,
    93, 94, 95, 96, 97, 98, 99, ...]: int list

> fun f x = f (x + 1);
val f = fn: int -> 'a

> time f 0; (* would need interrupting *)
```

See also

`Lib.end_time`, `Lib.start_time`, `Count.thm_count`

timeout (smlTimeout)

```
smlTimeout.timeout : real -> ('a -> 'b) -> 'a -> 'b
```

Interrupts a function `f` after a time limit.

In case the limit is reached, the exception FunctionTimeout is raised, otherwise it behaves in the same way as a call to “f” on its argument.

Failure

Fails when the call to `f` exceeds the time limit or if `f` fails.

Example

```
> load "smlTimeout"; open smlTimeout;
val it = (): unit

> timeout 0.1 (fn (x:int) => (raise Match):int) 5;
Exception- Match raised

> timeout 0.2 OS.Process.sleep (Time.fromReal 2.0);
Exception- FunctionTimeout raised

> timeout 1.0 (fn x => x) 5;
val it = 5: int
```

Comments

Relies on a conditional variable to decide if and when to send an Interrupt signal to the worker thread in which the function `f` is called. In the case an interrupt is needed, a bit of time is given to the function `f` to catch the Interrupt and return a result. This last step has been implemented with a busy waiting loop that has experimentally been determined to be more efficient than relying on an additional condition variable.

tmCases_on (bossLib)

```
bossLib.tmCases_on : term -> string list -> tactic
```

Begins a “cases” proof on the provided term

A call to `tmCases_on t names` will do the equivalent of a `FULL_STRUCT_CASES_TAC` on the term `t`, using the cases (or “nchotomy”) theorem stored in the `TypeBase` for `t`’s type. If the `names` list is not empty, the names encoded there will be used to give names to any existentially quantified names in the cases theorem. Each element of the `names` list corresponds to the cases of the theorem, and, as constructors may take multiple arguments, each corresponding to an existentially quantified variable, the element is itself a list of names, separated by spaces. For example, the cases theorem for lists could be passed a string list of the form `["", "head tail"]`. If the `names` is empty, then the system will choose names for the existentially quantified variables.

As a convenience, if the term argument is a variable, and there are variables of that name free in the goal, or bound by top-level universal quantifiers in the goal's conclusion, then the type of the variable is ignored and its name is used to generate the argument to the tactic. If a goal has multiple variables of the same name (always a bad idea!) the choice of variable is unspecified.

Failure

Fails if the term is not of a type occurring in the `TypeBase`.

Example

Note how in this example, the parser will give the argument `l` bare type "$:\alpha$", but it still picks the appropriately instantiated list cases theorem for the `l` that appears in the goal, which may have type "`:num list`", for example.

```
          ?- MAP f l = []
======================================   tmCases_on “l” ["", "e es"]
?- MAP f [] = []    ?- MAP f (e::es) = []
```

See also

`bossLib.Cases_on`, `Tactic.FULL_STRUCT_CASES_TAC`

TOP_DEPTH_CONV (Conv)

```
Conv.TOP_DEPTH_CONV : (conv -> conv)
```

Applies a conversion top-down to all subterms, retraversing changed ones.

`TOP_DEPTH_CONV c tm` repeatedly applies the conversion `c` to all the subterms of the term `tm`, including the term `tm` itself. The supplied conversion `c` is applied to the subterms of `tm` in top-down order and is applied repeatedly (zero or more times, as is done by `REPEATC`) at each subterm until it fails. If a subterm `t` is changed (up to alpha-equivalence) by virtue of the application of `c` to its own subterms, then the term into which `t` is transformed is retraversed by applying `TOP_DEPTH_CONV c` to it.

Failure

`TOP_DEPTH_CONV c tm` never fails but can diverge.

Comments

The implementation of this function uses failure to avoid rebuilding unchanged subterms. That is to say, during execution the exception `QConv.UNCHANGED` may be generated and later trapped. The behaviour of the function is dependent on this use of failure. So, if the conversion given as an argument happens to generate the same exception, the operation of `TOP_DEPTH_CONV` will be unpredictable.

See also

`Conv.DEPTH_CONV`, `Conv.ONCE_DEPTH_CONV`, `Conv.REDEPTH_CONV`

top_function_of (Feedback)

```
top_function_of : hol_error -> string
```

Extract last function name added to a `hol_error`.

An invocation `top_function_of holerr` returns the function name of the top `origin` element held in `holerr`.

Example

```
> top_function_of
    (mk_hol_error "Foo" "bar" locn.Loc_Unknown "bad input")
val it = "bar": string
```

Failure

The call fails if the `origin` stack of the `hol_error` is empty.

See also

`Feedback`, `Feedback.mk_hol_error`, `Feedback.origins_of`, `Feedback.message_of`, `Feedback.top_structure_of`, `Feedback.top_location_of`, `Feedback.set_top_function`

top_goal (proofManagerLib)

```
proofManagerLib.top_goal : unit -> goal
```

Returns the current goal of the subgoal package.

The function `top_goal` is part of the subgoal package. It returns the top goal of the goal stack in the current proof state. For a description of the subgoal package, see `set_goal`.

Failure

A call to `top_goal` will fail if there are no unproven goals. This could be because no goal has been set using `set_goal` or because the last goal set has been completely proved.

Examining the proof state after a proof fails.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

top_location_of (Feedback)

```
top_location_of : hol_error -> locn.locn
```

Extract last location added to a `hol_error`.

An invocation `top_location_of holerr` returns the location of the top `origin` element held in `holerr`.

Example

```
> val exloc = let open locn in Loc (LocA(0,0), LocA(3,42)) end;
val exloc = 1:0-4:42: locn.locn

> top_location_of (mk_hol_error "Foo" "bar" exloc "bad input");
val it = 1:0-4:42: locn.locn
```

Failure

The call fails if the `origin` stack of the `hol_error` is empty.

See also

`Feedback`, `Feedback.mk_hol_error`, `Feedback.origins_of`, `Feedback.message_of`, `Feedback.top_structure_of`, `Feedback.top_function_of`

top_structure_of (Feedback)

```
top_structure_of : hol_error -> string
```

Extract last structure name added to a `hol_error`.

An invocation `top_structure_of holerr` returns the structure name of the top `origin` element held in `holerr`.

Example

```
> top_structure_of
    (mk_hol_error "Foo" "bar" locn.Loc_Unknown "bad input")
val it = "Foo": string
```

Failure

The call fails if the `origin` stack of the `hol_error` is empty.

See also

`Feedback`, `Feedback.mk_hol_error`, `Feedback.origins_of`, `Feedback.message_of`, `Feedback.top_function_of`, `Feedback.top_location_of`, `Feedback.set_top_structure`

top_thm (proofManagerLib)

```
proofManagerLib.top_thm : unit -> thm
```

Returns the theorem just proved using the subgoal package.

The function `top_thm` is part of the subgoal package. A proof state of the package consists of either goal and justification stacks if a proof is in progress or a theorem if a proof has just been completed. If the proof state consists of a theorem, `top_thm` returns that theorem. For a description of the subgoal package, see `set_goal`.

Failure

`top_thm` will fail if the proof state does not hold a theorem. This will be so either because no goal has been set or because a proof is in progress with unproven subgoals.

Accessing the result of an interactive proof session with the subgoal package.

See also

`proofManagerLib.set_goal`, `proofManagerLib.restart`, `proofManagerLib.backup`, `proofManagerLib.redo`, `proofManagerLib.restore`, `proofManagerLib.save`, `proofManagerLib.set_backup`, `proofManagerLib.expand`, `proofManagerLib.expandf`, `proofManagerLib.p`, `proofManagerLib.top_thm`, `proofManagerLib.top_goal`

topsort (Lib)

```
Lib.topsort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Topologically sorts a list using a given partial order relation.

The call `topsort opr list` where `opr` is a curried partial order on the elements of `list`, will topologically sort the list, i.e., will permute it such that if `x opr y` then `x` will occur to the left of `y` in the resulting list.

Failure

If `opr` fails when applied to `x` and `y` in `list`. Also, `topsort` will fail if there is a chain of elements `x1`,...,`xn`, all in `list`, such that `opr x_1 x_2`, ..., `opr xn x1`. This displays a cyclic dependency.

Example

The following call arranges a list of terms in subterm order:

```
- fun is_subterm x y = Lib.can (find_term (aconv x)) y;
> val is_subterm = fn : term -> term -> bool

- topsort is_subterm
  [``x+1``, ``x:num``, ``y + (x + 1)``, ``y + x``, ``y + x + z``, ``y:num``];
> val it = [``y``, ``x``, ``x + 1``, ``y + x``, ``y + x + z``, ``y + (x + 1)``]
```

See also

`Lib.sort`

total (Lib)

```
Lib.total : ('a -> 'b) -> 'a -> 'b option
```

Converts a partial function to a total function.

In ML, there are two main ways for a function to signal that it has been called on an element outside of its intended domain of application: exceptions and options. The function `total` maps a function that may raise an exception to one that returns an element in the option type. Thus, if `f x` results in any exception other than `Interrupt`

being raised, then `total f x` returns `NONE`. If `f x` raises `Interrupt`, then `total f x` likewise raises `Interrupt`. If `f x` returns `y`, then `total f x` returns `SOME y`.

The function `total` has an inverse `partial`. Generally speaking, `(partial err o total) f` equals `f`, provided that `err` is the only exception that `f` raises. Similarly, `(total o partial err) f` is equal to `f`.

Failure

When application of the first argument to the second argument raises `Interrupt`.

Example

```
> 3 div 0;
Exception- Div raised

> total (op div) (3,0);
val it = NONE: int option

> (partial Div o total) (op div) (3,0);
Exception- Div raised
```

See also

`Lib.partial`

tprove (Defn)

```
Defn.tprove : defn * tactic -> thm * thm
```

Prove termination of a `defn`.

`tprove` takes a `defn` and a `tactic`, and uses the tactic to prove the termination constraints of the `defn`. A pair of theorems `(eqns,ind)` is returned: `eqns` is the unconstrained recursion equations of the `defn`, and `ind` is the corresponding induction theorem for the equations, also unconstrained.

`tprove` and `tgoal` can be seen as analogues of `prove` and `set_goal` in the specialized domain of proving termination of recursive functions.

It is up to the user to store the results of `tprove` in the current theory segment.

Failure

`tprove (defn,tac)` fails if `tac` fails to prove the termination conditions of `defn`.

`tprove (defn,tac)` fails if `defn` represents a non-recursive or primitive recursive function.

Example

Suppose that we have defined a version of Quicksort as follows:

```
- val qsort_defn =
    Hol_defn "qsort"
       `(qsort ___ [] = []) /\
        (qsort ord (x::rst) =
            APPEND (qsort ord (FILTER ($~ o ord x) rst))
              (x :: qsort ord (FILTER (ord x) rst)))`
```

Also suppose that a tactic `tac` proves termination of `qsort`. (This tactic has probably been built by interactive proof after starting a goalstack with `tgoal qsort_defn`.) Then

```
- val (qsort_eqns, qsort_ind) = tprove(qsort_defn, tac);

> val qsort_eqns =
    |- (qsort v0 [] = []) /\
       (qsort ord (x::rst) =
          APPEND (qsort ord (FILTER ($~ o ord x) rst))
              (x::qsort ord (FILTER (ord x) rst))) : thm

  val qsort_ind =
    |- !P.
         (!v0. P v0 []) /\
         (!ord x rst.
            P ord (FILTER ($~ o ord x) rst) /\
            P ord (FILTER (ord x) rst) ==> P ord (x::rst))
         ==>
        !v v1. P v v1 : thm
```

Comments

The recursion equations returned by a successful invocation of `tprove` are automatically added to the global `compset` accessed by `EVAL`.

See also

`Defn.tgoal`, `Defn.Hol_defn`, `bossLib.EVAL`

trace (Feedback)

```
Feedback.trace : string * int -> ('a -> 'b) -> 'a -> 'b
```

Invoke a function with a specified level of tracing.

The `trace` function is used to set the value of a tracing variable for the duration of one top-level function call.

A call to `trace (nm,i) f x` attempts to set the tracing variable associated with the string `nm` to value `i`. Then it evaluates `f x` and returns the resulting value after restoring the trace level of `nm`.

Failure

Fails if the name given is not associated with a registered tracing variable. Also fails if the function invocation fails.

Example

```
> load "mesonLib";
val it = (): unit

> trace ("meson",2) prove
    (concl SKOLEM_THM,mesonLib.MESON_TAC []);
0 inferences so far. Searching with maximum size 0.
0 inferences so far. Searching with maximum size 1.
Internal goal solved with 2 MESON inferences.
```

```
2 inferences so far. Searching with maximum size 0.
2 inferences so far. Searching with maximum size 1.
Internal goal solved with 4 MESON inferences.
4 inferences so far. Searching with maximum size 0.
4 inferences so far. Searching with maximum size 1.
Internal goal solved with 6 MESON inferences.
6 inferences so far. Searching with maximum size 0.
6 inferences so far. Searching with maximum size 1.
Internal goal solved with 8 MESON inferences.
  solved with 8 MESON inferences.
val it = ⊢ ∀P. (∀x. ∃y. P x y) ⇔ ∃f. ∀x. P x (f x): thm

> traces();
val it =
   [BasicProvers.var_eq_old: 0 [0..1], Definition.TC extraction: 0 [0..5],
    Definition.allow_schema_definition: 0 [0..1],
    Definition.auto Defn.tgoal: 1 [0..1],
    Definition.delete support defs: 1 [0..1],
    Definition.induction derivation: 0 [0..1],
    Definition.storage_message: 1 [0..1],
    Definition.termination candidates: 0 [0..4],
    EmitTeX: K&R record type defns: 1 [0..1],
    EmitTeX: dollar parens: 1 [0..1],
    EmitTeX: print datatype names as types: 0 [0..1],
    EmitTeX: print datatypes compactly: 0 [0..1],
    EmitTeX: print thm turnstiles: 1 [0..1],
    Goalstack.howmany_printed_assums: 1000000 [0..1000000],
    Goalstack.howmany_printed_subgoals: 10 [0..10000],
    Goalstack.other_subgoals_pretty_limit: 100 [0..100000],
    Goalstack.print_assums_reversed: 0 [0..1],
    Goalstack.print_goal_at_top: 0 [0..1],
    Goalstack.print_goal_fvs: 0 [0..1],
    Goalstack.show_proved_subtheorems: 1 [0..1],
    Goalstack.show_stack_subgoal_count: 1 [0..1], Greek tyvars: 1 [0..1],
    Ho_Rewrite: 0 [0..1], HolSatLib_warn: 1 [0..1],
    PAT_ABBREV_TAC: match var/const: 1 [0..1], PP.avoid_unicode: 0 [0..1],
    PP.catch_withpp_err: 1 [0..1], PP.print_firstcasebar: 0 [0..1],
    PP.qblock_smash_limit: 4 [0..1000], PPBackEnd show types: 1 [0..1],
    PPBackEnd use annotations: 1 [0..1], PPBackEnd use css: 1 [0..1],
    PPBackEnd use styles: 1 [0..1],
```

```
    Parse.unicode_trace_off_complaints: 1 [0..1], QUANT_INST_DEBUG: 0 [0..3],
    QUANT_INST_DEBUG_DEPTH: 5 [0..2000],
    QUANT_INST___REC_DEPTH: 250 [0..20000],
    QUANT_INST___print_term_length: 2000 [0..2000], RealArith dp: 0 [0..1],
    Rewrite: 0 [0..1], Theory.allow_rebinds: 0 [0..1],
    Theory.debug: 0 [0..1], Theory.save_thm_reporting: 1 [0..2],
    TheoryPP.include_html_docs: 1 [0..1], Unicode: 1 [0..1],
    Unicode Univ printing: 1 [0..1], Univ pretty-printing: 1 [0..1],
    Vartype Format Complaint: 1 [0..1], ambiguous grammar warning: 1 [0..2],
    assumptions: 0 [0..1], computeLib.auto_import_definitions: 1 [0..1],
    guess overloads: 1 [0..1], inddef strict: 0 [0..1], meson: 1 [0..2],
    metis: 1 [0..10], normalForms: 0 [0..10],
    notify type variable guesses: 1 [0..1], numeral types: 0 [0..1],
    paranoid string literal printing: 0 [0..1], pp_annotations: 1 [0..1],
    pp_array_types: 1 [0..1], pp_avoids_symbol_merges: 1 [0..1],
    pp_cases: 1 [0..1], pp_dollar_escapes: 1 [0..1], pp_num_types: 1 [0..1],
    pp_unambiguous_comprehensions: 0 [0..2], print_tyabbrevs: 1 [0..1],
    report_thy_times: 1 [0..1], show_alias_printing_choices: 1 [0..1],
    show_typecheck_errors: 0 [0..1], simplifier: 0 [0..7],
    syntax_error: 1 [0..1], types: 0 [0..2], use pmatch_pp: 1 [0..1]]:
   trace_elt list
```

See also

`Feedback`, `Feedback.register_trace`, `Feedback.reset_trace`, `Feedback.reset_traces`, `Feedback.set_trace`, `Feedback.traces`, `Lib.with_flag`

traces (Feedback)

```
Feedback.traces : unit -> {name : string, max : int, aliases : string list,
                  trace_level : int, default : int} list
```

Returns a list of registered tracing variables.

The function `traces` is part of the interface to a collection of variables that control the verboseness of various tools within the system. Tracing can be useful both when debugging proofs (with the simplifier for example), and also as a guide to how an automatic proof is proceeding (with `mesonLib` for example).

Failure

Never fails.

See also

`Feedback.register_trace`, `Feedback.set_trace`, `Feedback.reset_trace`, `Feedback.reset_traces`, `Feedback.trace`

train_tnn (mlTreeNeuralNetwork)

```
mlTreeNeuralNetwork.train_tnn : schedule -> tnn -> tnnex * tnnex -> tnn
```

Train a tree neural network (TNN) on a set of examples via backpropagation to minimize mean square error.

Hyperparameters such as batch size, learning rate and number of epochs can be set in the schedule arguments. The initial TNN can be constructed by calling `mlTreeNeuralNetwork.random_tnn`. Examples consists of a term `t` and a list `l`. The term `t` is expected to be lambda-free with each operator appearing with a unique arity. The list `l` is expected to be a list of real numbers between 0 and 1. In the case of a simple objective each example `(t,l)` is to be written as `[(h(t),l)]` where `h` is a variable representing the head network. For multiple objectives, one can write `[(h1(t),l1),...,(hn(t),ln)]` for a single example. The created list of examples is to be split into a training set and a test set (possibly empty).

Failure

Fails when dimension constraints are not respected (see `mlTreeNeuralNetwork.random_tnn`) or a variable/constant from the examples is not defined in the TNN.

Comments

See the end of the file `src/AI/machine_learning/mlTreeNeuralNetwork.sml` for a toy example.

See also

`mlTreeNeuralNetwork.random_tnn`

TRANS (Thm)

```
Thm.TRANS : (thm -> thm -> thm)
```

Uses transitivity of equality on two equational theorems.

When applied to a theorem `A1 |- t1 = t2` and a theorem `A2 |- t2 = t3`, the inference rule `TRANS` returns the theorem `A1 u A2 |- t1 = t3`.

```
 A1 |- t1 = t2   A2 |- t2 = t3
------------------------------  TRANS
      A1 u A2 |- t1 = t3
```

Failure

Fails unless the theorems are equational, with the right side of the first being the same as the left side of the second.

Example

```
- val t1 = ASSUME ``a:bool = b`` and t2 = ASSUME ``b:bool = c``;
val t1 = [.] |- a = b : thm
val t2 = [.] |- b = c : thm

- TRANS t1 t2;
val it = [..] |- a = c : thm
```

See also

`Thm.EQ_MP`, `Drule.IMP_TRANS`, `Thm.REFL`, `Thm.SYM`

TRANS_TAC (Tactic)

```
Tactic.TRANS_TAC : thm -> term -> tactic
```

Applies transitivity theorem to goal with chosen intermediate term.

When applied to a 'transitivity' theorem, i.e. one of the form

```
|- !xs. R1 x y /\ R2 y z ==> R3 x z
```

and a term `t`, `TRANS_TAC` produces a tactic that reduces a goal with conclusion of the form `R3 s u` to one with conclusion `R1 s t /\ R2 t u`.

```
     A ?- R3 s u
======================== TRANS_TAC (|- !xs. R1 x y /\ R2 y z ==> R3 x z) `t`
 A ?- R1 s t /\ R2 t u
```

Example

Consider the simple inequality goal:

```
> g `n < (m + 2) * (n + 1)`;
```

We can use the following transitivity theorem

```
> LESS_EQ_LESS_TRANS;
val it = |- !m n p. m <= n /\ n < p ==> m < p: thm


# e (TRANS_TAC LESS_EQ_LESS_TRANS ``1 * (n + 1)``);
OK..
1 subgoal:
val it =

 n <= 1 * (n + 1) /\ 1 * (n + 1) < (m + 2) * (n + 1)

 : proof
```

Failure

Fails unless the input theorem is of the expected form (some of the relations `R1`, `R2` and `R3` may be, and often are, the same) and the conclusion matches the goal, in the usual sense of higher-order matching.

Comments

The effect of `TRANS_TAC th t` can often be replicated by the more primitive tactic sequence `MATCH_MP_TAC th THEN EXISTS_TAC t`. The use of `TRANS_TAC` is not only less verbose, but it is also more general in that it ensures correct type-instantiation of the theorem, whereas in highly polymorphic theorems the use of `MATCH_MP_TAC` may leave the wrong types for the subsequent `EXISTS_TAC` step.

If `R1 x y`, etc. are actually overloads of negated terms, e.g., `~(R1' y x)`, `TRANS_TAC` can still work. Such overloads are common for many definitions of "less" as an overload of "not less-or-equal", i.e. `x < y` is an overload of `~(y <= x)`.

See also

`MATCH_MP_TAC`, `TRANS`

transform (computeLib)

```
computeLib.type transform
```

Type of elements in compset

An element of a compset can map to a collection of rewrite rules or a conversion (or both, in some cases). The type `transform` is declared as follows:

```
datatype transform
   = Conversion of (term -> thm * db fterm)
   | RRules of thm list
```

Failure

Can not fail.

See also

`computeLib.listItems`

triple (Lib)

```
Lib.triple : 'a -> 'b -> 'c -> 'a * 'b * 'c
```

Makes three values into a triple.

`triple x y z` returns `(x, y, z)`.

Failure

Never fails.

See also

`Lib.triple_of_list`, `Lib.pair`, `Lib.quadruple`

triple_of_list (Lib)

```
Lib.triple_of_list : 'a list -> 'a * 'a * 'a
```

Turns a three-element list into a triple.

`triple_of_list [x, y, z]` returns `(x, y, z)`.

Failure

Fails if applied to a list that is not of length 3.

See also

`Lib.singleton_of_list`, `Lib.pair_of_list`, `Lib.quadruple_of_list`

TRUE_CONSEQ_CONV (ConseqConv)

```
ConseqConv.TRUE_CONSEQ_CONV : conseq_conv
```

Given a term `t` of type `bool` this consequence conversion returns the theorem `|- t ==> T`.

See also

`ConseqConv.FALSE_CONSEQ_CONV`, `ConseqConv.REFL_CONSEQ_CONV`, `ConseqConv.TRUE_FALSE_REFL_CONSEQ_CONV`

TRUE_FALSE_REFL_CONSEQ_CONV (ConseqConv)

```
ConseqConv.TRUE_FALSE_REFL_CONSEQ_CONV : directed_conseq_conv
```

Given a term `t` of type `bool` this directed consequence conversion returns the theorem `|- F ==> t` for `CONSEQ_CONV_STRENGTHEN_direction`, the theorem `|- t ==> T` for `CONSEQ_CONV_WEAKEN_direction` and `|- t = t` for `CONSEQ_CONV_UNKNOWN_direction`.

See also

`ConseqConv.TRUE_CONSEQ_CONV`, `ConseqConv.FALSE_CONSEQ_CONV`, `ConseqConv.REFL_CONSEQ_CONV`

try (Lib)

```
Lib.try : ('a -> 'b) -> 'a -> 'b
```

Apply a function and print any exceptions.

The application `try f x` evaluates `f x`; if this evaluation raises an exception `e`, then `e` is examined and some information about it is printed before `e` is re-raised. If `f x` evaluates to `y`, then `y` is returned.

Often, a `HOL_ERR` exception can propagate all the way to the top level. Unfortunately, the information held in the exception is not then printed. `try` can often display this information.

Failure

When application of the first argument to the second raises an exception.

Example

```
> mk_comb (T,F);
Exception- HOL_ERR (at Term.mk_comb: incompatible types) raised

> try mk_comb (T,F);
Exception- Type error in function application.
   Function: try mk_comb :
      (term -> (term * term) option) list -> term -> term
   Argument: (T, F) : term * term
   Reason:
      Can't unify (term -> (term * term) option) list to term * term
         (Incompatible types)
Fail "Static Errors" raised
```

Evaluation order can be significant. ML evaluates `try M N` by evaluating `M` (yielding `f` say) and `N` (yielding `x` say), and then `f` is applied to `x`. Any exceptions raised in the course of evaluating `M` or `N` will not be detected by `try`. In such cases it is better to use `Raise`. In the following example, the erroneous construction of an abstraction is not detected by `try` and the exception propagates all the way to the top level; however, `Raise` does handle the exception.

```
- try mk_comb (T, mk_abs(T,T));
! Uncaught exception:
! HOL_ERR

- mk_comb (T, mk_abs(T,T)) handle e => Raise e;
```

```
Exception raised at Term.mk_abs:
Bvar not a variable
! Uncaught exception:
! HOL_ERR
```

See also

`Feedback.Raise`, `Lib.trye`

TRY (Tactical)

```
Tactical.TRY : (tactic -> tactic)
```

Makes a tactic have no effect rather than fail.

For any tactic `T`, the application `TRY T` gives a new tactic which has the same effect as `T` if that succeeds, and otherwise has no effect.

Failure

The application of `TRY` to a tactic never fails. The resulting tactic never fails.

See also

`Tactical.CHANGED_TAC`, `Tactical.VALID`

TRY_CONV (Conv)

```
Conv.TRY_CONV : conv -> conv
```

Attempts to apply a conversion; applies identity conversion in case of failure.

`TRY_CONV c t` attempts to apply the conversion `c` to the term `t`; if this fails, then the identity conversion is applied instead. That is, if `c` is a conversion that maps a term `t` to the theorem `|- t = t'`, then the conversion `TRY_CONV c` also maps `t` to `|- t = t'`. But if `c` fails when applied to `t`, then `TRY_CONV c t` raises the `UNCHANGED` exception (which is understood to mean the instance of reflexivity, `|- t = t`). If `c` applied to `t` raises the `UNCHANGED` exception, then so too does `TRY_CONV c t`.

Failure

Never fails, except that the `UNCHANGED` exception can be raised.

See also

`Conv.UNCHANGED`, `Conv.QCHANGED_CONV`, `Conv.ALL_CONV`, `Conv.QCONV`

TRY_LT (Tactical)

```
Tactical.TRY_LT : (list_tactic -> list_tactic)
```

Makes a list-tactic have no effect rather than fail.

For any list-tactic `ltac`, the application `TRY_LT ltac` gives a new list-tactic which has the same effect as `ltac` if that succeeds, and otherwise has no effect.

Failure

The application of `TRY_LT` to a list-tactic never fails. The resulting list-tactic never fails.

See also

`Tactical.TRY`

TRYALL (Tactical)

```
Tactical.TRYALL : tactic -> list_tactic
```

Tries to apply a tactic to every goal in a list

If `tac` is a tactic, `TRYALL tac` is a list-tactic which, when applied to a list of goals, applies the tactic `tac` to each goal for which it succeeds. When `tac` fails on a goal, `TRYALL tac` has no effect on that goal.

Failure

The application of `TRYALL` to a tactic never fails. The resulting list-tactic never fails.

Example

Where `tac1` and `tac2` are tactics, `tac1 THEN_LT TRYALL tac2` is equivalent to `tac1 THEN TRY tac2`

See also

`Tactical.TRY`, `Tactical.THEN_LT`, `Tactical.THEN`, `Tactical.TRY`, `Tactical.ALLGOALS`

trye (Lib)

```
Lib.trye : ('a -> 'b) -> 'a -> 'b
```

Maps exceptions into `HOL_ERR`

The standard exception for HOL applications to raise is `HOL_ERR`. The use of a single exception simplifies the writing of exception handlers and facilities for decoding and printing error messages. However, ML functions that raise exceptions, such as `hd` and many others, are often used to implement HOL programs. In such cases, `trye` may be used to coerce exceptions into applications of `HOL_ERR`. Note however, that the `Interrupt` exception is not coerced by `trye`.

The application `trye f x` evaluates `f x`; if this evaluates to `y`, then `y` is returned. However, if evaluation raises an exception `e`, there are three cases: if `e` is `Interrupt`, then it is raised; if `e` is `HOL_ERR`, then it is raised; otherwise, `e` is mapped to an application of `HOL_ERR` and then raised.

Failure

Fails if the function application fails.

Example

```
> hd [];
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
Empty raised

> trye hd [];
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
HOL_ERR (at Lib.trye: original exn. not a HOL_ERR) raised

> trye (fn _ => raise Interrupt) 1;
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
Interrupt raised
```

See also

Lib, Feedback.Raise, Lib.try

tryfind (Lib)

```
Lib.tryfind : ('a -> 'b) -> 'a list -> 'b
```

Returns the result of the first successful application of a function to the elements of a list.

`tryfind f [x1,...,xn]` returns `(f xi)` for the first `xi` in the list for which application of `f` does not raise an exception. However, if `Interrupt` is raised in the course of some application of `f xi`, then `tryfind f [x1,...,xn]` raises `Interrupt`.

Failure

Fails if the application of `f` fails for all elements in the list. This will always be the case if the list is empty.

See also

`Lib.first`, `Lib.mem`, `Lib.exists`, `Lib.all`, `Lib.assoc`, `Lib.rev_assoc`, `Lib.assoc1`, `Lib.assoc2`

trypluck (Lib)

```
Lib.trypluck : ('a -> 'b) -> 'a list -> 'b * 'a list
```

Pull an element out of a list.

An invocation `trypluck f [x1,...,xk,...,xn]` returns a pair

```
(f(xk),[x1,...,xk-1,xk+1,...xn])
```

where `xk` has been lifted out of the list without disturbing the relative positions of the other elements. For this to happen, `f(xk)` must hold, and `f(xi)` must fail for `x1,...,xk-1`.

Failure

If the input list is empty. Also fails if `f` fails on every member of the list.

Example

```
> val (x,rst) = trypluck BETA_CONV [``1``,``(\x. x+2) 3``, ``p + q``];
val rst = [“1”, “p + q”]: term list
val x = ⊢ (λx. x + 2) 3 = 3 + 2: thm
```

See also

`Lib.first`, `Lib.filter`, `Lib.mapfilter`, `Lib.tryfind`

trypluck' (Lib)

```
Lib.trypluck' : ('a -> 'b option) -> 'a list -> ('b option * 'a list)
```

Pull an element out of a list.

An invocation `trypluck' f [x1,...,xk,...,xn]` returns either the pair

```
(f(xk),[x1,...,xk-1,xk+1,...xn])
```

where `xk` has been lifted out of the list without disturbing the relative positions of the other elements, where `f(xk)` is `SOME(v)`, and where `f(xi)` returns `NONE` for `x1,...,xk-1`; or it returns `(NONE,[x1,..xn]` when `f` applied to every element of the list returns `NONE`.

This is an 'option' version of the other library function `trypluck`.

Failure

Never fails.

See also

`Lib.first`, `Lib.filter`, `Lib.mapfilter`, `Lib.tryfind`, `Lib.trypluck`

ttt (tacticToe)

```
tacticToe.ttt : tactic
```

General purpose tactic relying on a automatic selection of tactics extracted from human-written proof scripts. It returns an automatically generated proof script that solves the goal. A good practice is to replace the call of `tacticToe.ttt` by the generated proof script.

Select relevant tactics and theorems for proving a goal using the k-nearest neighbor premise selection algorithm and relies on Monte Carlo tree search to expand multiple proof trees. In practice, this means that the intermediate goals created by better ranked tactics are explored more deeply.

Failure

Fails if the supplied goal does not contain boolean terms only. Or if the search saturates, this typically happens when there is not enough recorded tactics. Or if the search times out. This timeout can be modifed by `tacticToe.set_timeout`. Or if the proof fails to reconstruct.

Example

```
- load "tttUnfold"; load "tacticToe"; open tacticToe;

- tttUnfold.ttt_record (); (* takes multiple hours the first time it is called *)

- ttt ([],``1+1=2``);
```

Comments

See `src/tactictoe/README` for more information on how to record the tactic data. See more examples in `src/tactictoe/examples`.

See also

`tacticToe.tactictoe`

ty_antiq (Parse)

```
Parse.ty_antiq : hol_type -> term
```

Make a variable named `ty_antiq`.

Given a type `ty`, the ML invocation `ty_antiq ty` returns the HOL variable `ty_antiq : ty`. This provides a way to antiquote types into terms, which is necessary because the HOL term parser only allows terms to be antiquoted. The use of `ty_antiq` promotes a type to a term variable which can be antiquoted. The HOL parser detects occurrences of `ty_antiq ty` and inserts `ty` as a constraint.

Example

Suppose we want to constrain a term to have type `num list`, which is bound to ML value `ty`. Attempting to antiquote ty directly into the term won't work:

```
> val ty = ``:num list``;
val ty = “:num list”: hol_type

> “x : ^ty”;
Exception- Type error in function application.
   Function: Parse.Term : term frag list -> term
   Argument: [QUOTE " (*#loc 1 4*)x : ", ANTIQUOTE ty] :
      hol_type frag list
   Reason:
      Can't unify term (*Created from opaque signature*) with
         hol_type (*Created from opaque signature*)
         (Different type constructors)
Fail "Static Errors" raised
```

Use of `ty_antiq` solves the problem:

```
> ``x : ^(ty_antiq ty)``;
val it = “x”: term

> type_of it;
val it = “:num list”: hol_type
```

See also

`Parse.Term`

type_abbrev (Parse)

```
Parse.type_abbrev : string * hol_type -> unit
```

Establishes a type abbreviation.

A call to `type_abbrev(s,ty)` sets up a type abbreviation that will cause the parser to treat the string `s` as a synonym for the type `ty`. Moreover, if `ty` includes any type variables, then the abbreviation is treated as a type operator taking as many parameters as appear in `ty`. The order of the parameters will be the alphabetic ordering of the type variables' names.

Abbreviations work at the level of the names of type operators. It is thus possible to link a binary infix to an operator that is in turn an abbreviation.

Failure

Fails if the given type is just a type variable.

Example

This is a simple abbreviation.

```
> type_abbrev ("set", ``:'a -> bool``);
val it = () : unit

> ``:num set``;
val it = ``:num -> bool`` : hol_type
```

Here, the abbreviation is set up and provided with its own infix symbol.

```
- type_abbrev ("rfunc", ``:'b -> 'a``);
> val it = () : unit

- add_infix_type {Assoc = RIGHT, Name = "rfunc",
                  ParseName = SOME "<-", Prec = 50};
> val it = () : unit

- ``:'a <- bool``;
> val it = ``:bool -> 'a`` : hol_type

- dest_thy_type it;
> val it = {Args = [``:bool``, ``:'a``], Thy = "min", Tyop = "fun"} :
   {Args : hol_type list, Thy : string, Tyop : string}
```

Comments

As is common with most of the parsing and printing functions, there is a companion `temp_type_abbrev` function that does not cause the abbreviation effect to persist when the theory is exported. As the examples show, this entrypoint does not affect the pretty-printing of types. If printing of abbreviations is desired as well as parsing, the entrypoint `type_abbrev_pp` should be used.

See also

`Parse.add_infix_type`, `Parse.disable_tyabbrev_printing`, `Parse.remove_type_abbrev`, `Parse.thytype_abbrev`, `Parse.type_abbrev_pp`

type_abbrev_pp (Parse)

```
Parse.type_abbrev_pp : string * hol_type -> unit
```

Installs type abbreviation affecting parsing and printing.

As with `type_abbrev(s,ty)`, a call to `type_abbrev_pp(s,ty)` sets up the string `s` to be an abbrevation for the type `ty` when types are parsed. In addition, it causes the type pretty-printer to prefer the abbreviation when it comes to print types that match the implicit pattern specified by `ty` (which may include type variables).

Failure

Fails if the provided type is a single type variable.

Example

```
> type_abbrev_pp ("foo", ``:num -> 'a # num``);
val it = () : unit

> ``:bool foo``;
val it = ``:bool foo``: hol_type
```

```
> dest_thy_type it;
val it = {Args = [``:num``, ``:bool # num``],
          Thy = "min", Tyop = "fun"}:
   {Args: hol_type list, Thy: string, Tyop: string}
```

See also

`Parse.type_abbrev`

type_of (Term)

```
Term.type_of : term -> hol_type
```

Returns the type of a term.

Failure

Never fails.

Example

```
> type_of boolSyntax.universal;
val it = ":(α -> bool) -> bool": hol_type
```

type_rws (bossLib)

```
bossLib.type_rws : hol_type -> thm list
```

List rewrites for a concrete type.

An application `type_rws ty`, where `ty` is a declared datatype, returns a list of rewrite rules corresponding to the type. The list typically contains theorems about the distinctness and injectivity of constructors, the definition of the case constant introduced at the time the type was defined, and any extra rewrites coming from the use of records.

Failure

If ty is not a declared datatype.

Example

```
> type_rws ``:'a list``;
val it =
   [⊢ (∀v f. list_CASE [] v f = v) ∧
      ∀a0 a1 v f. list_CASE (a0::a1) v f = f a0 a1, ⊢ ∀a1 a0. [] ≠ a0::a1,
    ⊢ ∀a1 a0. a0::a1 ≠ [],
    ⊢ ∀a0 a1 a0' a1'. a0::a1 = a0'::a1' ⇔ a0 = a0' ∧ a1 = a1',
    ⊢ (∀f. list_size f [] = 0) ∧
      ∀f a0 a1. list_size f (a0::a1) = 1 + (f a0 + list_size f a1)]: thm list

> Hol_datatype `point = <| x:num ; y:num |>`;
val it = (): unit

> type_rws ``:point``;
val it =
   [⊢ ∀p g f.
        p with <|y updated_by f; x updated_by g|> =
        p with <|x updated_by g; y updated_by f|>,
    ⊢ (∀g f. y_fupd f ∘ x_fupd g = x_fupd g ∘ y_fupd f) ∧
      ∀h g f. y_fupd f ∘ x_fupd g ∘ h = x_fupd g ∘ y_fupd f ∘ h,
    ⊢ (∀n n0. (point n n0).x = n) ∧ ∀n n0. (point n n0).y = n0,
    ⊢ (∀p f. (p with y updated_by f).x = p.x) ∧
      (∀p f. (p with x updated_by f).y = p.y) ∧
      (∀p f. (p with x updated_by f).x = f p.x) ∧
      ∀p f. (p with y updated_by f).y = f p.y,
    ⊢ ∀p n0 n. p with <|x := n0; y := n|> = <|x := n0; y := n|>,
    ⊢ ∀n01 n1 n02 n2.
        <|x := n01; y := n1|> = <|x := n02; y := n2|> ⇔ n01 = n02 ∧ n1 = n2,
    ⊢ (∀p g f.
         p with <|x updated_by f; x updated_by g|> =
         p with x updated_by f ∘ g) ∧
      ∀p g f.
        p with <|y updated_by f; y updated_by g|> = p with y updated_by f ∘ g,
    ⊢ ((∀g f. x_fupd f ∘ x_fupd g = x_fupd (f ∘ g)) ∧
       ∀h g f. x_fupd f ∘ x_fupd g ∘ h = x_fupd (f ∘ g) ∘ h) ∧
```

```
    (∀g f. y_fupd f ∘ y_fupd g = y_fupd (f ∘ g)) ∧
    ∀h g f. y_fupd f ∘ y_fupd g ∘ h = y_fupd (f ∘ g) ∘ h,
  ⊢ ∀a0 a1 f. point_CASE (point a0 a1) f = f a0 a1,
  ⊢ ∀a0 a1 a0' a1'. point a0 a1 = point a0' a1' ⇔ a0 = a0' ∧ a1 = a1',
  ⊢ ∀a0 a1. point_size (point a0 a1) = 1 + (a0 + a1)]: thm list
```

Comments

`RW_TAC` and `SRW_TAC` automatically include these rewrites.

See also

`bossLib.rewrites`, `bossLib.RW_TAC`

type_ssfrag (simpLib)

```
simpLib.type_ssfrag : hol_type -> ssfrag
```

Returns a simpset fragment for simplifying types' constructors.

A call to `type_ssfrag ty` function returns a simpset fragment that embodies simplification routines for the type `ty`. The fragment includes rewrites that express injectivity and disjointness of constructors, and which simplify `case` expressions applied to terms that have constructors at the outermost level.

Failure

Fails if the string argument does not correspond to a type stored in the `TypeBase`.

Example

```
> val ss = simpLib.type_ssfrag ":'a list";
val ss =
   Simplification set fragment: Datatype list$list
   Rewrite rules:
      [list$list simpl. 5]
```

```
    ⊢ (∀f. list_size f [] = 0) ∧
      ∀f a0 a1. list_size f (a0::a1) = 1 + (f a0 + list_size f a1)
  [list$list simpl. 4]
    ⊢ ∀a0 a1 a0' a1'. a0::a1 = a0'::a1' ⇔ a0 = a0' ∧ a1 = a1'
  [list$list simpl. 3]  ⊢ ∀a1 a0. a0::a1 ≠ []
  [list$list simpl. 2]  ⊢ ∀a1 a0. [] ≠ a0::a1
  [list$list simpl. 1]
    ⊢ (∀v f. list_CASE [] v f = v) ∧
      ∀a0 a1 v f. list_CASE (a0::a1) v f = f a0 a1: ssfrag

> SIMP_CONV (bool_ss ++ ss) [] ``h::t = []``;
val it = ⊢ h::t = [] ⇔ F: thm
```

Comments

RW_TAC and SRW_TAC automatically include these simpset fragments.

See also

BasicProvers.RW_TAC, BasicProvers.srw_ss, bossLib.type_rws, simpLib.SIMP_CONV, TypeBase

type_subst (Type)

```
Type.type_subst : (hol_type,hol_type) subst -> hol_type -> hol_type
```

Instantiates types in a type.

If theta = [{redex_1,residue_1},...,{redex_n,residue_n}] is a (hol_type,hol_type) subst, where the redex_i are the types to be substituted for, and the residue_i the replacements, and ty is a type to instantiate, the call type_subst theta ty will replace each occurrence of a redex_i by the corresponding residue_i throughout ty. The replacements will be performed in parallel. If several of the type instantiations are applicable, the choice is undefined. Each redex_i ought to be a type variable, but if it isn't, it will never be replaced in ty. Also, it is not necessary that any or all of the types redex_1...redex_n should in fact appear in ty.

Failure

Never fails.

Example

```
> type_subst [alpha |-> bool] (Type `:'a # 'b`);
val it = “:bool # β”: hol_type

> type_subst [Type`:'a # 'b` |-> Type `:num`, alpha |-> bool]
            (Type`:'a # 'b`);
val it = “:bool # β”: hol_type
```

See also

`Term.inst`, `Thm.INST_TYPE`, `Lib.|->`, `Term.subst`

type_unify (boolSyntax)

```
boolSyntax.type_unify : hol_type -> hol_type -> (hol_type,hol_type) subst
```

Performs classical type unification.

Calculates a substitution `theta` such that instantiating each of the arguments with `theta` gives the same result type.

If `type_unify ty1 ty2` succeeds, then

```
type_subst (type_unify ty1 ty2) ty1 = type_subst (type_unify ty1 ty2) ty2
```

Failure

If no such substitution can be found. This could be due to incompatible type constructors, or the failing of an occurs check.

Example

```
> let val ty1 = Type `:'a -> 'b -> 'a`
     val ty2 = Type `:'a -> 'b -> 'b`
  in
    type_subst (type_unify ty1 ty2) ty1 = type_subst (type_unify ty1 ty2) ty2
  end;
val it = true: bool

> let val alpha_list = Type `:'a list`
  in
   type_unify alpha alpha_list handle e => Raise e
  end;
Exception- HOL_ERR (at boolSyntax.type_unify: occurs check) raised
```

Note that attempting to use `Type.match_type` in the first example results in immediate error, as it can only attempt to substitute the first argument to match the second:

```
> let val ty1 = Type `:'a -> 'b -> 'a`
    val ty2 = Type `:'a -> 'b -> 'b`
  in
   match_type ty1 ty2 handle e => Raise e
  end;
Exception- HOL_ERR (at Type.raw_match_type: double bind on type variable 'a) raised
```

See also

`boolSyntax.sep_type_unify`, `Type.match_type`, `Type.type_subst`, `Term.inst`

type_var_in (Type)

```
Type.type_var_in : hol_type -> hol_type -> bool
```

Checks if a type variable occurs in a type.

An invocation `type_var_in tyv ty` returns `true` if `tyv` occurs in `ty`. Otherwise, it returns `false`.

Failure

Fails if `tyv` is not a type variable.

Example

```
> type_var_in alpha (bool --> alpha);
val it = true: bool

> type_var_in alpha bool;
val it = false: bool
```

Comments

Can be useful in enforcing side conditions on inference rules.

See also

`Type.type_vars`, `Type.type_varsl`, `Type.exists_tyvar`

type_vars (Type)

```
Type.type_vars : hol_type -> hol_type list
```

Returns the set of type variables in a type.

An invocation `type_vars ty` returns a list representing the set of type variables occurring in `ty`.

Failure

Never fails.

Example

```
> type_vars ((alpha --> beta) --> bool --> beta);
val it = [":β", ":α"]: hol_type list
```

Comments

Code should not depend on how elements are arranged in the result of `type_vars`.

See also

`Type.type_vars_acc`, `Type.type_varsl`, `Type.type_var_in`, `Type.exists_tyvar`, `Type.polymorphic`, `Term.free_vars`

type_vars_acc (Type)

```
Type.type_vars_acc : hol_type -> hol_type list -> hol_type list
```

Returns the set of type variables in a type.

An invocation `type_vars_acc ty A` returns a list representing the set-theoretic union of the type variables occurring in `ty` and `A`.

Failure

Never fails.

Example

```
> type_vars_acc ((alpha --> beta) --> bool --> beta) [];
val it = [“:β”, “:α”]: hol_type list
```

Comments

Code should not depend on how elements are arranged in the result of `type_vars_acc`.

See also

`Type.type_vars`, `Type.type_varsl`, `Type.type_var_in`, `Type.exists_tyvar`, `Type.polymorphic`, `Term.free_vars`

type_vars_in_term (Term)

```
Term.type_vars_in_term : term -> hol_type list
```

Return the type variables occurring in a term.

An invocation `type_vars_in_term M` returns the set of type variables occurring in `M`.

Failure

Never fails.

Example

```
> type_vars_in_term (concl boolTheory.ONE_ONE_DEF);
val it = [":β", ":α"]: hol_type list
```

See also

`Term.free_vars`, `Type.type_vars`

type_varsl (Type)

```
Type.type_varsl : hol_type list -> hol_type list
```

Returns the set of type variables in a list of types.

An invocation `type_varsl [ty1,...,tyn]` returns a list representing the set-theoretic union of the type variables occurring in `ty1,...,tyn`.

Failure

Never fails.

Example

```
> type_varsl [alpha, beta, bool, ((alpha --> beta) --> bool --> beta)];
val it = [“:β”, “:α”]: hol_type list
```

Comments

Code should not depend on how elements are arranged in the result of `type_varsl`.

See also

`Type.type_vars`, `Type.type_vars_acc`, `Type.type_var_in`, `Type.exists_tyvar`, `Type.polymorph`
`Term.free_vars`

TypeBase (structure)

```
structure TypeBase
```

A database of facts stemming from datatype declarations.

The structure `TypeBase` provides an interface to a database that is updated when a new datatype is introduced with `Hol_datatype`. When a new datatype is declared, a collection of theorems “about” the type can be automatically derived. These are indeed proved, and are stored in the current theory segment. They are also automatically stored in `TypeBase`.

The interface to `TypeBase` is intended to provide support for writers of high-level tools for reasoning about datatypes.

Example

```
> Datatype `tree = Leaf | Node 'a tree tree`;
<<HOL message: Defined type: "tree">>
val it = () : unit

> TypeBase.read {Thy = current_theory(), Tyop = "tree"};
val it =
```

```
SOME
 ----------------------
 ----------------------
 HOL datatype: "scratch$tree"
 Primitive recursion:
  |- !f0 f1.
        ?fn.
            (fn Leaf = f0) /\
            !a0 a1 a2. fn (Node a0 a1 a2) = f1 a0 a1 a2 (fn a1) (fn a2)
 Case analysis:
  |- (!v f. tree_CASE Leaf v f = v) /\
     !a0 a1 a2 v f. tree_CASE (Node a0 a1 a2) v f = f a0 a1 a2
 Size:
  |- (!f. tree_size f Leaf = 0) /\
     !f a0 a1 a2.
         tree_size f (Node a0 a1 a2) =
         1 + (f a0 + (tree_size f a1 + tree_size f a2))
 Induction:
  |- !P.
         P Leaf /\ (!t t0. P t /\ P t0 ==> !a. P (Node a t t0)) ==>
         !t. P t
 Case completeness: |- !tt. (tt = Leaf) \/ ?a t t0. tt = Node a t t0
 Case-const equality split:
  |- (tree_CASE x v f = v') <=>
     (x = Leaf) /\ (v = v') \/
     ?a t t0. (x = Node a t t0) /\ (f a t t0 = v')
 Extras: [ ]
 One-to-one:
  |- !a0 a1 a2 a0' a1' a2'.
         (Node a0 a1 a2 = Node a0' a1' a2') <=>
         (a0 = a0') /\ (a1 = a1') /\ (a2 = a2')
 Distinctness: |- !a2 a1 a0. Leaf <> Node a0 a1 a2: tyinfo option
```

See also

bossLib.Datatype

types (Theory)

```
Theory.types : string -> (string * int) list
```

Lists the types in the named theory.

The function `types` should be applied to a string which is the name of an ancestor theory (including the current theory; the special string `"-"` is always interpreted as the current theory). It returns a list of all the type constructors declared in the named theory, in the form of arity-name pairs.

Failure

Fails unless the named theory is an ancestor, or the current theory.

Example

```
> load "bossLib";
val it = (): unit

> itlist union (map types (ancestry "-")) [];
val it =
   [("hrat", 0), ("hreal", 0), ("real", 0), ("bit0", 1), ("bit1", 1),
    ("cart", 2), ("finite_image", 1), ("ordering", 0), ("char", 0),
    ("recspace", 1), ("cv", 0), ("one", 0), ("sum", 2), ("option", 1),
    ("list", 1), ("prod", 2), ("bool", 0), ("fun", 2), ("ind", 0),
    ("itself", 1), ("num", 0), ("int", 0), ("example", 0), ("atree", 2)]:
   (string * int) list
```

See also

`Theory.constants`, `Theory.current_axioms`, `Theory.current_definitions`, `Theory.current_theorems`, `Theory.new_type`, `Definition.new_type_definition`, `Theory.parents`, `Theory.ancestry`

tyvar_sequence (boolLib)

```
boolLib.tyvar_sequence : int -> hol_type list
```

Generates a canonical list of distinct type variables.

A call to `tyvar_sequence n` generates a list consisting of `n` distinct type variables, with early members of the sequence being `:'a` ("alpha"), `:'b` ("beta") etc. After the first 26 members of the list, the remainder are of the form `:'a1`, `:'a2` etc.

Failure

Never fails. If `n` is negative the generated list is empty.

Example

```
> tyvar_sequence 3;
val it = [":α", ":β", ":γ"]: hol_type list
```

See also

`Type.mk_vartype`

U (Lib)

```
Lib.U : ''a list list -> ''a list
```

Takes the union of a list of sets.

An application `U [l1, ..., ln]` is equivalent to `union l1 (... (union ln-1, ln)...)`. Thus, every element that occurs in one of the lists will appear in the result.

Failure

Never fails.

Example

```
> U [[1,2,3], [4,5,6], [1,2,5]];
val it = [3, 6, 4, 1, 2, 5]: int list
```

Comments

The order in which the elements occur in the resulting list should not be depended upon.

A high-performance implementation of finite sets may be found in structure HOLset.

ML equality types are used in the implementation of U and its kin. This limits its applicability to types that allow equality. For other types, typically abstract ones, use the 'op_' variants.

See also

Lib.op_U, Lib.union, Lib.mk_set, Lib.mem, Lib.insert, Lib.set_eq, Lib.intersect, Lib.set_diff

UNABBREV_TAC (Q)

```
Q.UNABBREV_TAC : term quotation -> tactic
```

Removes an abbreviation from a goal's assumptions by substituting it out.

The argument to UNABBREV_TAC must be a quotation containing the name of a variable that is abbreviated in the current goal. In other words, when calling UNABBREV_TAC `v`, there must be an assumption of the form Abbrev(v = e) in the goal's assumptions. This assumption is removed, and all occurrences of the variable v in the goal are replaced by e. If there are two abbreviation assumptions for v in the goal, the more recent is removed.

Example

The goal

```
Abbrev(v = 2 * x + 1), v + x < 10 ?- P(v)
```

is transformed by `Q.UNABBREV_TAC` `` `v` `` to

```
2 * x + 1 + x < 10 ?- P(2 * x + 1)
```

Failure

Fails if there is no abbreviation of the required form in the goal's assumptions, or if the quotation doesn't parse to a variable.

See also

`BasicProvers.Abbr`, `Q.ABBREV_TAC`

UNBETA_CONV (Conv)

```
Conv.UNBETA_CONV : term -> conv
```

Returns a reversed instance of beta-reduction.

`UNBETA_CONV t1 t2` returns a theorem of the form

```
|- t2 = (\v. t') t1
```

The choice of `v` and the nature of `t'` depend on whether or `t1` is a variable. If so, then `v` will be `t1` and `t'` will be `t2`. Otherwise, `v` will be generated with `genvar` and `t'` will be the result of substituting `v` for `t1`, wherever it occurs.

Failure

Never fails.

Comments

Very useful for setting up a higher-order match by hand. The use of `genvar` is predicated on the assumption that it will later be eliminated through the application of the function term to some other argument.

See also

Thm.BETA_CONV

UNCHANGED (Conv)

```
Conv.UNCHANGED : exception
```

Raised by a conversion to indicate that a term should remain unchanged.

When a conversion `c` is applied to a term `t` this can raise the exception `UNCHANGED` to indicate that `t` should not be changed to another term `t'`.

Since in this case we have a function raising an exception, we describe this as failure of the function `c`. However it may be the intended result (as used, for example, by `ALL_CONV` or `TRY_CONV`).

When conversions are combined using `THENC` or `ORELSEC`, raising `UNCHANGED` is treated as though `|- t = t` were returned.

When a conversion `c` is used to produce an inference rule `CONV_RULE c` or a tactic `CONV_TAC c`, and `c` raises `UNCHANGED`, the rule `CONV_RULE c` or tactic `CONV_TAC c` succeeds, returning the theorem or goal unchanged.

See also

Abbrev.conv, Conv.QCONV, Conv.QCHANGED_CONV, Conv.ALL_CONV, Conv.TRY_CONV, Conv.CONV_RULE, Tactic.CONV_TAC, Conv.THENC, Conv.ORELSEC

uncurry (Lib)

```
Lib.uncurry : ('a -> 'b -> 'c) -> ('a * 'b) -> 'c
```

Converts a function taking two arguments into a function taking a single paired argument.

The application `uncurry f` returns `fn (x,y) => f x y`, so that

```
uncurry f (x,y) = f x y
```

Failure

Never fails.

Example

```
> fun add x y = x + y
val add = fn: int -> int -> int

> uncurry add (3,4);
val it = 7: int
```

See also

`Lib`, `Lib.curry`

UNCURRY_CONV (PairRules)

```
PairRules.UNCURRY_CONV : conv
```

Uncurrys an application of an abstraction.

Example

```
> PairRules.UNCURRY_CONV (Term `(\x y. x + y) 1 2`);
val it = ⊢ (λx y. x + y) 1 2 = (λ(x,y). x + y) (1,2): thm
```

Failure

`UNCURRY_CONV tm` fails if `tm` is not double abstraction applied to two arguments

See also

`PairRules.CURRY_CONV`

UNCURRY_EXISTS_CONV (PairRules)

```
PairRules.UNCURRY_EXISTS_CONV : conv
```

Uncurrys consecutive existential quantifications into a paired existential quantification.

Example

```
> PairRules.UNCURRY_EXISTS_CONV (Term `?x y. x + y = y + x`);
val it = ⊢ (∃x y. x + y = y + x) ⇔ ∃(x,y). x + y = y + x: thm

> PairRules.UNCURRY_EXISTS_CONV (Term `?(w,x) (y,z). w+x+y+z = z+y+x+w`);
val it =
   ⊢ (∃(w,x) (y,z). w + x + y + z = z + y + x + w) ⇔
     ∃((w,x),y,z). w + x + y + z = z + y + x + w: thm
```

Failure

`UNCURRY_EXISTS_CONV tm` fails if `tm` is not a consecutive existential quantification.

See also

`PairRules.CURRY_CONV`, `PairRules.UNCURRY_CONV`, `PairRules.CURRY_EXISTS_CONV`, `PairRules.CURRY_FORALL_CONV`, `PairRules.UNCURRY_FORALL_CONV`

UNCURRY_FORALL_CONV (PairRules)

```
PairRules.UNCURRY_FORALL_CONV : conv
```

Uncurrys consecutive universal quantifications into a paired universal quantification.

Example

```
> PairRules.UNCURRY_FORALL_CONV (Term `!x y. x + y = y + x`);
val it = ⊢ (∀x y. x + y = y + x) ⇔ ∀(x,y). x + y = y + x: thm

> PairRules.UNCURRY_FORALL_CONV (Term `!(w,x) (y,z). w+x+y+z = z+y+x+w`);
val it =
   ⊢ (∀(w,x) (y,z). w + x + y + z = z + y + x + w) ⇔
     ∀((w,x),y,z). w + x + y + z = z + y + x + w: thm
```

Failure

`UNCURRY_FORALL_CONV tm` fails if `tm` is not a consecutive universal quantification.

See also

`PairRules.CURRY_CONV`, `PairRules.UNCURRY_CONV`, `PairRules.CURRY_FORALL_CONV`, `PairRules.CURRY_EXISTS_CONV`, `PairRules.UNCURRY_EXISTS_CONV`

underAIs (Drule)

```
Drule.underAIs : (thm -> thm) -> (thm -> thm)
```

Applies a derived rule underneath external "guarding" of universal variables and implications.

A call to `underAIs f th` strips away external guarding in `th`, applies the function `f`, and then restores the original guarding. In this context, "guarding" is the presence of universal quantifications and antecedents in implications. Thus, this function sees the theorem `∀x. p x ==> ∀y. q y ∧ r x y ==> s` as the body `s`, guarded by the universal variables `x` and `y` and the assumptions `p x` and `q y ∧ r x y`. Negations in the body are not viewed as implications.

Failure

As all theorems are of this shape, the stripping and restoration of guarding always succeeds. However, this function will fail if `f` fails when applied to the theorem `A' |- s`, with `s` the body (as above), and `A'` the original hypotheses of theorem augmented with the antecendents of guarding implications.

Example

```
> underAIs (EXISTS (“∃m. (k * n) MOD m = 0”, “n:num”))
          arithmeticTheory.MOD_EQ_0;
val it = ⊢ ∀n. 0 < n ⇒ ∀k. ∃m. k * n MOD m = 0: thm
```

See also

`Drule.cj`, `Drule.iffLR`

UNDISCH (Drule)

```
Drule.UNDISCH : thm -> thm
```

Undischarges the antecedent of an implicative theorem.

```
    A |- t1 ==> t2
   ----------------  UNDISCH
     A, t1 |- t2
```

Note that `UNDISCH` treats `"~u"` as `"u ==> F"`.

Failure

`UNDISCH` will fail on theorems which are not implications or negations.

Comments

If the antecedent already appears in (or is alpha-equivalent to one of) the hypotheses, it will not be duplicated.

See also

Thm.DISCH, Drule.DISCH_ALL, Tactic.DISCH_TAC, Thm_cont.DISCH_THEN, Tactic.FILTER_DISCH_TAC, Tactic.FILTER_DISCH_THEN, Drule.NEG_DISCH, Tactic.STRIP_TAC, Drule.UNDISCH_ALL, Drule.UNDISCH_SPLIT, Drule.UNDISCH_TM, Tactic.UNDISCH_TAC

UNDISCH_ALL (Drule)

```
Drule.UNDISCH_ALL : thm -> thm
```

Iteratively undischarges antecedents in a chain of implications.

```
 A |- t1 ==> ... ==> tn ==> t
------------------------------  UNDISCH_ALL
     A, t1, ..., tn |- t
```

Note that UNDISCH_ALL treats `"~u"` as `"u ==> F"`.

Failure

UNDISCH_ALL never fails. When called on something other than an implication or negation, it simply returns its argument unchanged.

Comments

Identical or alpha-equivalent terms which are repeated in `A, "t1", ..., "tn"` will not be duplicated in the hypotheses of the resulting theorem.

See also

Thm.DISCH, Drule.DISCH_ALL, Tactic.DISCH_TAC, Thm_cont.DISCH_THEN, Drule.NEG_DISCH, Tactic.FILTER_DISCH_TAC, Tactic.FILTER_DISCH_THEN, Tactic.STRIP_TAC, Drule.UNDISCH, Tactic.UNDISCH_TAC

UNDISCH_SPLIT (Drule)

```
Drule.UNDISCH_SPLIT : thm -> thm
```

Undischarges the antecedent of an implicative theorem, and splits it into its conjuncts.

```
 A |- t1a /\ t1b /\ t1c ==> t2
------------------------------  UNDISCH_SPLIT
  A, t1a, t1b, t1c |- t2
```

Note that `UNDISCH_SPLIT` treats `"~u"` as `"u ==> F"`.

Failure

`UNDISCH_SPLIT` will fail on theorems which are not implications or negations.

Comments

If a conjunct of the antecedent already appears in (or is alpha-equivalent to one of) the hypotheses, it will not be duplicated.

See also

`Thm.DISCH`, `Drule.DISCH_ALL`, `Tactic.DISCH_TAC`, `Thm_cont.DISCH_THEN`, `Tactic.FILTER_DISCH_TAC`, `Tactic.FILTER_DISCH_THEN`, `Drule.NEG_DISCH`, `Tactic.STRIP_TAC`, `Drule.UNDISCH`, `Drule.UNDISCH_ALL`, `Drule.UNDISCH_TM`, `Tactic.UNDISCH_TAC`, `Drule.ASSUME_CONJS`

UNDISCH_TAC (Tactic)

```
Tactic.UNDISCH_TAC : term -> tactic
```

Undischarges an assumption and deletes all assumptions that are alpha-equivalent to it.

Let `a1` to `an` be the assumptions that are alpha-equivalent to `v`, then

```
              A ?- t
==============================  UNDISCH_TAC v
 A - {a1, ..., an} ?- v ==> t
```

In particular, if `v` is among the assumptions of the goal and no other assumption is alpha-equivalent to it, then `UNDISCH_TAC` behaves as the opposite of `DISCH_TAC`:

```
        A ?- t
====================  UNDISCH_TAC v
 A - {v} ?- v ==> t
```

Failure

`UNDISCH_TAC` fails if no assumption is alpha-equivalent to `v`.

Comments

In the typical use `v` is among the assumptions. If only a single assumption alpha-equivalent to `v`, but it is different from `v` then the action of `UNDISCH_TAC` can be seen as undischarging followed by alpha-conversion.

See also

`Thm.DISCH`, `Drule.DISCH_ALL`, `Tactic.DISCH_TAC`, `Thm_cont.DISCH_THEN`, `Drule.NEG_DISCH`, `Tactic.FILTER_DISCH_TAC`, `Tactic.FILTER_DISCH_THEN`, `Tactic.STRIP_TAC`, `Drule.UNDISCH`, `Drule.UNDISCH_ALL`

UNDISCH_THEN (Thm_cont)

```
Thm_cont.UNDISCH_THEN : term -> thm_tactic -> tactic
```

Discharges the assumption given and passes it to a theorem-tactic.

`UNDISCH_THEN` finds the first assumption equal to the term given, removes it from the assumption list, `ASSUME`s it, passes it to the theorem-tactic and then applies the consequent tactic. Thus:

```
UNDISCH_THEN t f ([a1,... ai, t, aj, ... an], goal) =
  f (ASSUME t) ([a1,... ai, aj,... an], goal)
```

For example, if

```
 A ?- t
========  f (ASSUME t1)
 B ?- v
```

then

```
 A u {t1} ?- t
===============  UNDISCH_THEN t1 f
    B ?- v
```

Failure

`UNDISCH_THEN` will fail on goals where the given term is not in the assumption list.

See also

`Tactical.PRED_ASSUM`, `Tactical.PAT_ASSUM`, `Thm.DISCH`, `Drule.DISCH_ALL`, `Tactic.DISCH_TAC`, `Thm_cont.DISCH_THEN`, `Drule.NEG_DISCH`, `Tactic.FILTER_DISCH_TAC`, `Tactic.FILTER_DISCH_THEN`, `Tactic.STRIP_TAC`, `Drule.UNDISCH`, `Drule.UNDISCH_ALL`, `Tactic.UNDISCH_TAC`

UNDISCH_TM (Drule)

```
Drule.UNDISCH_TM : thm -> term * thm
```

Undischarges the antecedent of an implicative theorem, also returning that antecedent.

```
 A |- t1 ==> t2
----------------  UNDISCH_TM
  A, t1 |- t2
```

Note that `UNDISCH_TM` treats `"~u"` as `"u ==> F"`.

Failure

`UNDISCH_TM` will fail on theorems which are not implications or negations.

Comments

If the antecedent already appears in (or is alpha-equivalent to one of) the hypotheses, it will not be duplicated.

`UNDISCH_TM` is similar to `UNDISCH` except that it also returns the antecedent concerned, which may be useful, for example, when the new assumption is subsequently to be discharged

See also

`Drule.UNDISCH`, `Thm.DISCH`

UNFOLD_CONV (unwindLib)

```
unwindLib.UNFOLD_CONV : (thm list -> conv)
```

Expands sub-components of a hardware description using their definitions.

`UNFOLD_CONV thl "t1 /\ ... /\ tn"` returns a theorem of the form:

```
B |- t1 /\ ... /\ tn = t1' /\ ... /\ tn'
```

where each `ti'` is the result of rewriting `ti` with the theorems in `thl`. The set of assumptions `B` is the union of the instantiated assumptions of the theorems used for rewriting. If none of the rewrites are applicable to a `ti`, it is unchanged.

Failure

Never fails.

Example

```
#UNFOLD_CONV [ASSUME "!in out. INV (in,out) = !(t:num). out t = ~(in t)"]
# "INV (l1,l2) /\ INV (l2,l3) /\ (!(t:num). l1 t = l2 (t-1) \/ l3 (t-1))";;
. |- INV(l1,l2) /\ INV(l2,l3) /\ (!t. l1 t = l2(t - 1) \/ l3(t - 1)) =
     (!t. l2 t = ~l1 t) /\
     (!t. l3 t = ~l2 t) /\
     (!t. l1 t = l2(t - 1) \/ l3(t - 1))
```

See also

unwindLib.UNFOLD_RIGHT_RULE

UNFOLD_RIGHT_RULE (unwindLib)

unwindLib.UNFOLD_RIGHT_RULE : (thm list -> thm -> thm)

Expands sub-components of a hardware description using their definitions.

UNFOLD_RIGHT_RULE thl behaves as follows:

```
      A |- !z1 ... zr. t = ?y1 ... yp. t1 /\ ... /\ tn
   ------------------------------------------------------
    B u A |- !z1 ... zr. t = ?y1 ... yp. t1' /\ ... /\ tn'
```

where each ti' is the result of rewriting ti with the theorems in thl. The set of assumptions B is the union of the instantiated assumptions of the theorems used for rewriting. If none of the rewrites are applicable to a ti, it is unchanged.

Failure

Fails if the second argument is not of the required form, though either or both of r and p may be zero.

Example

```
#UNFOLD_RIGHT_RULE [ASSUME "!in out. INV(in,out) = !(t:num). out t = ~(in t)"]
# (ASSUME "!(in:num->bool) out. BUF(in,out) = ?l. INV(in,l) /\ INV(l,out)");;
.. |- !in out.
       BUF(in,out) = (?l. (!t. l t = ~in t) /\ (!t. out t = ~l t))
```

See also

`unwindLib.UNFOLD_CONV`

union (Lib)

```
Lib.union : ''a list -> ''a list -> ''a list
```

Computes the union of two 'sets'.

If `l1` and `l2` are both 'sets' (lists with no repeated members), `union l1 l2` returns the set union of `l1` and `l2`. In the case that `l1` or `l2` is not a set, all the user can depend on is that `union l1 l2` returns a list `l3` such that every unique element of `l1` and `l2` is in `l3` and each element of `l3` is found in either `l1` or `l2`.

Failure

Never fails.

Example

```
> union [1,2,3] [1,5,4,3];
val it = [2, 1, 5, 4, 3]: int list

> union [1,1,1] [1,2,3,2];
val it = [1, 2, 3, 2]: int list

> union [1,2,3,2] [1,1,1] ;
val it = [3, 2, 1, 1, 1]: int list
```

Comments

Do not make the assumption that the order of items in the list returned by `union` is fixed. Later implementations may use different algorithms, and return a different concrete result while still meeting the specification.

A high-performance implementation of finite sets may be found in structure `HOLset`.

ML equality types are used in the implementation of `union` and its kin. This limits its applicability to types that allow equality. For other types, typically abstract ones, use the 'op_' variants.

See also

`Lib.op_union`, `Lib.U`, `Lib.mk_set`, `Lib.mem`, `Lib.insert`, `Lib.set_eq`, `Lib.intersect`, `Lib.set_diff`, `Lib.subtract`

UNION_CONV (pred_setLib)

```
pred_setLib.UNION_CONV : conv -> conv
```

Reduce `{t1;...;tn} UNION s` to `t1 INSERT (... (tn INSERT s))`.

The function `UNION_CONV` is a parameterized conversion for reducing sets of the form `{t1;...;tn} UNION s`, where `{t1;...;tn}` and `s` are sets of type `ty->bool`. The first argument to `UNION_CONV` is expected to be a conversion that decides equality between values of the base type `ty`. Given an equation `e1 = e2`, where `e1` and `e2` are terms of type `ty`, this conversion should return the theorem `|- (e1 = e2) = T` or the theorem `|- (e1 = e2) = F`, as appropriate.

Given such a conversion, the function `UNION_CONV` returns a conversion that maps a term of the form `{t1;...;tn} UNION s` to the theorem

```
|- {t1;...;tn} UNION s = ti INSERT ... (tj INSERT s)
```

where `{ti;...;tj}` is the set of all terms `t` that occur as elements of `{t1;...;tn}` for which the conversion `IN_CONV conv` fails to prove that `|- (t IN s) = T` (that is, either by proving `|- (t IN s) = F` instead, or by failing outright).

Example

In the following example, `REDUCE_CONV` is supplied as a parameter to `UNION_CONV` and used to test for membership of each element of the first finite set `{1;2;3}` of the union in the second finite set `{SUC 0;3;4}`.

```
- UNION_CONV REDUCE_CONV (Term`{1;2;3} UNION {SUC 0;3;4}`);
> val it = |- {1; 2; 3} UNION {SUC 0; 3; 4} = {2; SUC 0; 3; 4} : thm
```

The result is `{2;SUC 0;3;4}`, rather than `{1;2;SUC 0;3;4}`, because `UNION_CONV` is able by means of a call to

```
- IN_CONV REDUCE_CONV (Term`1 IN {SUC 0;3;4}`);
```

to prove that `1` is already an element of the set `{SUC 0;3;4}`.

The conversion supplied to `UNION_CONV` need not actually prove equality of elements, if simplification of the resulting set is not desired. For example:

```
- UNION_CONV NO_CONV ``{1;2;3} UNION {SUC 0;3;4}``;
> val it = |- {1;2;3} UNION {SUC 0;3;4} = {1;2;SUC 0;3;4} : thm
```

In this case, the resulting set is just left unsimplified. Moreover, the second set argument to `UNION` need not be a finite set:

```
- UNION_CONV NO_CONV ``{1;2;3} UNION s``;
> val it = |- {1;2;3} UNION s = 1 INSERT (2 INSERT (3 INSERT s)) : thm
```

And, of course, in this case the conversion argument to `UNION_CONV` is irrelevant.

Failure

`UNION_CONV conv` fails if applied to a term not of the form `{t1;...;tn} UNION s`.

See also

`pred_setLib.IN_CONV`, `numLib.REDUCE_CONV`

unique_tmp_suffix (Portable)

```
Portable.unique_tmp_suffix : unit -> string
```

A short string suitable for embedding in temp filenames so that two processes writing to the same logical path pick disjoint temp paths and can each rename their own result into place without colliding.

Under Poly/ML the returned string is the decimal representation of the process id (`Posix.Process.pidToWord (Posix.ProcEnv.getpid ())`). Under Moscow ML, where HOL builds are never run in parallel, the result is derived from `OS.FileSys.tmpName`; it is still unique per call but is not the literal process id.

Failure

Never fails.

universal (boolSyntax)

```
boolSyntax.universal : term
```

Constant denoting universal quantification.

The ML variable `boolSyntax.universal` is bound to the term `bool$!`.

See also

`boolSyntax.equality`, `boolSyntax.implication`, `boolSyntax.select`, `boolSyntax.T`, `boolSyntax.F`, `boolSyntax.universal`, `boolSyntax.existential`, `boolSyntax.exists1`, `boolSyntax.conjunction`, `boolSyntax.disjunction`, `boolSyntax.negation`, `boolSyntax.condit` `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`

unmapped (computeLib)

```
computeLib.unmapped : compset -> (string * string) list
```

List unmapped elements in compset

The function `unmapped` takes a `compset` value and returns a listing of the elements of the compset that have no transformation attached to them.

Example

The listing omits constructors, but can include constants that effectively act as constructors for rewrites in the compset.

```
> val compset = reduceLib.num_compset;
val compset = <compset>: computeLib.compset

> computeLib.unmapped compset;
val it =
  [("BIT1", "arithmetic"),
   ("BIT2", "arithmetic"),
   ("ZERO", "arithmetic")]
  : (string * string) list
```

Example

In the following example, a function is added to a compset without also adding functions that get “called” by it:

```
> load "sortingTheory";
val it = (): unit

> sortingTheory.QSORT_DEF;
val it =
   |- (!ord. QSORT ord [] = []) /\
      !t ord h.
        QSORT ord (h::t) =
```

```
          (let (l1,l2) = PARTITION (\y. ord y h) t
          in
          QSORT ord l1 ++ [h] ++ QSORT ord l2) : thm

> val compset = computeLib.add_thms [sortingTheory.QSORT_DEF] compset;

> computeLib.unmapped compset;
val it =
   [("APPEND", "list"),
    ("BIT1", "arithmetic"),
    ("BIT2", "arithmetic"),
    ("PARTITION", "sorting"),
    ("UNCURRY", "pair"),
    ("ZERO", "arithmetic")]
:(string * string) list
```

Comments

Intended to support the construction of large compsets, where it is often unclear what functions and conversions still need to be added in order to make applications of `EVAL_CONV` terminate.

Failure

Never fails.

See also

`bossLib.EVAL`, `computeLib.listItems`

UNPBETA_CONV (PairRules)

```
PairRules.UNPBETA_CONV : (term -> conv)
```

Creates an application of a paired abstraction from a term.

The user nominates some pair structure of variables `p` and a term `t`, and `UNPBETA_CONV` turns `t` into an abstraction on `p` applied to `p`.

```
------------------  UNPBETA_CONV "p" "t"
 |- t = (\p. t) p
```

Failure

Fails if `p` is not a paired structure of variables.

See also

`PairRules.PBETA_CONV`, `PairedLambda.PAIRED_BETA_CONV`

UNWIND_ALL_BUT_CONV (unwindLib)

```
unwindLib.UNWIND_ALL_BUT_CONV : (string list -> conv)
```

Unwinds all lines of a device (except those in the argument list) as much as possible.

`UNWIND_ALL_BUT_CONV l` when applied to the following term:

```
"t1 /\ ... /\ eqn1 /\ ... /\ eqnm /\ ... /\ tn"
```

returns a theorem of the form:

```
|- t1  /\ ... /\ eqn1 /\ ... /\ eqnm /\ ... /\ tn =
   t1' /\ ... /\ eqn1 /\ ... /\ eqnm /\ ... /\ tn'
```

where `ti'` (for `1 <= i <= n`) is `ti` rewritten with the equations `eqni` (`1 <= i <= m`). These equations are those conjuncts with line name not in `l` (and which are equations).

Failure

Never fails but may loop indefinitely.

Example

```
#UNWIND_ALL_BUT_CONV [`l2`]
# "(!(x:num). l1 x = (l2 x) - 1) /\
#  (!x. f x = (l2 (x+1)) + (l1 (x+2))) /\
#  (!x. l2 x = 7)";;
|- (!x. l1 x = (l2 x) - 1) /\
   (!x. f x = (l2(x + 1)) + (l1(x + 2))) /\
   (!x. l2 x = 7) =
   (!x. l1 x = (l2 x) - 1) /\
   (!x. f x = (l2(x + 1)) + ((l2(x + 2)) - 1)) /\
   (!x. l2 x = 7)
```

See also

unwindLib.UNWIND_ONCE_CONV, unwindLib.UNWIND_CONV, unwindLib.UNWIND_AUTO_CONV, unwindLib.UNWIND_ALL_BUT_RIGHT_RULE, unwindLib.UNWIND_AUTO_RIGHT_RULE

UNWIND_ALL_BUT_RIGHT_RULE (unwindLib)

```
unwindLib.UNWIND_ALL_BUT_RIGHT_RULE : (string list -> thm -> thm)
```

Unwinds all lines of a device (except those in the argument list) as much as possible. `UNWIND_ALL_BUT_RIGHT_RULE l` behaves as follows:

```
    A |- !z1 ... zr.
          t =
          (?l1 ... lp. t1  /\ ... /\ eqn1 /\ ... /\ eqnm /\ ... /\ tn)
   ------------------------------------------------------------------
    A |- !z1 ... zr.
          t =
          (?l1 ... lp. t1' /\ ... /\ eqn1 /\ ... /\ eqnm /\ ... /\ tn')
```

where `ti'` (for `1 <= i <= n`) is `ti` rewritten with the equations `eqni` (`1 <= i <= m`). These equations are those conjuncts with line name not in `l` (and which are equations).

Failure

Fails if the argument theorem is not of the required form, though either or both of `p` and `r` may be zero. May loop indefinitely.

Example

```
#UNWIND_ALL_BUT_RIGHT_RULE [`l2`]
# (ASSUME
#   "!f. IMP(f) =
#     ?l2 l1.
#      (!(x:num). l1 x = (l2 x) - 1) /\
#      (!x. f x = (l2 (x+1)) + (l1 (x+2))) /\
#      (!x. l2 x = 7)");;
. |- !f.
      IMP f =
      (?l2 l1.
        (!x. l1 x = (l2 x) - 1) /\
        (!x. f x = (l2(x + 1)) + ((l2(x + 2)) - 1)) /\
        (!x. l2 x = 7))
```

See also

`unwindLib.UNWIND_AUTO_RIGHT_RULE`, `unwindLib.UNWIND_ALL_BUT_CONV`, `unwindLib.UNWIND_AUTO_CONV`, `unwindLib.UNWIND_ONCE_CONV`, `unwindLib.UNWIND_CONV`

UNWIND_AUTO_CONV (unwindLib)

```
unwindLib.UNWIND_AUTO_CONV : conv
```

Automatic unwinding of equations defining wire values in a standard device specification.

`UNWIND_AUTO_CONV "?l1 ... lm. t1 /\ ... /\ tn"` returns a theorem of the form:

```
   |- (?l1 ... lm. t1 /\ ... /\ tn) = (?l1 ... lm. t1' /\ ... /\ tn')
```

where `tj'` is `tj` rewritten with equations selected from the `ti`'s.

The function decides which equations to use for rewriting by performing a loop analysis on the graph representing the dependencies of the lines. By this means the term can be unwound as much as possible without the risk of looping. The user is left to deal with the recursive equations.

Failure

Fails if there is more than one equation for any line variable.

Example

```
#UNWIND_AUTO_CONV
# "(!(x:num). l1 x = (l2 x) - 1) /\
#  (!x. f x = (l2 (x+1)) + (l1 (x+2))) /\
#  (!x. l2 x = 7)";;
|- (!x. l1 x = (l2 x) - 1) /\
   (!x. f x = (l2(x + 1)) + (l1(x + 2))) /\
   (!x. l2 x = 7) =
   (!x. l1 x = 7 - 1) /\ (!x. f x = 7 + (7 - 1)) /\ (!x. l2 x = 7)
```

See also

`unwindLib.UNWIND_ONCE_CONV`, `unwindLib.UNWIND_CONV`, `unwindLib.UNWIND_ALL_BUT_CONV`, `unwindLib.UNWIND_ALL_BUT_RIGHT_RULE`, `unwindLib.UNWIND_AUTO_RIGHT_RULE`

UNWIND_AUTO_RIGHT_RULE (unwindLib)

```
unwindLib.UNWIND_AUTO_RIGHT_RULE : (thm -> thm)
```

Automatic unwinding of equations defining wire values in a standard device specification.

`UNWIND_AUTO_RIGHT_RULE` behaves as follows:

```
    A |- !z1 ... zr. t = ?l1 ... lm. t1  /\ ... /\ tn
```

```
   ------------------------------------------------
    A |- !z1 ... zr. t = ?l1 ... lm. t1' /\ ... /\ tn'
```

where `tj'` is `tj` rewritten with equations selected from the `ti`'s.

The function decides which equations to use for rewriting by performing a loop analysis on the graph representing the dependencies of the lines. By this means the term can be unwound as much as possible without the risk of looping. The user is left to deal with the recursive equations.

Failure

Fails if there is more than one equation for any line variable, or if the argument theorem is not of the required form, though either or both of `m` and `r` may be zero.

Example

```
#UNWIND_AUTO_RIGHT_RULE
# (ASSUME
#   "!f. IMP(f) =
#     ?l2 l1.
#      (!(x:num). l1 x = (l2 x) - 1) /\
#      (!x. f x = (l2 (x+1)) + (l1 (x+2))) /\
#      (!x. l2 x = 7)");;
. |- !f.
      IMP f =
      (?l2 l1.
        (!x. l1 x = 7 - 1) /\ (!x. f x = 7 + (7 - 1)) /\ (!x. l2 x = 7))
```

See also

`unwindLib.UNWIND_ALL_BUT_RIGHT_RULE`, `unwindLib.UNWIND_AUTO_CONV`, `unwindLib.UNWIND_ALL_BUT_CONV`, `unwindLib.UNWIND_ONCE_CONV`, `unwindLib.UNWIND_CONV`

UNWIND_CONV (unwindLib)

```
unwindLib.UNWIND_CONV : ((term -> bool) -> conv)
```

Unwinds device behaviour using selected line equations until no change.

`UNWIND_CONV p "t1 /\ ... /\ eqn1 /\ ... /\ eqnm /\ ... /\ tn"` returns a theorem of the form:

```
|- t1  /\ ... /\ eqn1 /\ ... /\ eqnm /\ ... /\ tn =
   t1' /\ ... /\ eqn1 /\ ... /\ eqnm /\ ... /\ tn'
```

where `ti'` (for `1 <= i <= n`) is `ti` rewritten with the equations `eqni` (`1 <= i <= m`). These equations are the conjuncts for which the predicate `p` is true. The `ti` terms are the conjuncts for which `p` is false. The rewriting is repeated until no changes take place.

Failure

Never fails but may loop indefinitely.

Example

```
#UNWIND_CONV (\tm. mem (line_name tm) [`l1`;`l2`])
# "(!(x:num). l1 x = (l2 x) - 1) /\
#  (!x. f x = (l2 (x+1)) + (l1 (x+2))) /\
#  (!x. l2 x = 7)";;
|- (!x. l1 x = (l2 x) - 1) /\
   (!x. f x = (l2(x + 1)) + (l1(x + 2))) /\
   (!x. l2 x = 7) =
   (!x. l1 x = (l2 x) - 1) /\ (!x. f x = 7 + (7 - 1)) /\ (!x. l2 x = 7)
```

See also

`unwindLib.UNWIND_ONCE_CONV`, `unwindLib.UNWIND_ALL_BUT_CONV`, `unwindLib.UNWIND_AUTO_CONV`, `unwindLib.UNWIND_ALL_BUT_RIGHT_RULE`, `unwindLib.UNWIND_AUTO_RIGHT_RULE`

UNWIND_ONCE_CONV (unwindLib)

```
unwindLib.UNWIND_ONCE_CONV : ((term -> bool) -> conv)
```

Basic conversion for parallel unwinding of equations defining wire values in a standard device specification.

`UNWIND_ONCE_CONV p tm` unwinds the conjunction `tm` using the equations selected by the predicate `p`. `tm` should be a conjunction, equivalent under associative-commutative reordering to:

```
t1 /\ t2 /\ ... /\ tn
```

`p` is used to partition the terms `ti` for `1 <= i <= n` into two disjoint sets:

```
REW = {{ti | p ti}}
OBJ = {{ti | ~p ti}}
```

The terms `ti` for which `p` is true are then used as a set of rewrite rules (thus they should be equations) to do a single top-down parallel rewrite of the remaining terms. The rewritten terms take the place of the original terms in the input conjunction. For example, if `tm` is:

```
t1 /\ t2 /\ t3 /\ t4
```

and `REW = {{t1,t3}}` then the result is:

```
|- t1 /\ t2 /\ t3 /\ t4 = t1 /\ t2' /\ t3 /\ t4'
```

where `ti'` is `ti` rewritten with the equations `REW`.

Failure

Never fails.

Example

```
> unwindLib.UNWIND_ONCE_CONV (fn tm => mem (unwindLib.line_name tm) [`l1`,`l2`])
  “(!(x:num). l1 x = (l2 x) - 1) /\
  (!x. f x = (l2 (x+1)) + (l1 (x+2))) /\
  (!x. l2 x = 7)”;
Exception- Type error in function application.
```

```
   Function: mem (unwindLib.line_name tm) : string list -> bool
   Argument: [[QUOTE " (*#loc 1 69*)l1"], [QUOTE " (*#loc 1 74*)l2"]]
      : 'a frag list list
   Reason:
      Can't unify string (*In Basis*) with 'a frag list (*In Basis*)
         (Different type constructors)
Fail "Static Errors" raised
```

See also

unwindLib.UNWIND_CONV, unwindLib.UNWIND_ALL_BUT_CONV, unwindLib.UNWIND_AUTO_CONV, unwindLib.UNWIND_ALL_BUT_RIGHT_RULE, unwindLib.UNWIND_AUTO_RIGHT_RULE

unzip (Lib)

```
Lib.unzip : ('a * 'b) list -> ('a list * 'b list)
```

Converts a list of pairs into a pair of lists.

`unzip [(x1,y1),...,(xn,yn)]` returns `([x1,...,xn],[y1,...,yn])`.

Failure

Never fails.

Comments

Identical to `Lib.split`.

See also

Lib.split, Lib.zip, Lib.combine

update_overload_maps (Parse)

```
Parse.update_overload_maps :
  string -> ({Name : string, Thy : string} list *
             {Name : string, Thy : string} list) -> unit
```

Adds to the parser's overloading maps.

The parser/pretty-printer for terms maintains two maps between constants and strings. From strings to terms, the map is from one string to a set of terms. Each term represents a possible overloading for the string. In the other direction, a term maps to just one string, its preferred representation.

The function `update_overload_maps` adds to (potentially overriding old mappings in) both of these maps. Its first parameter, a string, is the string involved in both directions. The two lists of `Name-Thy` records specify terms for the two maps. The first component of the tuple, specifies terms that the string will be overloaded to. (Note that it is perfectly reasonable to "overload" to just one term, and that this is the default situation for newly defined constants.)

The second component of the tuple sets the given string as the preferred identifier for the given terms.

Failure

Fails if any of the `Name-Thy` pairs doesn't correspond to an actual constant.

See also

`Parse.clear_overloads_on`, `Parse.hide`, `Parse.overload_on`, `Parse.remove_ovl_mapping`, `Parse.reveal`

upto (Lib)

```
Lib.upto : int -> int -> int list
```

Builds a list of integers.

An invocation `upto b t` returns the list `[b, b+1, ..., t]`, if `b <= t`. Otherwise, the empty list is returned.

Failure

Never fails.

Example

```
> upto 2 10;
val it = [2, 3, 4, 5, 6, 7, 8, 9, 10]: int list
```

uptodate_term (Theory)

```
Theory.uptodate_term : term -> bool
```

Tells if a term is out of date.

Operations in the current theory segment of HOL allow one to redefine types and constants. This can cause theorems to become invalid. As a result, HOL has a rudimentary consistency maintenance system built around the notion of whether type operators and term constants are "up-to-date".

An invocation `uptodate_term M` checks `M` to see if it has been built from any out-of-date components. The definition of `out-of-date` is mutually recursive among types, terms, and theorems. If `M` is a variable, it is out-of-date if its type is out-of-date. If `M` is a constant, it is out-of-date if it has been redeclared, or if its type is out-of-date, or if the witness theorem used to justify its existence is out-of-date. If `M` is a combination, it is out-of-date if either of its components are out-of-date. If `M` is an abstraction, it is out-of-date if either the bound variable or the body is out-of-date.

All items from ancestor theories are fixed, and unable to be overwritten, thus are always up-to-date.

Failure

Never fails.

Example

```
> Define `utdtfact x = if x=0 then 1 else x * utdtfact (x-1)`;
val it = ⊢ ∀x. utdtfact x = if x = 0 then 1 else x * utdtfact (x − 1): thm

> val M = Term `!x. 0 < utdtfact x`;
val M = “∀x. 0 < utdtfact x”: term

> uptodate_term M;
val it = true: bool

> delete_const "utdtfact";
val it = (): unit

> uptodate_term M;
val it = false: bool
```

See also

`Theory.uptodate_type`, `Theory.uptodate_thm`

uptodate_thm (Theory)

```
Theory.uptodate_thm : thm -> bool
```

Tells if a theorem is out of date.

Operations in the current theory segment of HOL allow one to redefine types and constants. This can cause theorems to become invalid. As a result, HOL has a rudimentary consistency maintenance system built around the notion of whether type operators and term constants are “up-to-date”.

An invocation `uptodate_thm th` should check `th` to see if it has been proved from any out-of-date components. However, HOL does not currently keep the proofs of theorems, so a simpler approach is taken. Instead, `th` is checked to see if its hypotheses and conclusions are up-to-date, and if any (locally asserted) axioms it depends on are also up-to-date.

All items from ancestor theories are fixed, and unable to be overwritten, thus are always up-to-date.

Failure

Never fails.

Example

```
Definition fact_def:
  fact (x:num) = if x=0 then 1 else x * fact (x-1)
End
Equations stored under "fact_def".
Induction stored under "fact_ind".
val fact_def = ⊢ ∀x. fact x = if x = 0 then 1 else x * fact (x − 1): thm

> val th = EVAL “fact 3”
val th = ⊢ fact 3 = fact 3: thm

> uptodate_thm th;
val it = true: bool

> delete_const "fact";
val it = (): unit

> uptodate_thm th;
val it = true: bool
```

Comments

It may happen that a theorem `th` is proved with the use of another theorem `th1` that subsequently becomes garbage because a constant `c` was deleted. If `c` does not occur in `th`, then `th` does not become garbage, which may be contrary to expectation. The conservative extension property of HOL says that `th` is still provable, even in the absence of `c`.

See also

`Theory.uptodate_type`, `Theory.uptodate_term`, `Theory.delete_const`, `Theory.delete_type`

uptodate_type (Theory)

```
Theory.uptodate_type : hol_type -> bool
```

Tells if a type is out of date.

Operations in the current theory segment of HOL allow one to redefine types and constants. This can cause theorems to become invalid. As a result, HOL has a rudimentary consistency maintenance system built around the notion of whether type operators and term constants are "up-to-date".

An invocation `uptodate_type ty`, checks `ty` to see if it has been built from any out of date components, returning `false` just in case it has. The definition of `out-of-date` is mutually recursive among types, terms, and theorems. A type variable never out-of-date. A compound type is out-of-date if either (a) its type operator is out-of-date, or (b) any of its argument types are out-of-date. A type operator is out-of-date if it has been re-declared or if the witness theorem used to justify the type in the call to `new_type_definition` is out-of-date. Only a component of the current theory segment may be out-of-date. All items from ancestor theories are fixed, and unable to be overwritten, thus are always up-to-date.

Failure

Never fails.

Example

```
> Hol_datatype `foo = A | B of 'a`;
val it = (): unit

> val ty = Type `:'a foo list`;
val ty = ":α foo list": hol_type
```

```
> uptodate_type ty;
val it = true: bool

> delete_type "foo";
val it = (): unit

> uptodate_type ty;
val it = false: bool
```

See also

Theory.uptodate_term, Theory.uptodate_thm

USE_SG_THEN (Tactical)

```
Tactical.USE_SG_THEN : thm_tactic -> int -> int -> list_tactic
```

Allows the user to use one subgoal to prove another

In USE_SG_THEN ttac nu np, of the current goal list, subgoal number nu can be used in proving subgoal number np. Subgoal number nu is used as a lemma by ttac to simplify subgoal number np. That is, if subgoal number nu is A ?- u, subgoal number np is A1 ?- t1, and

```
 A1 ?- t1
==========  ttac (u |- u)
 A2 ?- t2
```

then the list-tactic USE_SG_THEN ttac nu np gives this same result (new subgoal(s)) for subgoal np.

This list-tactic will be invalid unless A is a subset of A1.

Note that in the interactive system, subgoals are printed in reverse order of their numbering.

Failure

USE_SG_THEN will fail `ttac (u |- u)` fails on subgoal number np, or if indices np or nu are out of range. Note that the subgoals in the current subgoal list are numbered starting from 1.

Where two subgoals are similar and not easy to prove, one can be used to help prove the other.

Example

Here subgoal 1 is assumed, so as to help in proving subgoal 2.

```
r \/ s
------------------------------------
  0.  p
  1.  q


r
------------------------------------
  0.  p
  1.  q

2 subgoals
:
   proof

> elt (USE_SG_THEN ASSUME_TAC 1 2) ;
OK..
2 subgoals:
val it =

    0.  q
    1.  p
    2.  p'
   ------------------------------------
        q

    0.  q
    1.  p
```

```
   ------------------------------------
        p'
```

Here is an example where the assumptions differ. Subgoal 2 is used to solve subgoal 1, but the assumption `p'` of subgoal 2 remains to be proved. Without `VALIDATE_LT`, the list-tactic would be invalid.

```
r
------------------------------------
  0.  p'
  1.  q


r
------------------------------------
  0.  p
  1.  q

2 subgoals
:
   proof

> elt (VALIDATE_LT (USE_SG_THEN ACCEPT_TAC 2 1)) ;
Exception- OK..
HOL_ERR at Tactic.ACCEPT_TAC: raised
```

Comments

Some users may expect the generated tactic to be `ttac (A |- u)`, rather than `ttac (u |- u)`.

using (bossLib)

```
bossLib.using : tactic * thm -> tactic
```

Specifies alternative theorem to use for given tactic

The standard HOL environment has `using` an infix, so one writes `tac using thm`. Such a call stashes an encoding of `thm`'s name onto the goal's assumption list and then calls `tac`. If `tac` is aware of the possibility, it can use this theorem instead of the theorem it would usually consult. After `tac` completes, the implementation of `using` removes the reference.

This is typically used with the tactics `Induct`, `Induct_on`, `Cases`, or `Cases_on` which consult the TypeBase to find the theorems their underlying code requires.

Failure

Fails if the specified theorem has no hypotheses, is polymorphic, and cannot be found by reverse lookup in the theorem database (using `DB.revlookup`). Also fails if the underlying tactic fails.

Example

```
Induct_on ‘l’ using SNOC_INDUCT
```

sets up an induction on the term "`l`" using the `SNOC_INDUCT` principle ("structural induction from the back of the list").

Comments

Tactics unaware of the possibility of the presence of augmented assumption lists can behave strangely.

See also

`bossLib.Induct_on`, `markerSyntax.MK_USING`

VALID (Tactical)

```
Tactical.VALID : tactic -> tactic
```

Makes a tactic fail if it would otherwise return an invalid proof.

If tac applied to the goal (asl,g) produces a justification that does not create a theorem A |- g, with A a subset of asl, then VALID tac (asl,g) fails (raises an exception). If tac produces a valid proof on the goal, then the behaviour of VALID tac (asl,g) is the same as tac (asl,g)

Failure

Fails by design if tac produces an invalid proof when applied to a goal. Also fails if tac fails when applied to the given goal.

See also

proofManagerLib.expand, Tactical.VALIDATE

VALID_LT (Tactical)

```
Tactical.VALID_LT : list_tactic -> list_tactic
```

Makes a list-tactic fail if it would otherwise return an invalid proof.

When list-tactic ltac is applied to a goal list gl it produces a new goal list gl' and a justification. When the justification is applied to a list thl' of theorems which are the new goals gl', proved, it should produce a list thl of theorems which are the goals gl, proved.

Precisely, for each goal (asl, g) in gl, the corresponding theorem in thl should be A |- g, with A a subset of asl. If this is not the case, then the list-tactic is invalid, and VALID_LT ltac gl fails (raises an exception). Otherwise, VALID_LT ltac gl behaves the same as ltac gl.

Failure

VALID_LT ltac gl fails by design if ltac gl produces new goals and justification which do not prove the given goals gl. Also fails if its ltac gl fails.

See also

`Tactical.VALID`, `Tactical.VALIDATE_LT`, `proofManagerLib.elt`, `proofManagerLib.expand_list`

VALIDATE (Tactical)

```
Tactical.VALIDATE : tactic -> tactic
```

Makes a tactic valid if its invalidity is due to relying on assumptions not present in the goal.

Suppose `tac` applied to the goal `(asl,g)` produces a justification that creates a theorem `A |- g'`. If `A` a not a subset of `asl`, then the tactic is invalid (and `VALID tac (asl,g)` fails, ie, raises an exception). But `VALIDATE tac (asl,g)` produces a subgoal list augmented by the members of `A` missing from `asl`.

If `g'` differs from `g`, both `VALID tac (asl,g)` and `VALIDATE tac (asl,g)` fail.

Failure

Fails by design if `tac`, when applied to a goal, produces a proof which is invalid on account of proving a theorem whose conclusion differs from that of the goal.

Also fails if `tac` fails when applied to the given goal.

Example

For example, where theorem `uth'` is `[p'] |- q`

```
[...Lines elided...]
   5. Incomplete goalstack:
        Initial goal:
        ∃R. WF R ∧ (∀rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)) ∧
            ∀rst x ord. R (ord,FILTER ($¬ ∘ ord x) rst) (ord,x::rst)

   4. Incomplete goalstack:
        Initial goal:
```

```
        1 + 2 = 2 + 1

        Current goal:
        ∀(x,y). x + y = y + x

   3. Incomplete goalstack:
        Initial goal:
         0.  p ⇒ q
        ------------------------------------
            r

        Current goal:
        p

   2. Incomplete goalstack:
        Initial goal:
         0.  q
         1.  p
        ------------------------------------
            r

        Current goal:
         0.  q
         1.  p
        ------------------------------------
            p'

   1. Incomplete goalstack:
        Initial goal:
         0.  p
        ------------------------------------
            q


> e (ACCEPT_TAC uth') ;
Exception- OK..
HOL_ERR (at Tactical.VALID: Invalid tactic: theorem has bad hypothesis p') raised

> e (VALIDATE (ACCEPT_TAC uth')) ;
OK..
```

```
1 subgoal:
val it =

    0.  p
   ------------------------------------
        p'
```

Given a goal with an implication in the assumptions, one can split it into two subgoals.

```
[...Lines elided...]
   1. Incomplete goalstack:
        Initial goal:
         0.  p ⇒ q
        ------------------------------------
             r
```

```
> e (VALIDATE (POP_ASSUM (ASSUME_TAC o UNDISCH))) ;
OK..
2 subgoals:
val it =

    0.  q
   ------------------------------------
        r

    0.  p ⇒ q
   ------------------------------------
        p
```

Meanwhile, to propose a term, prove it as a subgoal and then use it to prove the goal, as is done using `SUBGOAL_THEN tm ASSUME_TAC`, can also be done by `VALIDATE (ASSUME_TAC (ASSUME tm))`

Where a tactic `tac` requires certain assumptions to be present in the goal, which are not present but are capable of being proved, `VALIDATE tac` will conveniently set up new subgoals to prove the missing assumptions.

See also

proofManagerLib.expand, Tactical.VALID, Tactical.GEN_VALIDATE, Tactical.ADD_SGS_TAC, Tactical.SUBGOAL_THEN

VALIDATE_LT (Tactical)

```
Tactical.VALIDATE_LT : list_tactic -> list_tactic
```

Makes a list-tactic valid if its invalidity is due to relying on assumptions not present in one of the goals.

When list-tactic `ltac` is applied to a goal list `gl` it produces a new goal list `gl'` and a justification. When the justification is applied to a list `thl'` of theorems which are the new goals `gl'`, proved, it should produce a list `thl` of theorems which are the goals `gl`, proved.

A list-tactic can be invalid due to proving a theorem whose conclusion differs from that of the corresponding goal, or due to proving a theorem which contains extra assumptions relative to the corresponding goal. In this latter case, `VALIDATE_LT ltac` makes the list-tactic valid by returning extra subgoals to prove those extra assumptions.

See `VALID_LT` for more details.

Failure

Fails by design if `ltac`, when applied to a goal list, produces a proof which is invalid on account of proving a theorem whose conclusion differs from that of the corresponding goal.

Also fails if `ltac` fails when applied to the given goals.

Example

Where `uthr'` is `[p', q] |- r` and `uths'` is `[p, q'] |- s`

```
OK..
2 subgoals:
val it =
```

```
    0.  p
    1.  q
   ------------------------------------
        s

    0.  p
    1.  q
   ------------------------------------
        r

> elt (ALLGOALS (FIRST (map ACCEPT_TAC [uthr', uths']))) ;
Exception- OK..
HOL_ERR
  (at Tactical.VALID_LT: Invalid list-tactic: theorem has bad hypothesis p') raised

> elt (VALIDATE_LT (ALLGOALS (FIRST (map ACCEPT_TAC [uthr', uths'])))) ;
OK..
2 subgoals:
val it =

    0.  p
    1.  q
   ------------------------------------
        q'

    0.  p
    1.  q
   ------------------------------------
        p'
```

Where a tactic `ltac` requires certain assumptions to be present in one of the goals, which are not present but are capable of being proved, `VALIDATE_LT ltac` will conveniently set up new subgoals to prove the missing assumptions.

See also

`Tactical.VALID`, `Tactical.VALID_LT`, `Tactical.VALIDATE`, `proofManagerLib.elt`, `proofManagerLib.expand_list`

var_compare (Term)

```
Term.var_compare : term * term -> order
```

Total ordering on variables.

An invocation `var_compare (v1,v2)` will return one of `{LESS, EQUAL, GREATER}`, according to an ordering on term variables. The ordering is transitive and total.

Failure

If `v1` and `v2` are not both variables.

Example

```
> var_compare (mk_var("x",bool), mk_var("x",bool --> bool));
val it = LESS: order
```

Comments

Used to build high performance datastructures for dealing with sets having many variables.

See also

`Term.empty_varset`, `Term.compare`

VAR_EQ_TAC (BasicProvers)

```
BasicProvers.VAR_EQ_TAC : tactic
```

Simplifies a goal using any assumption of the form `v = t` or `t = v`, where `v` is a variable

`VAR_EQ_TAC` simplifies the goal, including its assumptions, using one assumption of the form `v = t` or `t = v`, where `v` is a variable which is not contained in `t`.

In both cases, `v` is replaced throughout by `t`, and the relevant assumption is deleted.

Failure

`VAR_EQ_TAC` fails if there are no such assumptions

See also

`bossLib.FULL_SIMP_TAC`, `bossLib.ASM_SIMP_TAC`, `Rewrite.ASM_REWRITE_TAC`

var_occurs (Term)

```
Term.var_occurs : term -> term -> bool
```

Check if a variable occurs in free in a term.

An invocation `var_occurs v M` returns `true` just in case `v` occurs free in M.

Failure

If the first argument is not a variable.

Example

```
> var_occurs (Term`x:bool`) (Term `a /\ b ==> x`);
val it = true: bool

> var_occurs (Term`x:bool`) (Term `!x. a /\ b ==> x`);
val it = false: bool
```

Comments

Identical to `free_in`, except for the requirement that the first argument be a variable.

See also

`Term.free_vars`, `Term.free_in`

variant (Term)

```
Term.variant : term list -> term -> term
```

Modifies a variable name to avoid clashes.

When applied to a list of variables to avoid clashing with, and a variable to modify, `variant` returns a variant of the variable to modify, that is, it changes the name as intuitively as possible to make it distinct from any variables in the list, or any constants. This is done by adding primes to the name.

The exact form of the variable name should not be relied on, except that the original variable will be returned unmodified unless it is itself in the list to avoid clashing with, or if it is the name of a constant.

Failure

`variant l t` fails if any term in the list `l` is not a variable or if `t` is not a variable.

Example

The following shows a couple of typical cases:

```
> variant [“y:bool”, “z:bool”] “x:bool”;
val it = “x” : term

> variant [“x:bool”, “x':num”, “x'':num”] “x:bool”;
> val it = “x'''” : term
```

while the following shows that clashes with the names of constants are also avoided:

```
> variant [] (mk_var("T",bool));
val it = “T'” : term
```

The function `variant` is extremely useful for complicated derived rules which need to rename variables to avoid free variable capture while still making the role of the variable obvious to the user.

Comments

There is a `Term.numvariant` function that has the same type signature, but which varies by adding and then incrementing a numeric suffix to the variable's stem.

See also

`Term.genvar`, `Term.prim_variant`

version (Globals)

```
Globals.version : int
```

The version number of the `HOL` system being run.

Example

```
> Globals.version;
val it = 2: int
```

See also

`Globals.release`

view_struct (smlOpen)

```
smlOpen.view_struct : string -> (string list * string list * string list * string list)
```

Shows SML identifiers included in a structure (or substructure) `s`.

The first field contains values, the second exceptions (i.e Match), the third constructions (i.e SOME) and the fourth substructures. This output is computed by inspecting PolyML.globalNameSpace () in an external HOL4 environment.

Failure

Fails if the structure `s` cannot be loaded by a script in the directory `src/AI/sml_inspection/open`.

Example

```
> load "smlOpen"; open smlOpen;
val it = (): unit

> view_struct "Math";
val it =
   (["sin", "sinh", "cos", "tan", "cosh", "e", "asin", "tanh", "atan2", "ln",
     "acos", "log10", "pi", "sqrt", "atan", "exp", "pow"], [], [], []):
   string list * string list * string list * string list
> view_struct "HolKernel.Definition";
val it =
   (["new_definition", "new_specification", "located_new_type_definition",
     "located_new_definition", "gen_new_specification",
     "located_new_specification", "new_definition_hook",
     "new_type_definition", "located_gen_new_specification"], [], [], []):
   string list * string list * string list * string list
```

Comments

Including a Holmakefile in the directory `src/AI/sml_inspection/open` with the line `INCLUDES = path-to-your-structure` should remove the requirement for the structure to be in sigobj.

See also

`smlExecute.quse_string`, `smlExecScripts.exec_script`

W (Lib)

```
Lib.W : ('a -> 'a -> 'b) -> 'a -> 'b
```

Duplicates function argument : `W f x` equals `f x x`.

The `W` combinator can be understood as a planner: in the application `f x x`, the function `f` can scrutinize `x` and generate a function that then gets applied to `x`.

Failure

`W f` never fails. `W f x` fails if `f x` fails or if `f x x` fails.

Example

```
> load "tautLib";
val it = (): unit
> tautLib.TAUT_PROVE (Term `(a:bool = b) = (~a = ~b)`);
val it = ⊢ (a ⇔ b) ⇔ (¬a ⇔ ¬b): thm

> W (GENL o free_vars o concl) it;
val it = ⊢ ∀b a. (a ⇔ b) ⇔ (¬a ⇔ ¬b): thm
```

See also

`Lib.##`, `Lib.C`, `Lib.I`, `Lib.K`, `Lib.S`

WARNING_outstream (Feedback)

```
Feedback.WARNING_outstream : TextIO.outstream ref
```

Controlling output stream used when printing `HOL_WARNING`

The value of reference cell WARNING_outstream controls where HOL_WARNING prints its argument.

The default value of WARNING_outstream is a function that prints its argument to TextIO.stdOut.

Example

```
> val outputs = ref ([] : string list);
val outputs = ref []: string list ref

> WARNING_outstream := (fn s => outputs := !outputs @ [s]);
val it = (): unit

> HOL_WARNING "Module" "Function" "Sufferin' Succotash!";
val it = (): unit

> outputs;
val it = ref ["<<HOL warning: Module.Function: Sufferin' Succotash!>>\n"]:
   string list ref
```

See also

Feedback, Feedback.HOL_WARNING, Feedback.ERR_outstream, Feedback.MESG_outstream, Feedback.emit_WARNING

WARNING_to_string (Feedback)

```
Feedback.WARNING_to_string : (string -> string -> string -> string) ref
```

Alterable function for formatting HOL_WARNING.

WARNING_to_string is a reference to a function for formatting the argument to HOL_WARNING.

The default value of WARNING_to_string is format_WARNING.

Example

```
- fun alt_WARNING_report s t u =
    String.concat["WARNING---", s,".",t,": ",u,"---END WARNING\n"];

- WARNING_to_string := alt_WARNING_report;

- HOL_WARNING "Foo" "bar" "Look out";
WARNING---Foo.bar: Look out---END WARNING
> val it = () : unit
```

See also

`Feedback`, `Feedback.HOL_WARNING`, `Feedback.format_WARNING`, `Feedback.ERR_to_string`, `Feedback.MESG_to_string`

WEAKEN_CONSEQ_CONV_RULE (ConseqConv)

```
ConseqConv.WEAKEN_CONSEQ_CONV_RULE : (directed_conseq_conv -> thm -> thm)
```

Tries to weaken the conclusion of a theorem consisting of an implication.

Given a theorem of the form `|- A ==> C` and a directed consequence conversion `c` a call of `WEAKEN_CONSEQ_CONV_RULE c thm` tries to weaken `C` to a predicate `wC` using `c`. If it succeeds it returns the theorem `|- A ==> wC`.

See also

`ConseqConv.STRENGTHEN_CONSEQ_CONV_RULE`

WEAKEN_TAC (Tactic)

```
Tactic.WEAKEN_TAC : (term -> bool) -> tactic
```

Deletes assumption from goal.

Given an ML predicate `P` mapping terms to `true` or `false` and a goal `(asl,g)`, an invocation `WEAKEN_TAC P (asl,g)` removes the first element (call it `tm`) that `P` holds of from `asl`, returning the goal `(asl - tm,g)`.

Failure

Fails if the assumption list of the goal is empty, or if `P` holds of no element in `asl`.

Example

Suppose we want to dispose of the equality assumption in the following goal:

```
C x
------------------------------------
  0.  A = B
  1.  B x
```

The following application of `WEAKEN_TAC` does the job.

```
- e (WEAKEN_TAC is_eq);
OK..
1 subgoal:
> val it =
    C x
    ------------------------------------
      B x
```

Occasionally useful for getting rid of superfluous assumptions.

See also

`Tactical.PAT_ASSUM`, `Tactical.POP_ASSUM`

WF_REL_TAC (TotalDefn)

```
TotalDefn.WF_REL_TAC : term quotation -> tactic
```

Re-exported from `bossLib.WF_REL_TAC`. See that entry for full documentation.

WF_REL_TAC (bossLib)

```
bossLib.WF_REL_TAC : term quotation -> tactic
```

Also exported as `TotalDefn.WF_REL_TAC`.

Start termination proof.

`WF_REL_TAC` builds a tactic that starts a termination proof. An invocation `WF_REL_TAC q`, where `q` should parse into a term that denotes a well-founded relation, builds a tactic `tac` that is intended to be applied to a goal arising from an application of `tgoal` or `tprove`. Such a goal has the form

```
?R. WF R /\ ...
```

The tactic `tac` will instantiate `R` with the relation denoted by `q` and will attempt various simplifications of the goal. For example, it will try to automatically prove the well-foundedness of the relation denoted by `q`, and will also attempt to simplify the goal using some basic facts about well-founded relations. Often this can result in a much simpler goal.

Failure

`WF_REL_TAC q` fails if `q` does not parse into a term whose type is an instance of `'a -> 'a -> bool`.

Example

Suppose that a version of Quicksort had been defined as follows:

```
val qsort_defn =
     Hol_defn "qsort"
        `(qsort ___ [] = []) /\
```

```
            (qsort ord (x::rst) =
                APPEND (qsort ord (FILTER ($~ o ord x) rst))
                  (x :: qsort ord (FILTER (ord x) rst)))`;
```

Then one can start a termination proof as follows: set up a goalstack with `tgoal` and then apply `WF_REL_TAC` with a quotation denoting a suitable well-founded relation.

```
- tgoal qsort_defn;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
       Initial goal:
        ?R. WF R /\
         (!rst x ord. R (ord,FILTER ($~ o ord x) rst) (ord,x::rst)) /\
          !rst x ord. R (ord,FILTER (ord x) rst) (ord,x::rst)

- e (WF_REL_TAC `measure (LENGTH o SND)`);

OK..
2 subgoals:
> val it =
   !rst x ord. LENGTH (FILTER (ord x) rst) < LENGTH (x::rst)

   !rst x ord. LENGTH (FILTER (\x'. ~ord x x') rst) < LENGTH (x::rst)
```

Execution of `WF_REL_TAC` has automatically proved the well-foundedness of

```
measure (LENGTH o SND)
```

and the remainder of the goal has been simplified into a pair of easy goals.

Comments

There are two problems to deal with when trying to prove termination. First, one has to understand, intuitively and then mathematically, why the function under consideration terminates. Second, one must be able to phrase this in HOL. In the following, we shall give a few examples of how this is done.

There are a number of basic and advanced means of specifying well-founded relations. The most common starting point for dealing with termination problems for recursive

functions is to find some function, known as a 'measure' under which the arguments of a function call are larger than the arguments to any recursive calls that result.

For a very simple starter example, consider the following definition of a function that computes the greatest common divisor of two numbers:

```
- val gcd_defn = Hol_defn "gcd"
     `(gcd (0,n) = n) /\
      (gcd (m,n) = gcd (n MOD m, m))`;

- Defn.tgoal gcd_defn;

> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ?R. WF R /\ !v2 n. R (n MOD SUC v2,SUC v2) (SUC v2,n)
```

The recursion happens in the first argument, and the recursive call in that position is a smaller number. The way to phrase the termination of `gcd` in HOL is to use a 'measure' function to map from the domain of `gcd`—a pair of numbers—to a number. The definition of `measure` is equivalent to

```
measure f x y = (f x < f y).
```

(The actual definition of `measure` in `prim_recTheory` is more primitive.) Now we must pick out the argument position to measure and invoke `WF_REL_TAC`:

```
- e (WF_REL_TAC `measure FST`);
OK..

1 subgoal:
> val it =
 !v2 n. n MOD SUC v2 < SUC v2
```

This goal is easy to prove with a few simple arithmetic facts:

```
- e (PROVE_TAC [arithmeticTheory.DIVISION, prim_recTheory.LESS_0]);
OK..

Goal proved. ...
```

Sometimes one needs a measure function that is itself recursive. For example, consider a type of binary trees and a function that 'unbalances' trees. The algorithm works by rotating the tree until it gets a `Leaf` in the left branch, then it recurses into the right branch. At the end of execution the tree has been linearized.

```
- Hol_datatype
   `btree = Leaf
          | Brh of btree => btree`;

- val Unbal_defn =
   Hol_defn "Unbal"
   `(Unbal Leaf = Leaf)
/\  (Unbal (Brh Leaf bt) = Brh Leaf (Unbal bt))
/\  (Unbal (Brh (Brh bt1 bt2) bt) = Unbal (Brh bt1 (Brh bt2 bt)))`;

- Defn.tgoal Unbal_defn;

> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
       Initial goal:
        ?R. WF R /\
            (!bt. R bt (Brh Leaf bt)) /\
            !bt bt2 bt1. R (Brh bt1 (Brh bt2 bt)) (Brh (Brh bt1 bt2) bt)
```

Since the size of the tree is unchanged in the last clause in the definition of `Unbal`, a simple size measure will not work. Instead, we can assign weights to nodes in the tree such that the recursive calls of `Unbal` decrease the total weight in every case. One such assignment is

```
Weight (Leaf) = 0
Weight (Brh x y) = (2 * Weight x) + (Weight y) + 1
```

It is easiest to use `Define` to define `Weight`, but if one is worried about "polluting" the signature, one can also use `prove_rec_fn_exists` from the `Prim_rec` structure:

```
val Weight =
  Prim_rec.prove_rec_fn_exists (TypeBase.axiom_of ("", "btree"))
   (Term`(Weight (Leaf) = 0) /\
         (Weight (Brh x y) = (2 * Weight x) + (Weight y) + 1)`);
```

```
> val Weight =
   |- ?Weight.
         (Weight Leaf = 0) /\
         !x y. Weight (Brh x y) = 2 * Weight x + Weight y + 1 : thm

- e (STRIP_ASSUME_TAC Weight);
OK..

1 subgoal:
> val it =
    ?R.
      WF R /\ (!bt. R bt (Brh Leaf bt)) /\
      !bt bt2 bt1. R (Brh bt1 (Brh bt2 bt)) (Brh (Brh bt1 bt2) bt)
    ------------------------------------
      0.  Weight Leaf = 0
      1.  !x y. Weight (Brh x y) = 2 * Weight x + Weight y + 1
```

Now we can invoke `WF_REL_TAC`:

```
e (WF_REL_TAC `measure Weight`);
OK..

2 subgoals:
> val it =
 !bt bt2 bt1.
   Weight (Brh bt1 (Brh bt2 bt)) < Weight (Brh (Brh bt1 bt2) bt)
 ------------------------------------
   0.  Weight Leaf = 0
   1.  !x y. Weight (Brh x y) = 2 * Weight x + Weight y + 1

 !bt. Weight bt < Weight (Brh Leaf bt)
 ------------------------------------
   0.  Weight Leaf = 0
   1.  !x y. Weight (Brh x y) = 2 * Weight x + Weight y + 1
```

Both of these subgoals are quite easy to prove.

The technique of 'weighting' nodes in a tree in order to prove termination also goes by the name of 'polynomial interpretation'. It must be admitted that finding the correct

weighting for a termination proof is more an art than a science. Typically, one makes a guess and then tries the termination proof to see if it works.

Occasionally, there's a combination of factors that complicate the termination argument. For example, the following specification describes a naive pattern matching algorithm on strings (represented as lists here). The function takes four arguments: the first is the remainder of the pattern being matched. The second is the remainder of the string being searched. The third argument holds the original pattern to be matched. The fourth argument is the string being searched. If the pattern (first argument) becomes exhausted, then a match has been found and the function returns `T`. Otherwise, if the string being searched becomes exhausted, the function returns `F`.

```
val match0_defn =
  Hol_defn "match0"
       `(match0 [] __ __ __ = T)
   /\   (match0 __ [] __ __ = F)
   /\   (match0 (p::pp) (s::ss) p0 rs =
          if p=s then match0 pp ss p0 rs else
          if NULL rs then F
             else match0 p0 (TL rs) p0 (TL rs))`;

- val match = Define `match pat str = match0 pat str pat str`;
```

The remaining case is when there's more searching to do; the function checks if the head of the pattern is the same as the head of the string being searched. If yes, then we recursively search, using the tail of the pattern and the tail of the string being searched. If no, that means that we have failed to match the pattern, so we should move one character ahead in the string being searched and try again. If the string being searched is empty, however, then we return `F`. The second and third arguments both represent the string being searched. The second argument is a kind of 'local' version of the string being searched; we recurse into it as long as there are matches with the pattern. However, if the search eventually fails, then the fourth argument, which 'remembers' where the search started from, is used to restart the search.

So much for the behaviour of the function. Why does it terminate? There are two recursive calls. The first call reduces the size of the first and second arguments, and leaves the other arguments unchanged. The second call can increase the size of the first and second arguments, but reduces the size of the fourth.

This is a classic situation in which to use a lexicographic ordering: some arguments to the function are reduced in some recursive calls, and some others are reduced in other recursive calls. Recall that `LEX` is an infix operator, defined in `pairTheory` as follows:

```
LEX R1 R2 = \(x,y) (p,q). R1 x p \/ ((x=p) /\ R2 y q)
```

In the second recursive call, the length of `rs` is reduced, and in the first it stays the same. This motivates having the length of the fourth argument be the first component of the lexicographic combination, and the length of the second argument as the second component.

What we need now is to formalize this. We want to map from the four-tuple of arguments into a lexicographic combination of relations. This is enabled by `inv_image` from `relationTheory`:

```
inv_image R f = \x y. R (f x) (f y)
```

The actual relation maps from the four-tuple of arguments into a pair of numbers `(m,n)`, where `m` is the length of the fourth argument, and `n` is the length of the second argument. These lengths are then compared lexicographically with respect to less-than (<).

```
- Defn.tgoal match0_defn;

- e (WF_REL_TAC `inv_image ($< LEX $<)
                  (\(w,x,y,z). (LENGTH z, LENGTH x))`);

OK..
2 subgoals:
> val it =
 !rs ss s p.
   (p=s) ==> LENGTH rs < LENGTH rs \/ LENGTH ss < LENGTH (s::ss)


 !ss rs s p.
   ~(p = s) /\ ~NULL rs ==>
   LENGTH (TL rs) < LENGTH rs \/
   (LENGTH (TL rs) = LENGTH rs) /\ LENGTH (TL rs) < LENGTH (s::ss)
```

The first subgoal needs a case-split on `rs` before it is proved by rewriting, and the seconds is also easy to prove by rewriting.

As a final example, one occasionally needs to recurse over non-concrete data, such as finite sets or multisets. We can define a 'fold' function (of questionable utility) for finite sets as follows:

```
load "pred_setTheory"; open pred_setTheory;

val FOLD_SET_defn =
  Defn.Hol_defn "FOLD_SET"
  `FOLD_SET (s:'a->bool) (b:'b) =
     if FINITE s then
        if s={} then b
        else FOLD_SET (REST s) (f (CHOICE s) b)
     else ARB`;
```

Typically, such functions terminate because the cardinality of the set (or multiset) is reduced in the recursive call, and this is another application of `measure`:

```
val (FOLD_SET_0, FOLD_SET_IND) =
 Defn.tprove (FOLD_SET_defn,
   WF_REL_TAC `measure (CARD o FST)`
    THEN PROVE_TAC [CARD_PSUBSET, REST_PSUBSET]);
```

The desired recursion equation

```
|- FINITE s ==> (FOLD_SET f s b =
                  if s = {} then b
                  else FOLD_SET f (REST s) (f (CHOICE s) b))
```

is easy to obtain from `FOLD_SET_0`.

See also

`Defn.tgoal`, `Defn.tprove`, `bossLib.Hol_defn`

with_exn (Lib)

```
Lib.with_exn : ('a -> 'b) -> 'a -> exn -> 'b
```

Apply a function to an argument, raising supplied exception on failure.

An evaluation of `with_exn f x e` applies function `f` to argument `x`. If that computation finishes with `y`, then `y` is the result. Otherwise, `f x` raised an exception, and the exception `e` is raised instead. However, if `f x` raises the `Interrupt` exception, then `with_exn f x e` results in the `Interrupt` exception being raised.

Failure

When `f x` fails or is interrupted.

Example

```
> with_exn dest_comb (Term`\x. x /\ y`) (Fail "My kingdom for a horse");
Exception- Fail "My kingdom for a horse" raised

> with_exn (fn _ => raise Interrupt) 1 (Fail "My kingdom for a horse");
Exception- The type of (it) contains a free type variable. Setting it to a unique
   monotype.
Interrupt raised
```

Comments

Often `with_exn` can be used to clean up programming where lots of exceptions may be handled. For example, taking apart a compound term of a certain desired form may fail at several places, but a uniform error message is desired.

```
local val expected = mk_HOL_ERR "" "dest_quant" "expected !v.M or ?v.M"
in
fun dest_quant tm =
  let val (q,body) = with_exn dest_comb tm expected
      val (p as (v,M)) = with_exn dest_abs body expected
  in
     if q = universal orelse q = existential
       then p
       else raise expected
  end
end
```

See also

Feedback.wrap_exn, Lib.assert_exn, Lib.assert

with_flag (Lib)

```
Lib.with_flag : 'a ref * 'a -> ('b -> 'c) -> 'b -> 'c
```

Apply a function under a particular flag setting.

An invocation `with_flag (r,v) f x` sets the reference variable `r` to the value `v`, then evaluates `f x`, then resets `r` to its original value, and returns the value of `f x`.

Failure

Fails if `f x` fails. In that case, `r` is reset to its original value before raising the exception from `f x`.

Example

```
> fun print_term_nl tm = (print_term tm; Feedback.HOL_INFO "\n");
val print_term_nl = fn: term -> unit

> with_flag (show_types, true) print_term_nl (concl T_DEF);
T ⇔ (λ(x :bool). x) = (λ(x :bool). x)

val it = (): unit

> print_term_nl (concl T_DEF);
T ⇔ (λx. x) = (λx. x)

val it = (): unit
```

See also

Feedback.traces, Feedback.register_btrace, Feedback.trace, Lib.time

with_traces (Feedback)

```
Feedback.with_traces : (string * int) list -> ('a -> 'b) -> 'a -> 'b
```

Invoke a function with specified levels for traces.

The `with_traces` function is used to set values of a collection of tracing variables for the duration of one top-level function call.

In a state where trace variables designated by `nm_1,...,nm_k` have corresponding values `i_1,...,i_k`, a call

```
with_traces [(nm_1,j_1),...,(nm_k,j_k)] f x
```

first sets the trace variables designated by `nm_1,...,nm_k` to the corresponding values `j_1,...,j_k`. Then the call evaluates `f x` and returns the result, but only after restoring the original `i_1,...,i_k` values for the trace variables.

Failure

Fails if any of the designations fails to be associated with a registered tracing variable. Also fails if an attempt is made to set a trace variable outside of its specified range of values. Also fails if the function invocation fails.

Example

First we define a convenience function.

```
> fun pr_thm th = Feedback.HOL_INFO (thm_to_string th ^ "\n");
val pr_thm = fn: thm -> unit
```

Now we examine a theorem with and without traces set to local values.

```
> map current_trace ["assumptions", "types"] ;
val it = [0, 0]: int list

> pr_thm EQ_SYM_EQ;
```

```
⊢ ∀x y. x = y ⇔ y = x
val it = (): unit

> with_traces [("assumptions",1), ("types",1)] pr_thm EQ_SYM_EQ;
 [] ⊢ ∀(x :α) (y :α). x = y ⇔ y = x
val it = (): unit

> map current_trace ["assumptions", "types"] ;
val it = [0, 0]: int list
```

See also

`Feedback`, `Feedback.traces`, `Feedback.register_trace`, `Feedback.reset_trace`, `Feedback.reset_traces`, `Feedback.set_trace`, `Feedback.trace`, `Lib.with_flag`

wlog_tac (bossLib)

```
bossLib.wlog_tac : term quotation -> term quotation list -> tactic
```

Also exported as `wlogLib.wlog_tac`.

Enrich the hypotheses with a proposition that can be assumed without loss of generality.

The user provides term quotations that parse to a proposition `P` and a list of variables. Typically there are 2 subgoals. The first subgoal is to prove that the general case of the original goal follows from the specific case where `P` holds; the second subgoal is the original goal with `P` added to the assumptions. The first subgoal is always present, and the subgoals (if any) produced by `strip_assume_tac P |- P` follows.

If the goal is `hyp ?- t` then the first subgoal is `hyp, !vars. ant ==> t, ~P ?- t` where `ant` is the conjunction of `P` and those hypotheses of the original subgoal where any variable in the user-provided list occurs free. The universal quantification is over the variables in the user-provided list plus any variable that appears free in `P` or `t` and is not a local constant. For convenience `~P` is always added to the assumptions in the first subgoal because the case for `P` follows immediately from the hypothesis. Passing a non-empty list of variables allows to quantify over local constants in the hypothesis `!vars. ant ==> t`.

Detailed description: Given `wlog_tac q vars_q` let `asm ?- c` be the the goal. `q` is parsed in the goal context to a proposition `P`. `vars_q` are parsed to variables in the goal

context. Let `efv` (effectively free variables) be the free variables of `P` and `c` that are not free in the assumptions and are not in `vars` from left to right and first `P`, then `c`. Let `gen_vars` be `vars @ efv`. Let `asm'` be the elements of `asm` in which any of `vars` is a free variable. Let `ant` be the result of splicing `p :: asm'`. The first subgoal is `asm`, `(!(gen_vars). ant ==> c)`, `~P ?- c`. The proposition `P` is added to the assumptions with `strip_assume_tac`. If this generates subgoals (as is usually the case), then those subgoals follow.

A typical use case is to continue the proof assuming one case where all cases are symmetric. The first subgoal is a good candidate to be solved by a first order prover like `PROVE_TAC` or `METIS_TAC` providing to it the appropriate symmetry theorems.

Example

In the following examples assume `arithmeticTheory` is open.

```
> g(`ABS_DIFF x y + ABS_DIFF y z <= ABS_DIFF x z`);
val it =
   Proof manager status: 3 proofs.
   3. Incomplete goalstack:
        Initial goal:
         0.  p ⇒ q
        ------------------------------------
             r

        Current goal:
         0.  p ⇒ q
        ------------------------------------
             p

   2. Incomplete goalstack:
        Initial goal:
        p ∧ q ⇒ r ∧ s

        Current goal:
         0.  p
         1.  q
        ------------------------------------
             p'
```

```
   1. Incomplete goalstack:
        Initial goal:
        ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z
> e(wlog_tac `x <= z` []);
OK..
2 subgoals:
val it =

    0.  x ≤ z
   ------------------------------------
        ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z

    0.  ∀x z y. x ≤ z ⇒ ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z
    1.  ¬(x ≤ z)
   ------------------------------------
        ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z
```

The first subgoal can be solved by `prove_tac [ABS_DIFF_SYM, LESS_EQ_CASES, ADD_-COMM]`.

```
> g`MAX x y <= z <=> x <= z /\ y <= z`
val it =
   Proof manager status: 4 proofs.
   4. Incomplete goalstack:
        Initial goal:
         0.  p ⇒ q
        ------------------------------------
            r

        Current goal:
         0.  p ⇒ q
        ------------------------------------
            p

   3. Incomplete goalstack:
        Initial goal:
        p ∧ q ⇒ r ∧ s

        Current goal:
         0.  p
```

```
         1.  q
        ------------------------------------
            p'

   2. Incomplete goalstack:
        Initial goal:
        ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z

        Current goal:
         0.  ∀x z y. x ≤ z ⇒ ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z
         1.  ¬(x ≤ z)
        ------------------------------------
            ABS_DIFF x y + ABS_DIFF y z ≤ ABS_DIFF x z

   1. Incomplete goalstack:
        Initial goal:
        MAX x y ≤ z ⇔ x ≤ z ∧ y ≤ z
> e(wlog_tac `x <= y` []);
OK..
2 subgoals:
val it =

    0.  x ≤ y
   ------------------------------------
        MAX x y ≤ z ⇔ x ≤ z ∧ y ≤ z

    0.  ∀x y z. x ≤ y ⇒ (MAX x y ≤ z ⇔ x ≤ z ∧ y ≤ z)
    1.  ¬(x ≤ y)
   ------------------------------------
        MAX x y ≤ z ⇔ x ≤ z ∧ y ≤ z
```

The first subgoal can be solved by `prove_tac [LESS_EQ_CASES, MAX_COMM];`

Failure

Never fails.

See also

`bossLib.wlog_then`

wlog_tac (wlogLib)

```
wlogLib.wlog_tac : term quotation -> term quotation list -> tactic
```

Re-exported from `bossLib.wlog_tac`. See that entry for full documentation.

wlog_then (bossLib)

```
bossLib.wlog_then : term quotation -> term quotation list -> thm_tactic -> tactic
```

Also exported as `wlogLib.wlog_then`.

Apply a theorem-tactic using a proposition that can be assumed without loss of generality.

Like `wlog_tac`, but the theorem `P |- P` is passed to the user-provided theorem-tactic instead of `strip_assume_tac`.

Failure

Never fails when applied to a theorem-tactical. The resulting tactic fails if and only if the user-provided theorem-tactical fails when used as a tactic (i.e.: when applied to a theorem and a goal).

See also

`bossLib.wlog_tac`

wlog_then (wlogLib)

```
wlogLib.wlog_then : term quotation -> term quotation list -> thm_tactic -> tactic
```

Re-exported from `bossLib.wlog_then`. See that entry for full documentation.

WORD_ARITH_CONV (wordsLib)

```
wordsLib.WORD_ARITH_CONV : conv
```

Conversion based on WORD_ARITH_ss and WORD_ARITH_EQ_ss.

The conversion WORD_ARITH_CONV converts word arithmetic expressions into a canonical form.

Example

WORD_ARITH_CONV fixes the sign of equalities.

```
> SIMP_CONV (std_ss++wordsLib.WORD_ARITH_ss++wordsLib.WORD_ARITH_EQ_ss) [] “$- a = b : '
Exception- HOL_ERR
  (at Preterm.type-analysis: at line 1, character 83:

Type error in function application.
  Function: $= ($- a) :(α word -> α word) -> bool
  Argument: b :α word
  Reason: Attempt to unify different type operators: min$fun and fcp$cart
) raised

> wordsLib.WORD_ARITH_CONV “$- a = b : 'a word”
Exception- HOL_ERR
  (at Preterm.type-analysis: at line 1, character 35:

Type error in function application.
  Function: $= ($- a) :(α word -> α word) -> bool
  Argument: b :α word
  Reason: Attempt to unify different type operators: min$fun and fcp$cart
) raised
```

Comments

The fragment WORD_ARITH_EQ_ss and conversion WORD_CONV do not adjust the sign of equalities.

See also

wordsLib.WORD_ARITH_ss, wordsLib.WORD_ARITH_EQ_ss, wordsLib.WORD_LOGIC_CONV, wordsLib.WORD_MUL_LSL_CONV, wordsLib.WORD_CONV, wordsLib.WORD_BIT_EQ_CONV, wordsLib.WORD_EVAL_CONV

WORD_ARITH_EQ_ss (wordsLib)

wordsLib.WORD_ARITH_EQ_ss : ssfrag

Simplification fragment for words.

The fragment WORD_ARITH_EQ_ss simplifies "a = b : 'a word" to "a - b = 0w". It also simplifies using the theorems WORD_LEFT_ADD_DISTRIB, WORD_RIGHT_ADD_DISTRIB, WORD_MUL_LSL and WORD_NOT. When combined with wordsLib.WORD_ARITH_ss this fragment can be used to test for the arithmetic equality of words.

Example

```
> SIMP_CONV (pure_ss++wordsLib.WORD_ARITH_ss++wordsLib.WORD_ARITH_EQ_ss) [] “3w * (a
val it = ⊢ 3w * (a + b) = b + 3w * a ⇔ 2w * b = 0w: thm
```

Comments

This fragment is not included in WORDS_ss.

See also

wordsLib.WORD_ARITH_CONV, fcpLib.FCP_ss, wordsLib.BIT_ss, wordsLib.SIZES_ss, wordsLib.WORD_ARITH_ss, wordsLib.WORD_LOGIC_ss, wordsLib.WORD_SHIFT_ss, wordsLib.WORD_BIT_EQ_ss, wordsLib.WORD_EXTRACT_ss, wordsLib.WORD_MUL_LSL_ss, wordsLib.WORD_ss

WORD_ARITH_ss (wordsLib)

```
wordsLib.WORD_ARITH_ss : ssfrag
```

Simplification fragment for words.

The fragment `WORD_ARITH_ss` does AC simplification for word multiplication, addition and subtraction. It also simplifies `INT_MINw`, `INT_MAXw` and `UINT_MAXw`. If the word length is known then further simplification may occur, in particular for `$- (n2w n)` and `w2n (n2w n)`.

Example

```
> SIMP_CONV (pure_ss++wordsLib.WORD_ARITH_ss) [] “3w * b + a + 2w * b - a * 4w”
val it = ⊢ 3w * b + a + 2w * b − a * 4w = -3w * a + 5w * b: thm

> SIMP_CONV (pure_ss++wordsLib.WORD_ARITH_ss) [] “INT_MINw + INT_MAXw + UINT_MAXw”
val it = ⊢ INT_MINw + INT_MAXw + UINT_MAXw = -2w: thm
```

More simplification occurs when the word length is known.

```
> SIMP_CONV (pure_ss++wordsLib.WORD_ARITH_ss) [] “3w * b + a + 2w * b - a * 4w:word2”
val it = ⊢ 3w * b + a + 2w * b − a * 4w = a + b: thm

> SIMP_CONV (pure_ss++wordsLib.WORD_ARITH_ss) [] “w2n (33w:word4)”;
val it = ⊢ w2n 33w = 1: thm
```

Comments

Any term of value `UINT_MAXw` simplifies to `$- 1w` even when the word length is known - this helps when simplifying bitwise operations. If the word length is not known then `INT_MAXw` becomes `INT_MINw + $- 1w`.

See also

wordsLib.WORD_ARITH_CONV, wordsLib.WORD_CONV, fcpLib.FCP_ss, wordsLib.BIT_ss, wordsLib.SIZES_ss, wordsLib.WORD_LOGIC_ss, wordsLib.WORD_SHIFT_ss, wordsLib.WORD_ARITH_EQ_ss, wordsLib.WORD_BIT_EQ_ss, wordsLib.WORD_EXTRACT_ss, wordsLib.WORD_MUL_LSL_ss, wordsLib.WORD_ss

WORD_BIT_EQ_CONV (wordsLib)

```
wordsLib.WORD_BIT_EQ_CONV : conv
```

Conversion based on WORD_BIT_EQ_ss.

The conversion WORD_BIT_EQ_CONV performs simplification using fcpLib.FCP_ss.

Example

```
> wordsLib.WORD_BIT_EQ_CONV “a << 2 >>> 1 = ((5 -- 0) a << 1) :word8”
val it = ⊢ a ≪ 2 ⋙ 1 = (5 -- 0) a ≪ 1 ⇔ T: thm
```

See also

wordsLib.WORD_BIT_EQ_ss, blastLib.BBLAST_CONV, wordsLib.WORD_ARITH_CONV, wordsLib.WORD_LOGIC_CONV, wordsLib.WORD_MUL_LSL_CONV, wordsLib.WORD_CONV, wordsLib.WORD_EVAL_CONV

WORD_BIT_EQ_ss (wordsLib)

```
wordsLib.WORD_BIT_EQ_ss : ssfrag
```

Simplification fragment for words.

The fragment WORD_BIT_EQ_ss simplifies using fcpLib.FCP_ss and the definitions of “bitwise” operations, e.g., conjunction, disjunction, 1’s complement, shifts, concatenation

and sub-word extraction. Can be used in combination with decision procedures to test for the bitwise equality of words.

Example

```
> SIMP_CONV (std_ss++wordsLib.WORD_BIT_EQ_ss) [] “a = b : 'a word”
val it = ⊢ a = b ⇔ ∀i. i < dimindex (:α) ⇒ (a ' i ⇔ b ' i): thm
```

Further simplification occurs when the word length is known.

```
> SIMP_CONV (std_ss++wordsLib.WORD_BIT_EQ_ss) [] “a = b : word2”
val it = ⊢ a = b ⇔ (a ' 1 ⇔ b ' 1) ∧ (a ' 0 ⇔ b ' 0): thm
```

Best used in combination with decision procedures.

```
> (SIMP_CONV (std_ss++wordsLib.WORD_BIT_EQ_ss) [] THENC tautLib.TAUT_CONV) “a && b && a
val it = ⊢ a && b && a = a && b ⇔ T: thm
```

Comments

This fragment is not included in `WORDS_ss`.

See also

`wordsLib.WORD_BIT_EQ_CONV`, `fcpLib.FCP_ss`, `blastLib.BBLAST_CONV`, `wordsLib.BIT_-ss`, `wordsLib.SIZES_ss`, `wordsLib.WORD_ARITH_ss`, `wordsLib.WORD_LOGIC_ss`, `wordsLib.WORD_-SHIFT_ss`, `wordsLib.WORD_ARITH_EQ_ss`, `wordsLib.WORD_EXTRACT_ss`, `wordsLib.WORD_-MUL_LSL_ss`, `wordsLib.WORD_ss`

WORD_CANCEL_CONV (wordsLib)

```
wordsLib.WORD_CANCEL_CONV : conv
```

Conversion that forms the basis of `wordsLib.WORD_CANCEL_ss`.

See also

wordsLib.WORD_CANCEL_ss

WORD_CANCEL_ss (wordsLib)

wordsLib.WORD_CANCEL_ss : ssfrag

Simplification fragment for words.

The fragment WORD_CANCEL_ss helps simplify linear equations over bit-vectors. This fragment is designed to work in concert with wordsLib.WORD_ARITH_ss. The procedure will endeavour to ensure that all coefficients appear in positive form. Furthermore, the leftmost coefficient will be highest on the left-hand side of equations.

Example

```
> SIMP_CONV (pure_ss++wordsLib.WORD_ARITH_ss++wordsLib.WORD_CANCEL_ss) []
  “-3w * b + a = -2w * a + b: word32”;
val it = ⊢ -3w * b + a = -2w * a + b ⇔ 4w * b = 3w * a: thm
```

Comments

This fragment can be viewed as a superior version of wordsLib.WORD_ARITH_EQ_ss.

See also

wordsLib.WORD_ARITH_ss, wordsLib.WORD_ARITH_EQ_ss

WORD_CONV (wordsLib)

wordsLib.WORD_CONV : conv

Conversion for words.

The conversion `WORD_CONV` applies the simpset fragment `WORD_ss`.

Example

```
> wordsLib.WORD_CONV “c * (a + b) !! (b + a) * c”
val it = ⊢ c * (a + b) || (b + a) * c = a * c + b * c: thm
```

See also

`wordsLib.WORD_ss`, `wordsLib.WORD_ARITH_CONV`, `wordsLib.WORD_LOGIC_CONV`, `wordsLib.WORD_MUL_LSL_CONV`, `wordsLib.WORD_BIT_EQ_CONV`, `wordsLib.WORD_EVAL_CONV`

WORD_DECIDE (wordsLib)

```
wordsLib.WORD_DECIDE : conv
```

A decision procedure for words.

The conversion `WORD_DECIDE` is the same as `WORD_DP WORD_CONV bossLib.DECIDE`.

Example

```
> wordsLib.WORD_DECIDE “a && (b !! a) = a !! a && b”
val it = ⊢ a && (b || a) = a || a && b: thm

> wordsLib.WORD_DECIDE “a + 2w <+ 4w = a <+ 2w \/ 13w <+ a :word4”
Exception- Don't expect to find a = in this position after a <+
at line 1, character 37 and on line 1, characters 31-32.
HOL_ERR
  (at Absyn.Absyn: on line 1, characters 31-32:
       Don't expect to find a = in this position after a <+
at line 1, character 37 and on line 1, characters 31-32.
) raised
```

```
> wordsLib.WORD_DECIDE “a < 0w = 1w <+ a : word2”
Exception- Don't expect to find a = in this position after a <
at line 1, character 31 and at line 1, character 26.
HOL_ERR
  (at Absyn.Absyn: at line 1, character 26:
       Don't expect to find a = in this position after a <
at line 1, character 31 and at line 1, character 26.
) raised

> wordsLib.WORD_DECIDE “(?w:word4. 14w <+ w) /\ ~(?w:word4. 15w <+ w)”
val it = ⊢ (∃w. 14w <+ w) ∧ ¬∃w. 15w <+ w: thm
```

See also

wordsLib.WORD_DP

WORD_DECIDE_TAC (wordsLib)

```
wordsLib.WORD_DECIDE_TAC : tactic
```

A decision procedure tactic for words.

WORD_DECIDE_TAC is a tactical verion of WORD_DECIDE.

Failure

As for WORD_DECIDE.

See also

wordsLib.WORD_DECIDE

WORD_DIV_LSR_CONV (wordsLib)

```
wordsLib.WORD_DIV_LSR_CONV : conv
```

The conversion `WORD_DIV_LSR_CONV` replaces instances of unsigned division by a power of two with applications of logical right-shift.

Example

```
> wordsLib.WORD_DIV_LSR_CONV “w // 8w : word8”;
val it = ⊢ w // 8w = w ⋙ 3: thm
```

Comments

This conversion requires the word length to be known.

See also

`wordsLib.WORD_MUL_LSL_CONV`, `wordsLib.WORD_MOD_BITS_CONV`

WORD_DP (wordsLib)

```
wordsLib.WORD_DP : conv -> conv -> conv
```

Constructs a decision procedure for words.

The conversion `WORD_DP conv dp` is a decision procedure for words that makes use of the supplied conversion `conv` and decision procedure `dp`. Suitable decision procedures include `tautLib.TAUT_PROVE`, `bossLib.DECIDE`, `intLib.ARITH_PROVE` and `intLib.COOPER_PROVE`. The procedure will first apply `conv` and then `WORD_BIT_EQ_CONV`. If this is not sufficient then an attempt is made to solve the problem by applying an arithmetic decision procedure `dp`, e.g. “`(a = 0w) \/ (a = 1w :1 word)`” is mapped to the goal “`w2n a < 2 ==> (w2n a = 0) \/ (w2n a = 1)`”.

Failure

The invocation will fail when the decision procedure `dp` fails.

Example

```
> wordsLib.WORD_DP ALL_CONV tautLib.TAUT_PROVE “a && b && a = a && b”
val it = ⊢ a && b && a = a && b: thm

> wordsLib.WORD_DP ALL_CONV DECIDE “a < b /\ b < c ==> a < c : 'a word”
val it = ⊢ a < b ∧ b < c ⇒ a < c: thm

> wordsLib.WORD_DP ALL_CONV intLib.ARITH_PROVE “a <+ 3w:word16 ==> (a = 0w) \/ (a =
val it = ⊢ a <+ 3w ⇒ a = 0w ∨ a = 1w ∨ a = 2w: thm
```

Comments

On large problems `intLib.ARITH_PROVE` will perform much better than `bossLib.DECIDE`.

See also

`wordsLib.WORD_BIT_EQ_CONV`, `wordsLib.WORD_DECIDE`

WORD_EVAL_CONV (wordsLib)

```
wordsLib.WORD_EVAL_CONV : conv
```

Evaluation for words.

The conversion `WORD_EVAL_CONV` provides efficient evaluation for word operations. It uses `wordsLib.words_compset`.

Example

```
> wordsLib.WORD_EVAL_CONV “word_log2 (word_reverse (3w * (33w #<< 4))) : word32”
val it = ⊢ word_log2 (word_reverse (3w * 33w ⇆ 4)) = 27w: thm
```

Comments

This conversion is best suited to evaluating ground terms with known word lengths. The conversion `wordsLib.WORD_CONV` is a suitable alternative.

See also

`bossLib.EVAL`, `computeLib.CBV_CONV`, `wordsLib.WORD_LOGIC_CONV`, `wordsLib.WORD_MUL_LSL_CONV`, `wordsLib.WORD_CONV`, `wordsLib.WORD_BIT_EQ_CONV`, `wordsLib.WORD_EVAL_CONV`

WORD_EXTRACT_ss (wordsLib)

```
wordsLib.WORD_EXTRACT_ss : ssfrag
```

Simplification fragment for words.

The fragment `WORD_EXTRACT_ss` simplifies the operations `w2w`, `sw2sw` (signed word-to-word conversion), `word_lsb`, `word_msb`, `word_bit`, `>>` (arithmetic right shift), `>>>` (logical right shift), `#>>` (rotate right), `#<<` (rotate left), `@@` (concatenation), `--` (word bits) and `''` (word slice). The result is expressed in terms of `!!` (disjunction), `<<` (left shift) and `><` (word extract).

Example

```
> SIMP_CONV (std_ss++wordsLib.WORD_ss++wordsLib.WORD_EXTRACT_ss) [] “(((7 >< 5) (a:word8
val it = ⊢ (7 >< 5) a @@ (4 >< 0) a = a: thm

> SIMP_CONV (std_ss++wordsLib.WORD_ss++wordsLib.WORD_EXTRACT_ss) [] “(4 -- 2) ((a:word8)
val it = ⊢ (4 -- 2) (a ⇄ 4) = (0 >< 0) a ≪ 2 ‖ (7 >< 6) a: thm

> SIMP_CONV (std_ss++wordsLib.WORD_ss++wordsLib.WORD_EXTRACT_ss) [] “w2w (sw2sw (a:word4
val it = ⊢ w2w (sw2sw a) = a: thm
```

Comments

Best used in combination with WORD_ss.

See also

fcpLib.FCP_ss, wordsLib.BIT_ss, wordsLib.SIZES_ss, wordsLib.WORD_ARITH_ss, wordsLib.WORD_LOGIC_ss, wordsLib.WORD_ARITH_EQ_ss, wordsLib.WORD_BIT_EQ_ss, wordsLib.WORD_SHIFT_ss, wordsLib.WORD_MUL_LSL_ss, wordsLib.WORD_ss

WORD_LOGIC_CONV (wordsLib)

wordsLib.WORD_LOGIC_CONV : conv

Conversion based on WORD_LOGIC_ss.

The conversion WORD_LOGIC_CONV converts word logic expressions into a canonical form.

Example

```
> wordsLib.WORD_LOGIC_CONV “a && (b !! ~a !! c)”
val it = ⊢ a && (b ‖ ¬a ‖ c) = a && b ‖ a && c: thm
```

See also

wordsLib.WORD_LOGIC_ss, wordsLib.WORD_ARITH_CONV, wordsLib.WORD_MUL_LSL_CONV, wordsLib.WORD_CONV, wordsLib.WORD_BIT_EQ_CONV, wordsLib.WORD_EVAL_CONV

WORD_LOGIC_ss (wordsLib)

wordsLib.WORD_LOGIC_ss : ssfrag

Simplification fragment for words.

The fragment `WORD_LOGIC_ss` does AC simplification for word conjunction, disjunction and 1's complement (negation). If the word length is known then further simplification occurs, in particular for `~(n2w n)`.

Example

```
> SIMP_CONV (pure_ss++wordsLib.WORD_LOGIC_ss) [] “3w !! 12w && a !! ~4w !! a && 16w”
val it = ⊢ 3w ‖ 12w && a ‖ ¬4w ‖ a && 16w = 28w && a ‖ -5w: thm
```

More simplification occurs when the word length is known.

```
> SIMP_CONV (pure_ss++wordsLib.WORD_LOGIC_ss) [] “~12w !! ~14w : word8”
val it = ⊢ ¬12w ‖ ¬14w = 243w: thm
```

Comments

The term `$- 1w` represents `UINT_MAXw`, which is the supremum in bitwise operations.

See also

`wordsLib.WORD_LOGIC_CONV`, `wordsLib.WORD_CONV`, `fcpLib.FCP_ss`, `wordsLib.BIT_ss`, `wordsLib.SIZES_ss`, `wordsLib.WORD_ARITH_ss`, `wordsLib.WORD_SHIFT_ss`, `wordsLib.WORD_-ARITH_EQ_ss`, `wordsLib.WORD_BIT_EQ_ss`, `wordsLib.WORD_EXTRACT_ss`, `wordsLib.WORD_-MUL_LSL_ss`, `wordsLib.WORD_ss`

WORD_MOD_BITS_CONV (wordsLib)

```
wordsLib.WORD_MOD_BITS_CONV : conv
```

The conversion `WORD_MOD_BITS_CONV` replaces instances of `word_mod` by a power of two with applications of `word_bits`.

Example

```
> wordsLib.WORD_MOD_BITS_CONV “word_mod w 8w : word16”;
val it = ⊢ word_mod w 8w = (2 -- 0) w: thm
```

Comments

This conversion requires the word length to be known.

See also

wordsLib.WORD_DIV_LSR_CONV, wordsLib.BITS_INTRO_CONV

WORD_MUL_LSL_CONV (wordsLib)

```
wordsLib.WORD_MUL_LSL_CONV : conv
```

Conversion based on WORD_MUL_LSL_ss.

The conversion WORD_MUL_LSL_CONV converts a multiplication by a word literal into a sum of left shifts.

Example

```
> wordsLib.WORD_MUL_LSL_CONV “49w * a”
val it = ⊢ 49w * a = a ≪ 5 + a ≪ 4 + a: thm
```

See also

wordsLib.WORD_MUL_LSL_ss, wordsLib.WORD_DIV_LSR_CONV, wordsLib.WORD_ARITH_-CONV, wordsLib.WORD_LOGIC_CONV, wordsLib.WORD_CONV, wordsLib.WORD_BIT_EQ_-CONV, wordsLib.WORD_EVAL_CONV

WORD_MUL_LSL_ss (wordsLib)

```
wordsLib.WORD_MUL_LSL_ss : ssfrag
```

Simplification fragment for words.

The fragment `WORD_MUL_LSL_ss` simplifies a multiplication by a word literal into a sum of left shifts.

Example

```
> SIMP_CONV (std_ss++wordsLib.WORD_MUL_LSL_ss) [] “49w * a”
val it = ⊢ 49w * a = a ≪ 5 + a ≪ 4 + a: thm

> SIMP_CONV (std_ss++wordsLib.WORD_ss++wordsLib.WORD_MUL_LSL_ss) [] “2w * a + a << 1”
val it = ⊢ 2w * a + a ≪ 1 = a ≪ 2: thm
```

Comments

This fragment is not included in `WORDS_ss`. It should not be used in combination with `WORD_ARITH_EQ_ss` or `wordsLib.WORD_ARITH_CONV`, since these convert left shifts into multiplications.

See also

`wordsLib.WORD_MUL_LSL_CONV`, `fcpLib.FCP_ss`, `wordsLib.BIT_ss`, `wordsLib.SIZES_ss`, `wordsLib.WORD_ARITH_ss`, `wordsLib.WORD_LOGIC_ss`, `wordsLib.WORD_ARITH_EQ_ss`, `wordsLib.WORD_BIT_EQ_ss`, `wordsLib.WORD_SHIFT_ss`, `wordsLib.WORD_EXTRACT_ss`, `wordsLib.WORD_ss`

WORD_SHIFT_ss (wordsLib)

```
wordsLib.WORD_SHIFT_ss : ssfrag
```

Simplification fragment for words.

The fragment WORD_SHIFT_ss does some basic simplifications for the operations: << (left shift), >> (arithmetic right shift), >>> (logical right shift), #>> (rotate right) and #<< (rotate left). More simplification is possible when used in combination with wordsLib.SIZES_ss.

Example

```
> SIMP_CONV (std_ss++wordsLib.WORD_SHIFT_ss) [] “a << 2 << 3 + a >> 3 >> 2 + a >>> 1
val it =
   ⊢ a ≪ 2 ≪ 3 + a ≫ 3 ≫ 2 + a ⋙ 1 ⋙ 2 + a ⇆ 1 ⇆ 2 =
     a ≪ 5 + a ≫ 5 + a ⋙ 3 + a ⇆ 3: thm

> SIMP_CONV (std_ss++wordsLib.WORD_SHIFT_ss) [] “a >> 0 + 0w << n + a #<< 2 #>> 2”
val it = ⊢ a ≫ 0 + 0w ≪ n + a ⇆ 2 ⇄ 2 = a + 0w + a: thm
```

More simplification is possible when the word length is known.

```
> SIMP_CONV (std_ss++wordsLib.SIZES_ss++wordsLib.WORD_SHIFT_ss) [] “a << 4 + (a #<<
val it = ⊢ a ≪ 4 + a ⇆ 6 = 0w + a ⇆ 2: thm
```

Comments

When the word length is known the fragment WORD_ss simplifies #<< to #>>.

See also

fcpLib.FCP_ss, wordsLib.BIT_ss, wordsLib.SIZES_ss, wordsLib.WORD_ARITH_ss, wordsLib.WORD_LOGIC_ss, wordsLib.WORD_ARITH_EQ_ss, wordsLib.WORD_BIT_EQ_ss, wordsLib.WORD_EXTRACT_ss, wordsLib.WORD_MUL_LSL_ss, wordsLib.WORD_ss

WORD_ss (wordsLib)

wordsLib.WORD_ss : ssfrag

Simplification fragment for words.

The fragment `WORD_ss` contains `BIT_ss`, `SIZES_ss`, `WORD_LOGIC_ss`, `WORD_ARITH_ss` and `WORD_SHIFT_ss`. It also performs ground term evaluation.

Example

```
> SIMP_CONV (pure_ss++wordsLib.WORD_ss) [] “BIT i 42”
val it = ⊢ BIT i 42 ⇔ i ∈ {1; 3; 5}: thm

> SIMP_CONV (pure_ss++wordsLib.WORD_ss) [] “dimword(:42)”
val it = ⊢ dimword (:42) = 4398046511104: thm

> SIMP_CONV (pure_ss++wordsLib.WORD_ss) [] “((a #<< 2 #>> 2 + a) && $- 1w) - a”
Exception- HOL_ERR
  (at Preterm.type-analysis: on line 1, characters 69-72:

Type error in function application.
  Function: $&& (a ⇆ 2 ⇄ 2 + a) :α word -> α word
  Argument: $- 1w :α word -> α word
  Reason: Attempt to unify different type operators: fcp$cart and min$fun
) raised

> SIMP_CONV (pure_ss++wordsLib.WORD_ss) [] “(4 -- 2) ($- 1w : word8)”
Exception- HOL_ERR
  (at Preterm.type-analysis: on line 1, characters 55-58:

Type constraint failure:
  Term: $- 1w :α word -> α word
  Constraint: :word8
) raised
```

Comments

The `WORD_ss` fragment does not include `WORD_ARITH_EQ_ss`, `WORD_BIT_EQ_ss`, `WORD_-EXTRACT_ss` or `WORD_MUL_LSL_ss`. These extra fragments have more specialised applications.

See also

wordsLib.WORD_CONV, fcpLib.FCP_ss, wordsLib.BIT_ss, wordsLib.SIZES_ss, wordsLib.WORD_ARITH_ss, wordsLib.WORD_LOGIC_ss, wordsLib.WORD_ARITH_EQ_ss, wordsLib.WORD_BIT_EQ_ss, wordsLib.WORD_SHIFT_ss, wordsLib.WORD_EXTRACT_ss, wordsLib.WORD_MUL_LSL_ss

WORD_SUB_CONV (wordsLib)

wordsLib.WORD_SUB_CONV : conv

The conversion WORD_SUB_CONV is designed to eliminate occurrences of multiplication by a negative coefficient, introducing unary-minus (2's complement) and subtraction wherever possible.

Example

```
> wordsLib.WORD_SUB_CONV “a + -3w * b + -1w * c = -1w * d + e: 'a word”;
val it = ⊢ a + -3w * b + -1w * c = -1w * d + e ⇔ a − 3w * b − c = e − d: thm

> wordsLib.WORD_SUB_CONV “-1w * a: 'a word”;
val it = ⊢ -1w * a = -a: thm
```

Comments

This conversion forms the basis of wordsLib.WORD_SUB_ss. Care should be taken when using this conversion in combination with other bit-vector tools. For example, wordsLib.WORD_ARITH_CONV will undo all of the work of WORD_SUB_CONV.

See also

wordsLib.WORD_SUB_ss

WORD_SUB_ss (wordsLib)

```
wordsLib.WORD_SUB_ss : ssfrag
```

Simplification fragment for words.

The fragment `WORD_SUB_ss` applies the conversion `wordsLib.WORD_SUB_CONV`.

Comments

This fragment should not be used in combination with `wordsLib.WORD_ARITH_ss`, which also excludes use with `wordsLib.WORD_ss` and `srw_ss`.

See also

`wordsLib.WORD_SUB_CONV`

words2 (Lib)

```
Lib.words2 : string -> string -> string list
```

Splits a string into a list of substrings, breaking at occurrences of a specified character.

`words2 char s` splits the string `s` into a list of substrings. Splitting occurs at each occurrence of a sequence of the character `char`. The `char` characters do not appear in the list of substrings. Leading and trailing occurrences of `char` are also thrown away. If `char` is not a single-character string (its length is not 1), then `s` will not be split and so the result will be the list `[s]`.

Failure

Never fails.

Example

```
> words2  "/"  "/the/cat//sat/on//the/mat/";
val it = ["the", "cat", "sat", "on", "the", "mat"]: string list

> words2  "//"  "/the/cat//sat/on//the/mat/";
val it = ["/the/cat//sat/on//the/mat/"]: string list
```

Comments

The SML Library functions `String.tokens` and `String.fields` offer similar functionality.

wrap_exn (Feedback)

```
Feedback.wrap_exn : string -> string -> exn -> exn
```

Adds supplementary information to a `HOL_ERR` exception.

`wrap_exn s1 s2 (HOL_ERR e)` where `s1` is typically the name of a structure and `s2` is typically the name of a function, augments `e` by pushing `(s1,s2)` on to the stack of `origin` elements held in `e`. This can be used to achieve a kind of backtrace when an error occurs. `wrap_exn` can be applied to any exception.

Almost every non-`HOL_ERR` exception is mapped into an application of `HOL_ERR` by `wrap_exn`, but there is one special case: interrupts. A Unix `interrupt` signal is mapped into the `Interrupt` exception. If `wrap_exn` were to translate an `Interrupt` exception into a `HOL_ERR` exception, crucial information might be lost. For this reason, `wrap_exn s1 s2 Interrupt` raises the `Interrupt` exception.

Failure

Raises the exception argument when the exception argument is `Interrupt`.

Example

In the following example, the original `HOL_ERR` is from `Foo.bar`. After `wrap_exn` is called, the `HOL_ERR` is from `Fred.barney` and its message field has been augmented to reflect the original source of the exception.

```
> val orig_exn = mk_HOL_ERR "Foo" "bar" "incomprehensible input";
val orig_exn = HOL_ERR (at Foo.bar: incomprehensible input): exn

> wrap_exn "Fred" "barney" orig_exn;
val it =
   HOL_ERR
     (at Fred.barney:
        at Foo.bar: incomprehensible input): exn
```

The following example shows how `wrap_exn` treats the `Interrupt` exception.

```
> wrap_exn "Fred" "barney" Interrupt;
Exception- Interrupt raised
```

The following example shows how `wrap_exn` translates all exceptions that aren't either `HOL_ERR` or `Interrupt` into applications of `HOL_ERR`.

```
> wrap_exn "Fred" "barney" Div;
val it = HOL_ERR (at Fred.barney: Div): exn
```

See also

`Feedback`, `Feedback.HOL_ERR`, `Feedback.wrap_exn_loc`

wrap_exn_loc (Feedback)

```
Feedback.wrap_exn_loc : string -> string -> locn.locn -> exn -> exn
```

Adds supplementary information to a `HOL_ERR` exception, with location.

`wrap_exn_loc s1 s2 loc holerr` behaves like `wrap_exn` except that it includes the given location information with the other data pushed on to the `origins` stack.

Failure

Raises the exception argument when the exception argument is `Interrupt`.

Example

In the following example, the original HOL_ERR is from Foo.bar. After wrap_exn_loc is called, the HOL_ERR is from Fred.barney and its message field has been augmented to reflect the original source of the exception.

```
> val orig_exn = mk_HOL_ERR "Foo" "bar" "incomprehensible input";
val orig_exn = HOL_ERR (at Foo.bar: incomprehensible input): exn

> val exloc = let open locn in Loc (LocA(0,0), LocA(3,42)) end;
val exloc = 1:0-4:42: locn.locn

> wrap_exn_loc "Fred" "barney" exloc orig_exn;
val it =
   HOL_ERR
     (at Fred.barney: between line 1, character 0 and line 4, character 42:
        at Foo.bar: incomprehensible input): exn
```

See also

Feedback, Feedback.HOL_ERR, Feedback.wrap_exn

wrap_hol_error (Feedback)

```
wrap_hol_error : string -> string -> locn.locn -> hol_error -> hol_error
```

Add supplementary information to a hol_error value.

wrap_hol_error s1 s2 loc holerr where s1 is typically the name of a structure and s2 is typically the name of a function and loc is a location, augments holerr by pushing (s1,s2,loc) on to the stack of origin elements held in holerr.

Failure

Never fails.

Example

```
> val orig = mk_hol_error "Foo" "bar" locn.Loc_Unknown "mucho badness";
val orig = at Foo.bar: mucho badness: hol_error

> wrap_hol_error "Fred" "barney" locn.Loc_Unknown orig;
val it =
   at Fred.barney:
   at Foo.bar: mucho badness: hol_error
```

See also

`Feedback`, `Feedback.mk_hol_error`

X_CASES_THEN (Thm_cont)

```
Thm_cont.X_CASES_THEN : term list list -> thm_tactical
```

Applies a theorem-tactic to all disjuncts of a theorem, choosing witnesses.

Let `[yl1,...,yln]` represent a list of variable lists, each of length zero or more, and `xl1,...,xln` each represent a vector of zero or more variables, so that the variables in each of `yl1...yln` have the same types as the corresponding `xli`. `X_CASES_THEN` expects such a list of variable lists, `[yl1,...,yln]`, a tactic generating function `f:thm->tactic`, and a disjunctive theorem, where each disjunct may be existentially quantified:

```
th = |-(?xl1.B1)  \/...\/  (?xln.Bn)
```

each disjunct having the form `(?xi1 ... xim. Bi)`. If applying `f` to the theorem obtained by introducing witness variables `yli` for the objects `xli` whose existence is asserted by each disjunct, typically `({Bi[yli/xli]} |- Bi[yli/xli])`, produce the following results when applied to a goal `(A ?- t)`:

```
 A ?- t
========= f ({B1[yl1/xl1]} |- B1[yl1/xl1])
 A ?- t1
```

```
    ...

    A ?- t
  =========  f ({Bn[yln/xln]} |- Bn[yln/xln])
    A ?- tn
```

then applying `(X_CHOOSE_THEN [yl1,...,yln] f th)` to the goal `(A ?- t)` produces `n` subgoals.

```
           A ?- t
  ======================  X_CHOOSE_THEN [yl1,...,yln] f th
    A ?- t1  ...  A ?- tn
```

Failure

Fails (with `X_CHOOSE_THEN`) if any `yli` has more variables than the corresponding `xli`, or (with `SUBST`) if corresponding variables have different types. Failures may arise in the tactic-generating function. An invalid tactic is produced if any variable in any of the `yli` is free in the corresponding `Bi` or in `t`, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

Example

Given the goal `?- (x MOD 2) <= 1`, the following theorem may be used to split into 2 cases:

```
th = |- (?m. x = 2 * m) \/ (?m. x = (2 * m) + 1)
```

by the tactic

```
X_CASES_THEN [[Term`n:num`],[Term`n:num]] ASSUME_TAC th
```

to produce the subgoals:

```
{x = (2 * n) + 1} ?- (x MOD 2) <= 1
```

```
{x = 2 * n} ?- (x MOD 2) <= 1
```

See also

`Thm_cont.DISJ_CASES_THENL`, `Thm_cont.X_CASES_THENL`, `Thm_cont.X_CHOOSE_THEN`

X_CASES_THENL (Thm_cont)

```
Thm_cont.X_CASES_THENL : term list list -> thm_tactic list -> thm_tactic
```

Applies theorem-tactics to corresponding disjuncts of a theorem, choosing witnesses.

Let `[yl1,...,yln]` represent a list of variable lists, each of length zero or more, and `xl1,...,xln` each represent a vector of zero or more variables, so that the variables in each of `yl1...yln` have the same types as the corresponding `xli`. The function `X_CASES_THENL` expects a list of variable lists, `[yl1,...,yln]`, a list of tactic-generating functions `[f1,...,fn]:(thm->tactic)list`, and a disjunctive theorem, where each disjunct may be existentially quantified:

```
th = |-(?xl1.B1)  \/...\/  (?xln.Bn)
```

each disjunct having the form `(?xi1 ... xim. Bi)`. If applying each `fi` to the theorem obtained by introducing witness variables `yli` for the objects `xli` whose existence is asserted by the `i`th disjunct, `({Bi[yli/xli]} |- Bi[yli/xli])`, produces the following results when applied to a goal `(A ?- t)`:

```
 A ?- t
=========  f1 ({B1[yl1/xl1]} |- B1[yl1/xl1])
 A ?- t1

 ...

 A ?- t
=========  fn ({Bn[yln/xln]} |- Bn[yln/xln])
 A ?- tn
```

then applying `X_CASES_THENL [yl1,...,yln] [f1,...,fn] th` to the goal `(A ?- t)` produces `n` subgoals.

```
        A ?- t
======================  X_CASES_THENL [yl1,...,yln] [f1,...,fn] th
 A ?- t1  ...  A ?- tn
```

Failure

Fails (with X_CASES_THENL) if any yli has more variables than the corresponding xli, or (with SUBST) if corresponding variables have different types, or (with combine) if the number of theorem tactics differs from the number of disjuncts. Failures may arise in the tactic-generating function. An invalid tactic is produced if any variable in any of the yli is free in the corresponding Bi or in t, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

Example

Given the goal ?- (x MOD 2) <= 1, the following theorem may be used to split into 2 cases:

```
th = |- (?m. x = 2 * m) \/ (?m. x = (2 * m) + 1)
```

by the tactic

```
X_CASES_THENL [[Term`n:num`], [Term`n:num`]] [ASSUME_TAC, SUBST1_TAC] th
```

to produce the subgoals:

```
?- (((2 * n) + 1) MOD 2) <= 1
```

```
{x = 2 * n} ?- (x MOD 2) <= 1
```

See also

Thm_cont.DISJ_CASES_THEN, Thm_cont.X_CASES_THEN, Thm_cont.X_CHOOSE_THEN

X_CHOOSE_TAC (Tactic)

```
Tactic.X_CHOOSE_TAC : term -> thm_tactic
```

Assumes a theorem, with existentially quantified variable replaced by a given witness.

X_CHOOSE_TAC expects a variable y and theorem with an existentially quantified conclusion. When applied to a goal, it adds a new assumption obtained by introducing the variable y as a witness for the object x whose existence is asserted in the theorem.

```
         A ?- t
==================  X_CHOOSE_TAC y (A1 |- ?x. w)
 A u {w[y/x]} ?- t          (y not free anywhere)
```

Failure

Fails if the theorem's conclusion is not existentially quantified, or if the first argument is not a variable. Failures may arise in the tactic-generating function. An invalid tactic is produced if the introduced variable is free in `w`, `t` or `A`, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

Example

Given a goal of the form

```
{n < m} ?- ?x. m = n + (x + 1)
```

the following theorem may be applied:

```
th = [n < m] |- ?p. m = n + p
```

by the tactic `(X_CHOOSE_TAC (Term`q:num`) th)` giving the subgoal:

```
{n < m, m = n + q} ?- ?x. m = n + (x + 1)
```

See also

`Thm.CHOOSE`, `Thm_cont.CHOOSE_THEN`, `Thm_cont.X_CHOOSE_THEN`

X_CHOOSE_THEN (Thm_cont)

```
Thm_cont.X_CHOOSE_THEN : (term -> thm_tactical)
```

Replaces existentially quantified variable with given witness, and passes it to a theorem-tactic.

X_CHOOSE_THEN expects a variable y, a tactic-generating function f:thm->tactic, and a theorem of the form (A1 |- ?x. w) as arguments. A new theorem is created by introducing the given variable y as a witness for the object x whose existence is asserted in the original theorem, (w[y/x] |- w[y/x]). If the tactic-generating function f applied to this theorem produces results as follows when applied to a goal (A ?- t):

```
 A ?- t
=========  f ({w[y/x]} |- w[y/x])
 A ?- t1
```

then applying (X_CHOOSE_THEN “y” f (A1 |- ?x. w)) to the goal (A ?- t) produces the subgoal:

```
 A ?- t
=========  X_CHOOSE_THEN “y” f (A1 |- ?x. w)
 A ?- t1          (y not free anywhere)
```

Failure

Fails if the theorem’s conclusion is not existentially quantified, or if the first argument is not a variable. Failures may arise in the tactic-generating function. An invalid tactic is produced if the introduced variable is free in w, t or A, or if the theorem has any hypothesis which is not alpha-convertible to an assumption of the goal.

Example

Given a goal of the form

```
{n < m} ?- ?x. m = n + (x + 1)
```

the following theorem may be applied:

```
th = [n < m] |- ?p. m = n + p
```

by the tactic X_CHOOSE_THEN “q:num” SUBST1_TAC th giving the subgoal:

```
{n < m} ?- ?x. n + q = n + (x + 1)
```

See also

Thm.CHOOSE, Thm_cont.CHOOSE_THEN, Thm_cont.CONJUNCTS_THEN, Thm_cont.CONJUNCTS_-THEN2, Thm_cont.DISJ_CASES_THEN, Thm_cont.DISJ_CASES_THEN2, Thm_cont.DISJ_-CASES_THENL, Thm_cont.STRIP_THM_THEN, Tactic.X_CHOOSE_TAC

X_FUN_EQ_CONV (Conv)

```
Conv.X_FUN_EQ_CONV : (term -> conv)
```

Performs extensionality conversion for functions (function equality).

The conversion X_FUN_EQ_CONV embodies the fact that two functions are equal precisely when they give the same results for all values to which they can be applied. For any variable "x" and equation "f = g", where x is of type ty1 and f and g are functions of type ty1->ty2, a call to X_FUN_EQ_CONV "x" "f = g" returns the theorem:

```
|- (f = g) = (!x. f x = g x)
```

Failure

X_FUN_EQ_CONV x tm fails if x is not a variable or if tm is not an equation f = g where f and g are functions. Furthermore, if f and g are functions of type ty1->ty2, then the variable x must have type ty1; otherwise the conversion fails. Finally, failure also occurs if x is free in either f or g.

See also

Drule.EXT, Conv.FUN_EQ_CONV

X_GEN_TAC (Tactic)

```
Tactic.X_GEN_TAC : (term -> tactic)
```

Specializes a goal with the given variable.

When applied to a term `x'`, which should be a variable, and a goal `A ?- !x. t`, the tactic `X_GEN_TAC` returns the goal `A ?- t[x'/x]`.

```
  A ?- !x. t
============  X_GEN_TAC "x'"
 A ?- t[x'/x]
```

Failure

Fails unless the goal's conclusion is universally quantified and the term a variable of the appropriate type. It also fails if the variable given is free in either the assumptions or (initial) conclusion of the goal.

See also

`Tactic.FILTER_GEN_TAC`, `Thm.GEN`, `Thm.GENL`, `Drule.GEN_ALL`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`, `Tactic.SPEC_TAC`, `Tactic.STRIP_TAC`

X_SKOLEM_CONV (Conv)

```
Conv.X_SKOLEM_CONV : (term -> conv)
```

Introduces a user-supplied Skolem function.

`X_SKOLEM_CONV` takes two arguments. The first is a variable `f`, which must range over functions of the appropriate type, and the second is a term of the form `!x1...xn. ?y. P`. Given these arguments, `X_SKOLEM_CONV` returns the theorem:

```
|- (!x1...xn. ?y. P) = (?f. !x1...xn. tm[f x1 ... xn/y])
```

which expresses the fact that a skolem function `f` of the universally quantified variables `x1...xn` may be introduced in place of the the existentially quantified value `y`.

Failure

`X_SKOLEM_CONV f tm` fails if `f` is not a variable, or if the input term `tm` is not a term of the form `!x1...xn. ?y. P`, or if the variable `f` is free in `tm`, or if the type of `f` does not match its intended use as an `n`-place curried function from the variables `x1...xn` to a value having the same type as `y`.

See also

`Conv.SKOLEM_CONV`

xDefine (TotalDefn)

```
TotalDefn.xDefine : string -> term quotation -> thm
```

Re-exported from `bossLib.xDefine`. See that entry for full documentation.

xDefine (bossLib)

```
bossLib.xDefine : string -> term quotation -> thm
```

Also exported as `TotalDefn.xDefine`.

General-purpose function definition facility.

`xDefine` behaves exactly like `Define`, except that it takes an alphanumeric string which is used as a stem for building names with which to store the definition, associated induction theorem (if there is one), and any auxiliary definitions used to construct the specified function (if there are any) in the current theory segment.

Failure

`xDefine` allows the definition of symbolic identifiers, but `Define` doesn't. In all other respects, `xDefine` and `Define` succeed and fail in the same way.

Example

The following example shows how Define fails when asked to define a symbolic identifier.

```
- set_fixity ("/", Infixl 600);    (* tell the parser about "/" *)
> val it = () : unit

- Define
   `x/y = if y=0 then NONE else
          if x<y then SOME 0
           else OPTION_MAP SUC ((x-y)/y)`;

Definition failed! Can't make name for storing definition
because there is no alphanumeric identifier in:

   "/".

Try "xDefine <alphanumeric-stem> <eqns-quotation>" instead.
```

Next the same definition is attempted with xDefine, supplying the name for binding the definition and the induction theorem with in the current theory.

```
- xDefine "div"
     `x/y = if y=0 then NONE else
            if x<y then SOME 0
             else OPTION_MAP SUC ((x-y)/y)`;

Equations stored under "div_def".
Induction stored under "div_ind".

> val it =
    |- x / y =
       (if y = 0 then NONE
        else
         (if x < y then SOME 0
                   else OPTION_MAP SUC ((x - y) / y))) : thm
```

Comments

Define can be thought of as an application of xDefine, in which the stem is taken to be the name of the function being defined.

`bossLib.xDefine` is most commonly used. `TotalDefn.xDefine` is identical to `bossLib.xDefine`, except that the `TotalDefn` structure comes with less baggage—it depends only on `numLib` and `pairLib`.

See also

`bossLib.Define`

zDefine (bossLib)

```
bossLib.zDefine : term quotation -> thm
```

General-purpose function definition facility.

`zDefine` behaves exactly like `Define`, except that it does not add the definition to `computeLib.the_compset`. Consequently the definition is not used by `bossLib.EVAL` when evaluating expressions.

Failure

`zDefine` and `Define` succeed and fail in the same way.

Example

```
> zDefine `foo = 10 ** 10 ** 10`
val it = ⊢ foo = 10 ** 10 ** 10: thm
> EVAL ``foo``;
val it = ⊢ foo = foo: thm
```

Comments

`zDefine` is helpful when users wish to derive and use their own efficient evaluation theorems, which can be added using `computeLib.add_funs` or `computeLib.add_persistent_funs`.

See also

bossLib.Define

zip (Lib)

```
Lib.zip : 'a list -> 'b list -> ('a * 'b) list
```

Transforms a pair of lists into a list of pairs.

`zip [x1,...,xn] [y1,...,yn]` returns `[(x1,y1),...,(xn,yn)]`.

Failure

Fails if the two lists are of different lengths.

Comments

Has much the same effect as the SML Basis function `ListPair.zip` except that it fails if the arguments are not of equal length. `zip` is a curried version of `combine`

See also

`Lib.combine`, `Lib.unzip`, `Lib.split`