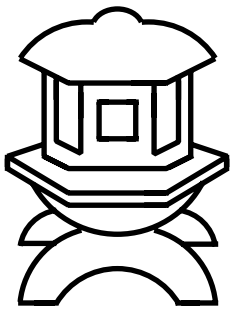
[For HOL Kananaskis-1] June 17, 2002

The HOL System

REFERENCE



Preface

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

(i) *TUTORIAL*: a tutorial introduction to HOL, with case studies.

(ii) *DESCRIPTION*: a description of higher order logic, the ML programming language, and theorem proving methods in the HOL system;

(iii) *REFERENCE*: the reference documentation of the tools available in HOL.

These three 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 manual entries for these ML identifiers are divided into two chapters. The first chapter is an alphabetical sequence of manual entries for all ML identifiers in the system except those identifiers that are bound to theorems. The theorems are listed in the second chapter, roughly grouped into sections based on subject matter.

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*.

Acknowledgements

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.

Later editions

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

The third edition of all three volumes represents a wide-ranging and still incomplete revision of material written for HOL88 so that it applies to the HOL system a decade later. The third edition has been prepared by Konrad Slind and Michael Norrish.

[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.

Contents

Chapter 1

Pre-defined ML Identifiers

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)

```
op ## : ('a -> 'b) * ('c -> 'd) -> 'a * 'c -> 'b * 'd
```

Synopsis
Infix combinator for applying two functions to the two projections of a pair.

Description
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`))
```

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`.

&& (BasicProvers)

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

Synopsis
Infix operator for adding theorems into a simpset.

Description
`BasicProvers.&&` is identical to `bossLib.&&`.

See also
`bossLib.&&`.

&& (bossLib)

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

Synopsis
Infix operator for adding theorems into a simpset.

Description
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

```
- open bossLib;
... <output elided> ...
- val ss = boolSimps.bool_ss && pairTheory.pair_rws;
> val ss = <simpset> : simpset
```

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, bossLib.ZAP_TAC.

++ (simpLib)

```
op ++ : simpset * ssdata -> simpset
```

Synopsis
Infix operator for adding an ssdata item into a simpset.

Description
bossLib.++ is identical to simpLib.++.

See also
bossLib.++.

-- (Parse)

```
-- : term quotation -> 'a -> term
```

Synopsis
Parses a quotation into a term value

Description
An invocation --‘ ... ‘-- is identical to Term ‘ ... ‘.

Failure
As for Parse.Term.

Uses
Turns strings into terms.

See also
Parse.Term, Parse.Type, Parse.==.

--> (Type)

```
op --> : hol_type * hol_type -> hol_type
```

Synopsis
Right associative infix operator for building a function type.

Description
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 -> ’a‘ : 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)

```
== : hol_type quotation -> ’a -> hol_type
```

Synopsis
Parses a quotation into a HOL type.

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

Failure
As for `Parse.Type`.

Uses
Turns strings into types.

See also
`Parse.Term, Parse.Type, Parse.--.`

A (Lib)

```
A : ('a -> 'b) -> 'a -> 'b
```

Synopsis
Combinator for function application

Description
The application `A f x` equals `f x`.

Failure
`A f` never fails. `A f x` fails if `f x` fails.

Example

```
- map2 A [I, K 3, fn x => x + 1]  [1,2,3];
```

```
> val it = [1, 3, 4] : int list
```

See also
`Lib, Lib.##, Lib.B, Lib.C, Lib.I, Lib.K, Lib.S, Lib.W.`

ABS (Thm)

```
ABS : term -> thm -> thm
```

Synopsis
Abstracts both sides of an equation.

Description

```
        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 = Term `m:bool`
  in
      ABS m (REFL m)
  end;

> val it = |- (\m. m) = (\m. m) : thm
```

See also

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

ABS_CONV (Conv)

```
ABS_CONV : conv -> conv
```

Synopsis

Applies a conversion to the body of an abstraction.

Description

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 = |- (\x. 1 = x) = (\x. 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`.

Absyn (Parse)

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

Synopsis

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

Description

`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.

Uses

`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_CONV (Conv)

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

Synopsis

Proves equality of terms using associative and commutative laws.

Description

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)

```
ACCEPT_TAC : thm_tactic
```

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

Description
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)
```

Uses
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)

```
aconv : term -> term -> bool
```

Synopsis
Tests for alpha-convertibility of terms.

Description
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)

```
ADD_ASSUM : term -> thm -> thm
```

Synopsis
Adds an assumption to a theorem.

Description
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
```

`ADD_ASSUM` performs straightforward set union with the new assumption; it checks for identical assumptions, but not for alpha-equivalent ones. The position at which the new assumption is inserted into the assumption list should not be relied on.

Failure
Fails unless the given term has type `bool`.

See also
`Thm.ASSUME, Drule.UNDISCH.`

add_bare_numeral_form (Parse)

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

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

Description
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
```

Uses

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_implicit_rewrites (Rewrite)

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

Synopsis

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

Uses

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

See also

base_rewrites, Rewrite.set_implicit_rewrites.

add_infix (Parse)

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

Synopsis

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

Description

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.

Uses

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_binder`, `Parse.add_rule`, `Parse.add_listform`, `Parse.Term`.

add_listform (Parse)

```
add_listform :
  {separator : string, leftdelim : string, rightdelim : string,
   cons : string, nilstr : string} -> unit
```

Synopsis

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

Description

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>`.

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

Should never fail itself, but subsequent calls to the term parser may well fail 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 = ";", leftdelim = "[", rightdelim = "]",
              cons = "CONS", nilstr = "NIL"};
```

while the set syntax is defined similarly:

```
add_listform {leftdelim = "{", rightdelim = "}", separator = ";",
              cons = "INSERT", nilstr = "EMPTY"};
```

Uses

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
```

Synopsis

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

Description

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)

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

Synopsis

Add theorems to a collection of rewrite rules.

Description

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::pair_rws);

> 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
```

Uses

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)

```
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
```

Synopsis

Adds a parsing/printing rule to the global grammar.

Description

The function `add_rule` is a fundamental method for adding parsing (and thus printing) rules to the global term grammar that sits behind the functions `Term` and `--`, 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 Prefix     : fixity
  val Binder     : fixity
  val Closefix   : fixity
  val Infixl     : int -> fixity
  val Infixr     : int -> fixity
  val Infix      : HOLgrammars.associativity * int -> fixity
  val TruePrefix : int -> fixity
  val Suffix     : int -> fixity
```

The `Prefix` fixity has an unfortunate name, as it is a fixity corresponding to no special treatment. In fact, when a `Prefix` fixity is specified, the `add_rule` function performs no action. When an element list is meant to form a genuine prefix, the `TruePrefix` fixity must be used instead, as is done below in the conditional expression example and as is

also done with ~ (logical negation). The Prefix fixity is useful elsewhere, in situations where standard interfaces require fixities to be provided, but where the user may wish to leave an identifier as a normal symbol.

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 *)
TruePrefix  :      [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 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_element`s. 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 bound-

ary 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. 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

There are two conditional expression syntaxes defined in the theory `bool`. The first is the traditional HOL88/90 syntax. Because the syntax involves “dangling” terms 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 second rule added uses the more familiar `if-then-else` syntax. Here there is only a "dangling" term to the right of the construction, so this rule's fixity is of type `TruePrefix`. (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 = TruePrefix 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 pretty-printer prefers later rules over earlier rules by default (though this choice can be changed with `prefer_form_with_tok` (q.v.)), so conditional expressions print using the `if-then-else` syntax rather than the `_ => _  | _` syntax.

Uses

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.

The `Prefix/TruePrefix` situation may be transitory. It has the advantage of maintaining a deal of backwards compatibility, but at the cost of confusing the terminology. Where the `Prefix` value is acceptable, the `fixity` type should be replaced by a `fixity option` type to better reflect the semantics of what is really happening.

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_user_printer (Parse)

```
add_user_printer :
  ({Tyop : string, Thy : string} * userprinter * string) -> unit
```

Synopsis

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

Description

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 type of a term to be printed is as given by the first parameter of the triple. This first parameter specifies an operator that is not necessarily nullary, so that if the `Tyop-Thy` pair is `list-list` for example, then the printing function will be called on all values of type `:x list`, where `x` is any type.

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 of the given type.

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

```
type userprinter =
  sysprinter -> (grav * grav * grav) -> int -> Portable.ppstream ->
  term -> unit
```

where the type `grav` is

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

and the type `sysprinter` is another abbreviation

```
type sysprinter = (grav * grav * grav) -> int -> term -> unit
```

Thus, when the user's printing function is called, it is passed seven parameters, including three "gravity" values in a triple. The first parameter is the system's own printer, so that the user function can 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.

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.

Finally, the string parameter to the `add_user_printer` function is a string corresponding to the ML function. Best practice is probably to define the printing function in an independent structure and to then have the string be of the form `"module.fnname"`. This parameter is not present in the accompanying `temp_add_user_printer`, as this latter function does not affect the grammar exported to disk with `export_theory`.

Failure

Will not fail directly, but 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. If the string parameter does not correspond to valid ML code, then the theory file generated by `export_theory` will not compile.

Example

This example uses the system printer to print sub-terms, and concerns itself only with

printing conjunctions:

```
- fun myprint sys gravs d pps t = let
    open Portable term_pp_types
    val (l,r) = dest_conj t
  in
    add_string pps "CONJ:";
    add_break pps (1,0);
    sys (Top, Top, Top) (d - 1) l;
    add_string " and then ";
    sys (Top, Top, Top) (d - 1) r;
    add_string "ENDCONJ"
  end handle HOL_ERR _ => raise term_pp_types.UserPP_Failed;
> val ('a, 'b) myprint = fn :
  (grav * grav * grav -> int -> term -> 'a) -> 'b -> int ->
  ppstream -> term -> unit

- temp_add_user_printer ({Tyop = "bool", Thy = "min"}, 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.

Another approach to printing conjunctions is not possible with `add_rule`:

```
- fun myprint2 sys (pg,lg,rg) d pps t = let
    open Portable term_pp_types
    val (l,r) = dest_conj t
    fun delim act = case pg of
                      Prec(_, "CONJ") => ()
                    | _ => act()
  in
    delim (fn () => (begin_block pps CONSISTENT 0;
                     add_string pps "CONJ";
                     add_break pps (1,2);
                     begin_block pps INCONSISTENT 0));
    sys (Prec(0, "CONJ"), Top, Top) (d - 1) l;
    add_string pps ",";
    add_break pps (1,0);
    sys (Prec(0, "CONJ"), Top, Top) (d - 1) r;
    delim (fn () => (end_block pps;
                     add_break pps (1,0);
                     add_string pps "ENDCONJ";
                     end_block pps))
  end handle HOL_ERR _ => raise term_pp_types.UserPP_Failed;
> val ('a, 'b, 'c) myprint2 = fn :
  (grav * grav * grav -> int -> term -> 'a) -> grav * 'b * 'c ->
  int -> ppstream -> term -> unit

- temp_add_user_printer ({Tyop = "bool", Thy = "min"}, myprint2);
> val it = () : unit

- ``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 examples demonstrates using pretty-printer blocks in order to get a pleasing effect, and also using parent gravities to print out a big term. Note also how the flow of control doubles backwards and forwards between the system printer and the user's. 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. Finally, this example demonstrates how easy it is to conceal genuine syntactic structure with a pretty-printer.

Uses

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

See also

`Parse.remove_user_printer`.

adjoin_to_theory (Theory)

```
adjoin_to_theory : thy_addon -> unit
```

Synopsis

Include arbitrary ML in exported theory.

Description

It often happens that algorithms and flag settings accompany a logical theory (call it `thy`). One would want to simply load the `thyTheory` module and have the appropriate proof support, etc. loaded automatically as well.

There are several ways to support this. One simple way would be to define another ML structure, `thySupport` say, that depended on `thyTheory`. The algorithms, etc, could be placed in `thySupport` and the interested user would know that by loading `thySupport`, its contents, and those of `thyTheory`, would become available. This approach, and extensions of it are accomodated already in the notion of a HOL library.

However, it is sometimes more appropriate to actually include the support code directly in `thyTheory`. The function `adjoin_to_theory` performs this operation.

A call `adjoin_to_theory {sig_ps, struct_ps}` adds a signature prettyprinter `sig_ps` and a structure prettyprinter `struct_ps` to an internal queue of prettyprinters. When `export_theory ()` is eventually called two things happen: (a) the signature file `thyTheory.sig` is written, and (b) the structure file `thyTheory.sml` is written. When `thyTheory.sig` is written, each signature prettyprinter in the queue is called, in the order that they were added to the queue. This printing activity happens after the rest of the signature (coming from the declarations in the theory) has been written. Similarly, when `thyTheory.sml` is written, the structure prettyprinters are invoked in queue order, after the bindings of the theory have been written.

If `sig_ps` is `NONE`, then no signature additions are made. Likewise, if `struct_ps` is `NONE`, then no structure additions are made. (This latter possibility doesn't seem to be useful.)

Failure
It is up to the writer of a prettyprinter to ensure that it generates valid ML. If a prettyprinter added by a call to `adjoin_to_theory` fails, `thyTheory.sig` or `thyTheory.sml` could be malformed, and therefore not properly exported, or compiled.

Example
The following excerpt from the script for the theory of pairs is a fairly typical use of `adjoin_to_theory`. It adds the declaration of an ML variable `pair_rws` to the structure `pairTheory`.

```
val _ = adjoin_to_theory
{sig_ps =
   SOME(fn ppstrm => PP.add_string ppstrm "val pair_rws:thm list"),
 struct_ps =
   SOME(fn ppstrm => PP.add_string ppstrm
                           "val pair_rws = [PAIR, FST, SND];")
}
```

Comments
The `PP` structure is documented in the MoscowML library documentation.

See also
`Theory.after_new_theory`, `Theory.thy_addon`, `BasicProvers.export_rewrites`.

after_new_theory (Theory)

```
after_new_theory : (string -> unit) -> unit
```

Synopsis
Initialize package once a theory is declared.

Description
Some HOL infrastructure depends on certain packages being informed each time a new theory is created. The function `after_new_theory` supports this. An invocation `after_new_theory f` adds the function `f` to an internal queue of 'initializers'. All subsequent calls to `new_theory` will cause each initializer to be run, in queue order. Each initializer will be given the name of the theory as argument.

Failure

It can be that an initializer fails for some reason when it is executed. Any exceptions will be caught, and an attempt will be made to print out a message. Then execution of the remaining initializers will continue.

Example

```
- fun every8 s (a::b::c::d::e::f::g::h::rst) =
                a::b::c::d::e::f::g::h::s::every8 s rst
    | every8 s otherwise = otherwise;
> val 'a every8 = fn : 'a -> 'a list -> 'a list

- after_new_theory (fn s =>
    (print ("Ancestors of "^s^":\n  ");
     print (String.concat (every8 "\n  " (commafy (ancestry s))));
     print ".\n"));
> val it = () : unit

- new_theory"foo";
<<HOL message: Created theory "foo">>
Ancestors of foo:
  one, option, pair, sum,
  combin, relation, min, bool,
  num, prim_rec, arithmetic, numeral,
  ind_type, list.
> val it = () : unit

- new_theory"bar";
Exporting theory "foo" ... done.
<<HOL message: Created theory "bar">>
Ancestors of bar:
  one, option, pair, sum,
  combin, relation, min, bool,
  num, prim_rec, arithmetic, numeral,
  ind_type, list, foo.
> val it = () : unit
```

Comments

Perhaps there should be a `before_export_theory` call as well?

Uses

Fairly low level system support tasks.

See also

`Theory.adjoin_to_theory`.

all (Lib)

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

Synopsis
Tests whether a predicate holds throughout a list.

Description
`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` 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];
! Uncaught exception:
! Fail  ""
```

See also
`Lib.all2`, `Lib.exists`, `Lib.first`.

all2 (Lib)

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

Synopsis
Tests whether a predicate holds pairwise throughout two lists.

Description

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 raised at Lib.all2:
different length lists
! Uncaught exception:
! HOL_ERR

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

- all2 (fn _ => fn _ => raise Fail "") [1] [1];
! Uncaught exception:
! Fail  ""
```

See also

`Lib.all`.

all_consts (Term)

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

Synopsis
All known constants in the current theory.

Description
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 =
    [‘transitive‘, ‘CONS‘, ‘RES_ABSTRACT‘, ‘COND‘, ‘OPTION_MAP‘, ‘FCONS‘,
     ‘FACT‘, ‘&‘, ‘RPROD‘, ‘mk_list‘, ‘ZIP‘, ‘IS_NUM_REP‘, ‘ABS_sum‘, ‘SUM‘,
     ‘SUC‘, ‘OPTION_JOIN‘, ‘REP_sum‘, ‘RTC‘, ‘SND‘, ‘RES_SELECT‘, ‘THE‘,
     ‘APPEND‘, ‘option_REP‘, ‘PRE‘, ‘ABS_num‘, ‘PRIM_REC‘, ‘EXISTS‘, ‘REP_num‘,
     ‘approx‘, ‘case‘, ‘CONSTR‘, ‘[]‘, ‘$MOD‘, ‘ODD‘, ‘MIN‘, ‘case‘, ‘MEM‘,
     ‘ISR‘, ‘MAX‘, ‘$LEX‘, ‘ISO‘, ‘case_arrow__magic‘, ‘ISL‘, ‘LET‘, ‘MAP‘,
     ‘INR‘, ‘INL‘, ‘$EXP‘, ‘FST‘, ‘case‘, ‘mk_rec‘, ‘IS_SOME‘, ‘$DIV‘, ‘ARB‘,
     ‘option_ABS‘, ‘wellfounded‘, ‘iiSUC‘, ‘SIMP_REC_REL‘, ‘RES_FORALL‘,
     ‘$==>‘, ‘MK_PAIR‘, ‘ZBOT‘, ‘IS_NONE‘, ‘TYPE_DEFINITION‘, ‘case‘,
     ‘dest_rec‘, ‘IS_PAIR‘, ‘ONE_ONE‘, ‘case‘, ‘RES_EXISTS_UNIQUE‘, ‘NUMRIGHT‘,
     ‘NUMPAIR‘, ‘FILTER‘, ‘BOTTOM‘, ‘SOME‘, ‘reflexive‘, ‘EMPTY_REL‘,
     ‘REVERSE‘, ‘ABS_prod‘, ‘NUMERAL_BIT2‘, ‘NUMERAL_BIT1‘, ‘FRONT‘, ‘OUTR‘,
     ‘OUTL‘, ‘SIMP_REC‘, ‘measure‘, ‘NUMLEFT‘, ‘REP_prod‘, ‘list1‘, ‘list0‘,
     ‘NULL‘, ‘ONTO‘, ‘EVERY‘, ‘inv_image‘, ‘list_size‘, ‘NONE‘, ‘ALT_ZERO‘,
     ‘case__magic‘, ‘UNCURRY‘, ‘UNZIP‘, ‘FOLDR‘, ‘FOLDL‘, ‘iBIT_cases‘,
     ‘NUMERAL‘, ‘ZRECSPACE‘, ‘iZ‘, ‘case‘, ‘iSUB‘, ‘iSQR‘, ‘ZCONSTR‘, ‘WFREC‘,
     ‘WF‘, ‘$\/‘, ‘TL‘, ‘TC‘, ‘RC‘, ‘case_split__magic‘, ‘$IN‘, ‘NUMSUM‘, ‘HD‘,
     ‘EL‘, ‘MAP2‘, ‘CURRY‘, ‘RES_EXISTS‘, ‘LAST‘, ‘NUMSND‘, ‘()‘, ‘$>=‘, ‘$<=‘,
     ‘INJP‘, ‘INJN‘, ‘INJF‘, ‘$?!‘, ‘INJA‘, ‘$/\‘, ‘IS_SUM_REP‘, ‘RESTRICT‘,
     ‘iDUB‘, ‘$##‘, ‘FUNPOW‘, ‘NUMFST‘, ‘EVEN‘, ‘SUC_REP‘, ‘$~‘, ‘dest_list‘,
     ‘$o‘, ‘FNIL‘, ‘W‘, ‘the_fun‘, ‘T‘, ‘S‘, ‘LENGTH‘, ‘PRIM_REC_FUN‘, ‘K‘,
     ‘I‘, ‘F‘, ‘combin$C‘, ‘$@‘, ‘$?‘, ‘$>‘, ‘$=‘, ‘$<‘, ‘ZERO_REP‘, ‘0‘, ‘$-‘,
     ‘$,‘, ‘FLAT‘, ‘$+‘, ‘$*‘, ‘$!‘] : term list
```

See also
`Parse.term_grammar`.

ALL_CONV (Conv)

```
ALL_CONV : conv
```

Synopsis
Conversion that always succeeds and leaves a term unchanged.

Description
When applied to a term `t`, the conversion `ALL_CONV` returns the theorem `|- t = t`.

Failure
Never fails.

Uses
Identity element for `THENC`.

See also
`Conv.NO_CONV, Thm.REFL`.

ALL_TAC (Tactical)

```
ALL_TAC : tactic
```

Synopsis
Passes on a goal unchanged.

Description
`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.

Uses
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)

```
ALL_THEN : thm_tactical
```

Synopsis
Passes a theorem unchanged to a theorem-tactic.

Description
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.

Uses
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_thys (DB)

```
all_thys : unit -> data list
```

Synopsis
All theorems, axioms, and definitions in the currently loaded theory segments.

Description
An invocation `all_thys()` returns everything that has been stored in all theory segments currently loaded.

Example

```
- length (all_thys());
> val it = 736 : int
```

See also

DB.thy, DB.theorems, DB.definitions, DB.axioms, DB.find, DB.match.

all_vars (Term)

```
all_vars : term -> term list
```

Synopsis

Returns the set of all variables in a term.

Description

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

Failure

Never fails.

Example

```
- all_vars (Term `!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.free_vars, Term.all_varsl.

all_varsl (Term)

```
all_varsl : term list -> term list
```

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

Description
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.FVL`, `Term.free_vars_lr`, `Term.free_vars`, `Term.free_varsl`, `Term.empty_varset`, `Type.type_vars`.

allowed_term_constant (Lexis)

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

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

Description
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`, `is_constant`, `new_alphanum`, `new_special_symbol`, `special_symbols`, `Lexis.allowed_type_constant`.

allowed_type_constant (Lexis)

```
allowed_type_constant : string -> bool
```

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

Description
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)

```
ALPHA : term -> term -> thm
```

Synopsis
Proves equality of alpha-equivalent terms.

Description
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)

```
alpha : hol_type
```

Synopsis
Common type variable.

Description
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)

```
ALPHA_CONV : term -> conv
```

Synopsis
Renames the bound variable of a lambda-abstraction.

Description
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)

```
ancestry : string -> string list
```

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

Description
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 = "scratch" : string

- ancestry "-";
> val it =
    ["one", "option", "pair", "sum", "combin", "relation", "min", "bool",
     "num", "prim_rec", "arithmetic", "numeral", "ind_type", "list"] :
  string list
```

See also

Theory.parents.

AND_EXISTS_CONV (Conv)

```
AND_EXISTS_CONV : conv
```

Synopsis

Moves an existential quantification outwards through a conjunction.

Description

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_
and RIGHT_EXISTS_AND_THM.

See also

Conv.EXISTS_AND_CONV, Conv.LEFT_AND_EXISTS_CONV, Conv.RIGHT_AND_EXISTS_CONV, BOTH_EXISTS_AND_THM, LEFT_EXISTS_AND_THM, RIGHT_EXISTS_AND_THM.

AND_FORALL_CONV (Conv)

```
AND_FORALL_CONV : conv
```

Synopsis
Moves a universal quantification outwards through a conjunction.

Description
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)

```
AND_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification outwards through a conjunction.

Description
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)

```
AND_PFORALL_CONV : conv
```

Synopsis

Moves a paired universal quantification outwards through a conjunction.

Description

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)

```
ANTE_CONJ_CONV : conv
```

Synopsis

Eliminates a conjunctive antecedent in favour of implication.

Description

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"`.

Uses

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)

```
ANTE_RES_THEN : thm_tactical
```

Synopsis

Resolves implicative assumptions with an antecedent.

Description

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`.

Uses

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)

```
AP_TERM : term -> thm -> thm
```

Synopsis

Applies a function to both sides of an equational theorem.

Description

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)

```
AP_TERM_TAC : tactic
```

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

Description
`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`.

AP_THM (Thm)

```
AP_THM : thm -> term -> thm
```

Synopsis
Proves equality of equal functions applied to a term.

Description
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`, `Thm.ETA_CONV`, `Drule.EXT`, `Conv.FUN_EQ_CONV`, `Thm.MK_COMB`.

AP_THM_TAC (Tactic)

```
AP_THM_TAC : tactic
```

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

Description
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`, `Drule.EXT`.

append (Lib)

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

Synopsis
Curried form of list append

Description
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
```

apropos (DB)

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

Synopsis

Attempt to find matching theorems in the currently loaded theories.

Description

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 =
    [(("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)),
     (("pair", "pair_induction"),
      (|- (!p_1 p_2. P (p_1,p_2)) ==> !p. P p, Thm))] :
  ((string * string) * (thm * class)) list
```

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`.

arb (boolSyntax)

```
arb : term
```

Synopsis

Constant denoting arbitrary items.

Description

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_ss (bossLib)

```
arith_ss : simpset
```

Synopsis

Simplification set for arithmetic.

Description

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)
```

ables and the operators SUC,PRE,+,-,<,>,<=,>=. Multiplication by constants is accomodated by translation to repeated addition. An attempt is made to generalize subformulas 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_CASES_TAC (Tactic)

```
ASM_CASES_TAC : term -> tactic
```

Synopsis

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

Description

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
```

Uses

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, SingleStep.Cases, Cases_on.

ASM_MESON_TAC (mesonLib)

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

Synopsis

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

Description

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_REWRITE_RULE (Rewrite)

```
ASM_REWRITE_RULE : thm list -> thm -> thm
```

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

Description
ASM_REWRITE_RULE rewrites using the tautologies in basic_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, basic_rewrites.

ASM_REWRITE_TAC (Rewrite)

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

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

Description
ASM_REWRITE_TAC generates rewrites with the tautologies in basic_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

basic_rewrites, 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.

ASM_SIMP_RULE (simpLib)

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

Synopsis
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

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

Uses
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)

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

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

Description
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`.

ASM_SIMP_TAC (simpLib)

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

Synopsis
Simplify a term with the given simpset and theorems.

Description
`bossLib.ASM_SIMP_TAC` is identical to `simpLib.ASM_SIMP_TAC`.

See also
`bossLib.ASM_SIMP_TAC`.

assert (Lib)

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

Synopsis
Checks that a value satisfies a predicate.

Description
`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];
> false : bool

- assert null [1];
! Uncaught exception:
! HOL_ERR <poly>
```

See also

`Lib.can`, `Lib.assert_exn`, `Feedback.with_exn`.

assert_exn (Lib)

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

Synopsis

Checks that a value satisfies a predicate.

Description

`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];
> false : bool

- assert_exn null [1] (Fail "non-empty list");;
! Uncaught exception:
! Fail  "non-empty list"
```

See also

`Lib.can`, `Lib.assert`, `Feedback.with_exn`.

assoc (Lib)

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

Synopsis

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

Description

`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)];
> val it = (2, 5) : (int * int)
```

See also

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

assoc1 (Lib)

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

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

Description
`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.find`, `Lib.mem`, `Lib.tryfind`, `Lib.exists`, `Lib.all`.

assoc2 (Lib)

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

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

Description
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.find`, `Lib.mem`, `Lib.tryfind`, `Lib.exists`, `Lib.all`.

associate_restriction (Parse)

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

Synopsis
Associates a restriction semantics with a binder.

Description
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.

Associations between user defined binders and their restrictions are not stored in the theory, so they have to be set up for each hol session (e.g. with a `hol-init.ml` file).

The flag `‘#restrict(Globals.pp_flags)‘` has default `true`, but if set to `false` will disable the pretty printing. This is useful for seeing what the semantics of particular restricted abstractions are.

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_ABSTRACT, RES_FORALL, RES_EXISTS and RES_SELECT are defined in the theory ‘restr_binder‘ by:

```
|- RES_ABSTRACT P B =  \x:'a. (P x => B x | ARB:'b)

|- 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
```

where ARB is defined in the theory ‘restr_binder‘ by:

```
|- ARB  =  @x:'a. T
```

Failure

Never fails.

Example

```
- new_binder_definition("DURING", --‘DURING(p:num#num->bool) = $!p‘--);
  |- !p. $DURING p = $! p

- --‘DURING x::(m,n). p x‘--;

  Exception raised at Parse_support.restr_binder:
  no restriction associated with "DURING"

- new_definition("RES_DURING",
                 --‘RES_DURING(m,n)p = !x. m<=x /\ x<=n ==> p x‘--);

  |- !m n p. RES_DURING (m,n) p = (!x. m <= x /\ x <= n ==> p x) : thm

- associate_restriction("DURING","RES_DURING");
  () : unit

-  --‘DURING x::(m,n). p x‘--;
  (--‘DURING x ::(m,n). p x‘--) : term

- Globals.show_restrict := false;
  () : unit

- --‘DURING x::(m,n). p x‘--;
  (--‘RES_DURING (m,n) (\x. p x)‘--) : term
```

See also

binder_restrictions, delete_restriction.

ASSUM_LIST (Tactical)

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

Synopsis

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

Description

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 ASSUMEd 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.

Uses
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)

```
ASSUME : term -> thm
```

Synopsis
Introduces an assumption.

Description

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_TAC (Tactic)

```
ASSUME_TAC : thm_tactic
```

Synopsis

Adds an assumption to a goal.

Description

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.)

Uses

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)

```
augment_srw_ss : ssdata list -> unit
```

Synopsis
Augments the ”stateful” simpset used by SRW_TAC with a list of simpset fragments.

Description
bossLib.augment_srw_ss is identical to BasicProvers.augment_srw_ss

See also
bossLib.augment_srw_ss.

augment_srw_ss (bossLib)

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

Synopsis
Augments the “stateful rewriter” with a list of simpset fragments.

Description
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)

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

Synopsis
All the axioms stored in the named theory.

Description
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 =
    [("INFINITY_AX", |- ?f. ONE_ONE f /\ ~ONTO f),
     ("SELECT_AX", |- !P x. P x ==> P ($@ P)),
     ("ETA_AX", |- !t. (\x. 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`.

axioms (Theory)

```
axioms : unit -> (string * thm) list
```

Synopsis
Returns the axioms of the current theory.

Description
A call `axioms ()` returns the axioms of the current theory segment together with their names. The names are those given to the axioms by the user when they were originally added to the theory segment (by a call to `new_axiom`).

Failure
Never fails.

See also
`Theory.axiom`, `Theory.definitions`, `Theory.theorems`, `Theory.new_axiom`.

b (goalstackLib)

```
b : unit -> goalstack
```

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

Description
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`.

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

See also
`goalstackLib.backup`, `backup_limit`, `goalstackLib.e`, `goalstackLib.expand`, `goalstackLib.expandf`, `goalstackLib.g`, `get_state`, `goalstackLib.p`, `print_state`, `goalstackLib.r`, `rotate`, `save_top_thm`, `goalstackLib.set_goal`, `set_state`, `goalstackLib.top_goal`, `goalstackLib.top_thm`.

B (Lib)

```
B : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Synopsis

Performs curried function-composition: `B f g x = f (g x)`.

Failure

Never fails.

See also

`Lib`, `Lib.##`, `Lib.A`, `Lib.C`, `Lib.I`, `Lib.K`, `Lib.S`, `Lib.W`.

backup (goalstackLib)

```
backup : unit -> goalstack
```

Synopsis

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

Description

The function `backup` is part of the subgoal package. It allows backing up from the last state change (caused by calls to `expand`, `set_goal`, `rotate` and their abbreviations, or to `set_state`). 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 is one greater than the value of the assignable variable `backup_limit`. This variable is initially set to 12. 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. `backup` is abbreviated by the function `b`. 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: 1 proof.
    1. Incomplete:
         Initial goal:
         (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])

     : proofs

- e CONJ_TAC;
OK..
2 subgoals:
> val it =
    TL [1; 2; 3] = [2; 3]


    HD [1; 2; 3] = 1

     : goalstack

- backup();
> val it =
    Initial goal:

    (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])

     : goalstack

- e (REWRITE_TAC[listTheory.HD, listTheory.TL]);
OK..
> val it =
    Initial goal proved.
    |- (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3]) : goalstack
```

Uses

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

See also

goalstackLib.b, goalstackLib.e, goalstackLib.expand, goalstackLib.expandf, goalstackLib.g, goalstackLib.p, goalstackLib.r, goalstackLib.rotate, goalstackLib.set_goal, goalstackLib.top_goal, goalstackLib.top_thm, goalstackLib.restart, goalstackLib.drop, goalstackLib.dropn.

Beta (Thm)

```
Beta : thm -> thm
```

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

Description
`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 (Term `(K:'a ->'b->'a) x`);
> val th = |- K x = K x : thm

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

- Beta it;
> val it = |- K x = (\y. x) : thm
```

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

See also
`Drule.RIGHT_BETA, Thm.Eta.`

beta (Type)

```
beta : hol_type
```

Synopsis
Common type variable.

Description
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)

```
beta_conv : term -> term
```

Synopsis
Performs one step of beta-reduction.

Description
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 = `\y. P Q` : term

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

Uses
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)

```
BETA_CONV : conv
```

Synopsis

Performs a single step of beta-conversion.

Description

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 = |- (\x. x + 1)y = y + 1 :thm

- BETA_CONV (Term `(\x y. x+y)y`);
> val it = |- (\x y. x + y)y = (\y'. 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)

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

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

Description
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 = Parse.Term `f = ((\x y. x + y) y)`
  in
  val x = ASSUME tm
  end;
val x = [f = (\x y. x + y)y] |- f = (\x y. x + y)y : thm

- BETA_RULE x;
val it = [f = (\x y. x + y)y] |- f = (\y'. y + y') : thm
```

See also
`Thm.BETA_CONV`, `Tactic.BETA_TAC`, `PairedLambda.PAIRED_BETA_CONV`, `Drule.RIGHT_BETA`.

BETA_TAC (Tactic)

```
BETA_TAC : tactic
```

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

Description

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)

```
BINDER_CONV : conv -> conv
```

Synopsis

Applies a conversion underneath a binder.

Description

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 = |- (\x. x + 0 = x) = \x. 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)

```
BINOP_CONV : conv -> conv
```

Synopsis
Applies a conversion to both arguments of a binary operator.

Description
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 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`.

body (Term)

```
body : term -> term
```

Synopsis
Returns the body of an abstraction.

Description
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)

```
BODY_CONJUNCTS : (thm -> thm list)
```

Synopsis
Splits up conjuncts recursively, stripping away universal quantifiers.

Description
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)

```
bool : hol_type
```

Synopsis
Basic type constant.

Description
The ML variable `Type.bool` is bound to the type constant `bool`.

See also
`alpha`, `Type.beta`, `Type.gamma`, `Type.delta`.

bool_case (boolSyntax)

```
bool_case : term
```

Synopsis
Constant denoting case expressions for `bool`.

Description
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)

```
BOOL_CASES_TAC : (term -> tactic)
```

Synopsis
Performs boolean case analysis on a (free) term in the goal.

Description
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[]
```

Uses
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)

```
bool_compset : unit -> compset
```

Synopsis
Creates a new simplification set to use with CBV_CONV for basic computations.

Description
This function creates a new simplification set to 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

```
- CBV_CONV (bool_compset()) (Term `F ==> (T \/ F)`);
> val it = |- F ==> (T \/ F) = T : thm
```

See also
computeLib.CBV_CONV.

bool_EQ_CONV (Conv)

```
bool_EQ_CONV : conv
```

Synopsis
Simplifies expressions involving boolean equality.

Description
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)

```
bool_rewrites: rewrites
```

Synopsis

Contains a number of basic equalities useful in rewriting.

Description

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
```

Uses

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)

bool_ss : simpset

Synopsis
Basic simpset containing standard propositional and first order logic simplifications, plus beta and eta conversion.

Description
BasicProvers.bool_ss is identical to boolSimps.bool_ss.

See also
boolSimps.bool_ss.

bool_ss (boolSimps)

bool_ss : simpset

Synopsis
Basic simpset containing standard propositional and first order logic simplifications, plus beta-conversion.

Description
bossLib.bool_ss is identical to boolSimps.bool_ss.

See also
bossLib.bool_ss.

bool_ss (bossLib)

bool_ss : simpset

Synopsis

Basic simpset containing standard propositional and first order logic simplifications, plus beta conversion.

Description

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 fol-

lowing 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.

Uses
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`.

butlast (Lib)

```
butlast : 'a list -> 'a list
```

Synopsis
Computes the sub-list of a list consisting of all but the last element.

Description
`butlast [x1,...,xn]` returns `[x1,...,x(n-1)]`.

Failure
Fails if the list is empty.

See also
`Lib.last`, `Lib.el`, `Lib.front_last`.

bvar (Term)

```
bvar : term -> term
```

Synopsis
Returns the bound variable of an abstraction.

Description
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`.

by (bossLib)

```
op by : term quotation * tactic -> tactic
```

Synopsis
Prove and place a theorem on the assumptions of the goal.

Description
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 any remaining subgoals generated are added to `A,tm ?- g`.

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`.

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 (Term ‘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`.

Since the tactic supplied need not solve the generated subgoal, `by` gives a useful way of generating proof obligations while pursuing a particular line of reasoning. For example, the above goal could also be attacked by

```
‘?n. y = n + w‘ by ALL_TAC
```

with the result being the goal `{x <= y, w < x} ?- ?n. y = n + w` and the augmented original `{y = n + w, x <= y, w < x} ?- P`. Now either may be attempted.

Comments

Use of `by` can be more convenient than `IMP_RES_TAC` and `RES_TAC` when they would generate many useless assumptions.

See also

`Tactical.SUBGOAL_THEN`, `Tactic.IMP_RES_TAC`, `Tactic.RES_TAC`, `Tactic.STRIP_ASSUME_TAC`.

C (Lib)

```
C : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
```

Synopsis

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.A, Lib.B, Lib.I, Lib.K, Lib.S, Lib.W.

can (Lib)

```
can : ('a -> 'b) -> 'a -> bool
```

Synopsis

Tests for failure.

Description

`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 [];
! Uncaught exception:
! Empty

- can hd [];
> val it = false : bool

- can (fn _ => raise Interrupt) 3;
> Interrupted.
```

See also

Lib.assert, try, Lib.trye, Lib.partial, Lib.total, Feedback.with_exn, Lib.assert_exn.

Cases (bossLib)

```
Cases : tactic
```

Synopsis

Performs case analysis on the variable of the leading universally quantified variable of the goal.

Description

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 (SingleStep)

```
Cases : tactic
```

Synopsis

Case split on leading universally quantified variable in a goal.

Description

`bossLib.Cases` is identical to `SingleStep.Cases`.

See also

`bossLib.Cases`.

Cases_on (bossLib)

```
Cases_on : term -> tactic
```

Synopsis
Performs case analysis on the type of a given term.

Description
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_on (SingleStep)

```
Cases_on : term -> tactic
```

Synopsis
Case split on type of supplied term.

Description
`bossLib.Cases_on` is identical to `SingleStep.Cases_on`.

See also
`bossLib.Cases_on`.

CASES_THENL (Thm_cont)

```
CASES_THENL : (thm_tactic list -> thm_tactic)
```

Synopsis
Applies the theorem-tactics in a list to corresponding disjuncts in a theorem.

Description
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.

Uses
Performing very general disjunctive case splits.

See also
`Thm_cont.DISJ_CASES_THENL`, `Thm_cont.X_CASES_THENL`.

CBV_CONV (computeLib)

```
CBV_CONV : comp_rws -> conv
```

Synopsis
Call by value rewriting.

Description
The conversion CBV_CONV expects an 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_rws will create a new set containing only reflexivity (REFL_CLAUSE). Theorems can be added to a set with the function add_thms. The function from_list simply combines new_rws and add_thms.

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, `initial_rws` provides a way to create a new simplification set with all the rules needed for basic boolean and arithmetical calculations built in.

Example

```
- val rws = from_list (lazyfy_thm [COND_CLAUSES]);
> val rws = RWS<hash_table> : comp_rws

- 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

- CBV_CONV (initial_rws())
           (--‘if 100 - 5 * 5 < 80  then 2 EXP 16 else 3‘--);
> val it = |- (if 100 - 5 * 5 < 80 then 2 EXP 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 rws = bossLib.initial_rws();
val _ = add_thms(true,[eqn]) rws;

- CBV_CONV rws (--‘exp 2 n‘--);
> Interrupted.
- set_skip rws "COND" (SOME 1);
> val it = () : unit
- 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 = --‘\(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

`REDUCE_CONV`, `reduce_rws`, `computeLib.initial_rws`.

CCONTR (Thm)

```
CCONTR : term -> thm -> thm
```

Synopsis

Implements the classical contradiction rule.

Description

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)

```
CCONTR_TAC : tactic
```

Synopsis
Prepares for a proof by Classical contradiction.

Description
`CCONTR_TAC` takes a theorem `A' |- F` and completely solves the goal. This is an invalid tactic unless `A'` is a subset of `A`.

```
 A ?- t
========  CCONTR_TAC (A' |- F)
```

Failure
Fails unless the theorem is contradictory, i.e. has `F` as its conclusion.

See also
`Tactic.CHECK_ASSUME_TAC, Thm.CCONTR, CCCONTR, Drule.CONTRAPOS, Thm.NOT_ELIM.`

CHANGED_CONV (Conv)

```
CHANGED_CONV : (conv -> conv)
```

Synopsis
Makes a conversion fail if applying it leaves a term unchanged.

Description
If `c` is a conversion that maps a term `"t"` to a theorem `|- t = t'`, where `t'` is alpha-equivalent 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).

Failure
`CHANGED_CONV c "t"` fails if `c` maps `"t"` to `|- t = t'`, where `t'` is alpha-equivalent 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'`).

Uses
`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.

CHANGED_TAC (Tactical)

```
CHANGED_TAC : (tactic -> tactic)
```

Synopsis
Makes a tactic fail if it has no effect.

Description
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, VALID.

CHECK_ASSUME_TAC (Tactic)

```
CHECK_ASSUME_TAC : thm_tactic
```

Synopsis
Adds a theorem to the assumption list of goal, unless it solves the goal.

Description
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)

```
CHOOSE : term * thm -> thm -> thm
```

Synopsis
Eliminates existential quantification using deduction from a particular witness.

Description
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)

```
CHOOSE_TAC : thm_tactic
```

Synopsis
Adds the body of an existentially quantified theorem to the assumptions of a goal.

Description
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 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
```

See also
`Thm_cont.CHOOSE_THEN`, `Tactic.X_CHOOSE_TAC`.

CHOOSE_THEN (Thm_cont)

```
CHOOSE_THEN : thm_tactical
```

Synopsis
Applies a tactic generated from the body of existentially quantified theorem.

Description

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]
```

See also

`Tactic.CHOOSE_TAC`, `Thm_cont.X_CHOOSE_THEN`.

class (DB)

```
datatype class
```

Synopsis
Datatype for classifying theory elements.

Description
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_on (Parse)

```
Parse.clear_overloads_on : string -> unit
```

Synopsis
Clears all overloading on the specified operator.

Description
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

```
- load "realTheory";
> val it = () : unit
- realTheory.REAL_INV_LT1;
> val it = |- !x. 0 < x /\ x < 1 ==> 1 < inv x : thm
- clear_overloads_on "<";
> val it = () : unit
- realTheory.REAL_INV_LT1;
> val it = |- !x. 0 real_lt x /\ x real_lt 1 ==> 1 real_lt inv x : thm
- clear_overloads_on "&";
> val it = () : unit
- realTheory.REAL_INV_LT1;
> val it = |- !x. 0r real_lt x /\ x real_lt 1r ==> 1r real_lt inv x : thm
```

Uses

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`.

clear_prefs_for_term (Parse)

```
Parse.clear_prefs_for_term : string -> unit
```

Synopsis

Removes pretty-printing preference information from the global grammar.

Description

The `clear_prefs_for_term` function removes the information stored in the global grammar as to which (if any) rule should be preferred when terms are pretty-printed. This will cause terms of the given name to be printed using “raw” syntax.

Failure

Never fails.

Example

The initial grammar has two rules for conditional expressions, with the `if-then-else` form preferred, so that even if the old HOL88 style syntax is used for input, the term is printed out in the `if-then-else` style:

```
- Term‘p => q | r‘;
<<HOL message: inventing new type variable names: ’a.>>
> val it = ‘(if p then q else r)‘ : term
```

If `clear_prefs_for_term` is applied, neither syntax will print:

```
- clear_prefs_for_term "COND";
> val it = () : unit
- Term‘p => q | r‘;
<<HOL message: inventing new type variable names: ’a.>>
> val it = ‘COND p q r‘ : term
```

See also

`Parse.prefer_form_with_tok`.

CNF_CONV (normalForms)

```
CNF_CONV : conv
```

Synopsis

Converts a formula into Conjunctive Normal Form (CNF).

Description

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

```
- 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 x'. Q (y x) \/ ~P x' \/ ~Q x' \/ ~P x : thm
```

Example

```
- CNF_CONV ‘‘~(~(x = y) = z) = ~(x = ~(y = z))‘‘;
> val it = |- (~(~(x = y) = z) = ~(x = ~(y = z))) = T : thm
```

combine (Lib)

```
combine : 'a list * 'b list -> ('a * 'b) list
```

Synopsis
Transforms a pair of lists into a list of pairs.

Description
`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)

```
commafy : string list -> string list
```

Synopsis
Add commas into a list of strings.

Description
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

- print (String.concat it ^ "\n");
donkey, mule, horse, camel, llama
> val it = () : unit

- commafy ["foo"];
> val it = ["foo"] : string list
```

compare (Term)

```
Term.compare : term * term -> order
```

Synopsis
Ordering on terms.

Description
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
```

Synopsis
An ordering on HOL types.

Description
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)

```
completeInduct_on : term quotation -> tactic
```

Synopsis
Perform complete induction

Description
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`.

concat (Lib)

```
concat : string -> string -> string
```

Synopsis
Concatenates two ML strings.

Failure
Never fails.

Example
```
- concat "1" "";
> val it = "1" : string

- concat "hello" "world";
> val it = "helloworld" : string

- concat "hello" (concat " " "world");
> val it = "hello world" : string
```

Comments
This function is open at the top level and is not the same as the Basis function `String.concat`. The latter concatenates a list of strings.

concl (Thm)

```
concl : thm -> term
```

Synopsis
Returns the conclusion of a theorem.

Description
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)

```
COND_CASES_TAC : tactic
```

Synopsis
Induces a case split on a conditional expression in the goal.

Description
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)]   A u {~p} ?- t[v/(p=>u|v)]]
```

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
```

Uses
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)

```
COND_CONV : conv
```

Synopsis
Simplifies conditional terms.

Description
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.

conditional (boolSyntax)

```
conditional : term
```

Synopsis
Constant denoting conditional expressions.

Description
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`.

CONJ (Thm)

```
CONJ : thm -> thm -> thm
```

Synopsis
Introduces a conjunction.

Description

```
   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_DISCH (Drule)

```
CONJ_DISCH : (term -> thm -> thm)
```

Synopsis
Discharges an assumption and conjoins it to both sides of an equation.

Description
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)

```
CONJ_DISCHL : (term list -> thm -> thm)
```

Synopsis

Conjoins multiple assumptions to both sides of an equation.

Description

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_LIST (Drule)

```
CONJ_LIST : (int -> thm -> thm list)
```

Synopsis
Extracts a list of conjuncts from a theorem (non-flattening version).

Description
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)

```
CONJ_PAIR : thm -> thm * thm
```

Synopsis
Extracts both conjuncts of a conjunction.

Description

```
      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_SET_CONV (Drule)

```
CONJ_SET_CONV : (term list -> term list -> thm)
```

Synopsis
Proves the equivalence of the conjunctions of two equal sets of terms.

Description
The arguments to CONJ_SET_CONV are two ML lists of terms [t1,...,tn] and [u1,...,um]. If these are equal when considered as sets, that is if the sets

```
{t1,...,tn} and {u1,...,um}
```

are equal, then CONJ_SET_CONV returns the theorem:

```
|- (t1 /\ ... /\ tn) = (u1 /\ ... /\ um)
```

Otherwise CONJ_SET_CONV fails.

Failure
`CONJ_SET_CONV [t1,...,tn] [u1,...,um]` fails if `[t1,...,tn]` and `[u1,...,um]`, regarded as sets of terms, are not equal. Also fails if any `ti` or `ui` does not have type `bool`.

Uses
Used to order conjuncts. First sort a list of conjuncts `l1` into the desired order to get a new list `l2`, then call `CONJ_SET_CONV l1 l2`.

Comments
This is not a true conversion, so perhaps it ought to be called something else.

See also
`Drule.CONJUNCTS_CONV`.

CONJ_TAC (Tactic)

```
CONJ_TAC : tactic
```

Synopsis
Reduces a conjunctive goal to two separate subgoals.

Description
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`.

CONJUNCT1 (Thm)

```
CONJUNCT1 : thm -> thm
```

Synopsis
Extracts left conjunct of theorem.

Description

```
 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)

```
CONJUNCT2 : thm -> thm
```

Synopsis
Extracts right conjunct of theorem.

Description

```
 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)

```
conjunction : term
```

Synopsis
Constant denoting logical conjunction.

Description
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)

```
CONJUNCTS : (thm -> thm list)
```

Synopsis
Recursively splits conjunctions into a list of conjuncts.

Description
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_CONV (Drule)

```
CONJUNCTS_CONV : term * term -> thm
```

Synopsis

Prove equivalence under idempotence, symmetry and associativity of conjunction.

Description

CONJUNCTS_CONV 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_CONV (t1,t2) returns |- t1 = t2. Otherwise, it fails.

Failure

Fails if t1 and t2 are not equivalent, as described above.

Example

```
- CONJUNCTS_CONV (Term `(P /\ Q) /\ R`, Term `R /\ (Q /\ R) /\ P`);
> val it = |- (P /\ Q) /\ R = R /\ (Q /\ R) /\ P : thm
```

Uses

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_CONV(t1,t2).

Comments

This is not a true conversion, so perhaps it ought to be called something else.

See also

`Drule.CONJ_SET_CONV`.

CONJUNCTS_THEN (Thm_cont)

```
CONJUNCTS_THEN : thm_tactical
```

Synopsis

Applies a theorem-tactic to each conjunct of a theorem.

Description

`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:

```
let CONJUNCTS_THENL (f:thm_tactic) thm =
      itlist $THEN (map f (CONJUNCTS thm)) ALL_TAC;;
```

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)

```
CONJUNCTS_THEN2 : (thm_tactic -> thm_tactic -> thm_tactic)
```

Synopsis

Applies two theorem-tactics to the corresponding conjuncts of a theorem.

Description

`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)`.

Uses

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)

```
cons : 'a -> 'a list -> 'a list
```

Synopsis

Curried form of list cons operation

Description

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
```

constants (Theory)

```
constants : string -> term list
```

Synopsis
Returns a list of the constants defined in a named theory.

Description
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 = [‘$o‘, ‘W‘, ‘S‘, ‘K‘, ‘I‘, ‘combin$C‘] : term list
```

See also
`Theory.types`, `Theory.current_axioms`, `Theory.current_definitions`, `Theory.current_theorems`.

CONTR (Drule)

```
CONTR : term -> thm -> thm
```

Synopsis
Implements the intuitionistic contradiction rule.

Description
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)

```
CONTR_TAC : thm_tactic
```

Synopsis
Solves any goal from contradictory theorem.

Description
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)

```
CONTRAPOS : (thm -> thm)
```

Synopsis
Deduces the contrapositive of an implication.

Description
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)

```
CONTRAPOS_CONV : conv
```

Synopsis
Proves the equivalence of an implication and its contrapositive.

Description
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_RULE (Conv)

```
CONV_RULE : (conv -> thm -> thm)
```

Synopsis
Makes an inference rule from a conversion.

Description
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.

Failure
CONV_RULE c th fails if c fails 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
Tactic.CONV_TAC, Conv.RIGHT_CONV_RULE.

CONV_TAC (Tactic)

```
CONV_TAC : (conv -> tactic)
```

Synopsis
Makes a tactic from a conversion.

Description

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'
```

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 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'`).

Uses

`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

`Conv.CONV_RULE`.

current_axioms (Theory)

```
current_axioms : unit -> (string * thm) list
```

Synopsis
Return the axioms in the current theory segment.

Description
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)

```
current_definitions : unit -> (string * thm) list
```

Synopsis
Return the definitions in the current theory segment.

Description
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_defs (Theory)

```
current_defs : unit -> (string * thm) list
```

Synopsis
Return the definitions in the current theory segment.

Description
An invocation `current_defs ()` returns a list of the definitions made in the current theory segment.

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_thms`, `Theory.constants`, `Theory.types`, `Theory.parents`.

current_theorems (Theory)

```
current_theorems : unit -> (string * thm) list
```

Synopsis
Return the theorems stored in the current theory segment.

Description
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)

```
current_theory : unit -> string
```

Synopsis

Returns the name of the current theory segment.

Description

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 = "scratch" : string

- new_theory "foo";
<<HOL message: Created theory "foo">>
> val it = () : unit

- current_theory();
> val it = "foo" : string
```

Failure
Never fails.

See also
`Theory.new_theory`, `Theory.export_theory`.

current_thms (Theory)

```
current_thms : unit -> (string * thm) list
```

Synopsis
Return the theorems stored in the current theory segment.

Description
An invocation `current_thms ()` returns a list of the theorems that have been 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_defs`, `Theory.current_thms`, `Theory.constants`, `Theory.types`, `Theory.parents`.

current_trace (Feedback)

```
current_trace : string -> int
```

Synopsis
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)

```
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

Synopsis
Converts a function on a pair to a corresponding curried function.

Description
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 it = increment = fn : int -> int

- increment 6;
> val it = 7 : int
```

See also
`Lib`, `Lib.uncurry`.

CURRY_CONV (PairRules)

```
CURRY_CONV : conv
```

Synopsis
Currys an application of a paired abstraction.

Example

```
- CURRY_CONV (Term `(\(x,y). x + y) (1,2)`);
> val it = |- (\(x,y). x + y) (1,2) = (\x y. x + y) 1 2 : thm

- CURRY_CONV (Term `(\(x,y). x + y) z`);
> val it = |- (\(x,y). x + y) z = (\x 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)

```
CURRY_EXISTS_CONV : conv
```

Synopsis
Currys paired existential quantifications into consecutive existential quantifications.

Example

```
- 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

- 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)

```
CURRY_FORALL_CONV : conv
```

Synopsis
Currys paired universal quantifications into consecutive universal quantifications.

Example

```
- 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

- 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`.

data (DB)

```
type data
```

Synopsis
Type abbreviation used in `DB` structure.

Description
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))]
   : ((string * string) * (thm * class)) list
```

See also

DB.class, DB.thy, DB.find, DB.match, DB.apropos, DB.listDB.

DECIDE (bossLib)

```
DECIDE : term -> thm
```

Synopsis

Invoke decision procedure(s).

Description

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, intLib.ARITH_TAC.

DECIDE_TAC (bossLib)

```
DECIDE_TAC : tactic
```

Synopsis
Invoke decision procedure(s).

Description
DECIDE_TAC is the tactical version of DECIDE.

Failure
As for DECIDE

See also
bossLib.DECIDE.

decls (Term)

```
decls : string -> term list
```

Synopsis
Returns a list of constants having the same name.

Description
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`}] : ...
```

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)

```
decls : string -> {Thy : string, Tyop : string} list
```

Synopsis

Lists all theories a named type operator is declared in.

Description

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`.

Define (bossLib)

```
Define : term quotation -> thm
```

Synopsis

General-purpose function definition facility.

Description

`Define` takes a high-level specification of an 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 inter-

mediate 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 `tprove`.

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 gen-

erated 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.xDefine`, `TotalDefn.DefineSchema`, `bossLib.Hol_defn`, `Defn.tgoal`, `Defn.tprove`, `bossLib.WF_REL_TAC`, `bossLib.recInduct`, `bossLib.EVAL`, `bossLib.EVAL_TAC`.

Define (TotalDefn)

```
Define : term quotation -> thm
```

Synopsis
General purpose function definition facility.

Description
`bossLib.Define` is identical to `TotalDefn.Define`.

See also
`bossLib.Define`.

define_new_type_bijections (Drule)

```
define_new_type_bijections :
  {name:string, ABS:string, REP:string, tyax:thm} -> thm
```

Synopsis
Introduces abstraction and representation functions for a defined type.

Description
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`, `Prim_rec.prove_abs_fn_one_one`, `Prim_rec.prove_abs_fn_onto`, `Drule.prove_rep_fn_one_one`, `Drule.prove_rep_fn_onto`.

define_type (Define_type)

```
define_type : {name :string, type_spec :term frag list,
               fixities : fixity list} -> thm
```

Synopsis

Automatically defines a user-specified concrete recursive data type.

Description

This function is deprecated, and its use is not recommended. The following documentation is old and not likely to be updated as the system further evolves. The more flexible and powerful `Hol_datatype` function should be used instead of `define_type`.

The ML function `define_type` automatically defines any required concrete recursive type in the logic. The `name` argument is the name under which the results of making the definition will be stored in the current theory segment. The `type_spec` argument is a user-supplied specification of the type to be defined. This specification (explained below) simply states the names of the new type's constructors and the logical types of their arguments. The `fixities` argument gives the parsing status of the introduced constants: it may be `Prefix`, `Binder`, or `Infix <positive int>`. The theorem returned by `define_type` is an automatically-proved abstract characterization of the concrete data type described by this specification.

The `type_spec` argument to `define_type` must be a quotation of the form:

```
‘op = C1 of ty => ... => ty | C2 of ty=> ...=>ty | ... | Cn of ty=> ... =>ty‘
```

where `op` is the name of the type constant or type operator to be defined, `C1`, ..., `Cn` are identifiers, and each `ty` is either a (logical) type expression valid in the current theory (in which case `ty` must not contain `op`) or just the identifier '`op`' itself.

A quotation of this form describes an n-ary type operator `op`, where n is the number of distinct type variables in the types `ty` on the right hand side of the equation. If n is zero then `op` is a type constant; otherwise `op` is an n-ary type operator. The type described by the specification has `n` distinct constructors `C1`, ..., `Cn`. Each constructor `Ci` is a function that takes arguments whose types are given by the associated type expressions `ty` in the specification. If one or more of the type expressions `ty` is the type `op` itself, then the equation specifies a recursive data type. In any specification, at least one constructor must be non-recursive, i.e. all its arguments must have types which already exist in the current theory.

Given a type specification of the form described above, `define_type` makes an appropriate type definition for the type operator `op`. It then makes appropriate definitions for the constants `C1`, ..., `Cn`, and automatically proves a theorem that states an abstract characterization of the newly-defined type `op`. This theorem, which is stored in the current theory segment under the name supplied as the first argument and also returned by `define_type`, has the form of a 'primitive recursion theorem' for the concrete type `op` (see the examples given below). This property provides an abstract characterization of the type `op` which is both succinct and complete, in the sense that it completely determines the structure of the values of `op` up to isomorphism.

Failure

Evaluating

```
define_type{type_spec = ‘op = C1 of ty=>...=>ty | ... | Cn of ty=>...=>ty‘,
            name, fixities}
```

fails if the supplied constant names `C1`, ..., `Cn` are not distinct; if any one of `C1`, ..., `Cn`

is already a constant in the current theory or is not an allowed name for a constant; if ABS_op or REP_op are already constants in the current theory; if there is already an axiom, definition, constant specification or type definition stored under either the name op_TY_DEF or the name op_ISO_DEF in the current theory segment; or (finally) if the input type specification does not conform in any other respect to the syntax described above.

Example

The following call to define_type defines tri to be a simple enumerated type with exactly three distinct values:

```
- define_type{name = "tri_DEF",
              type_spec = ‘tri = ONE | TWO | THREE‘,
              fixities = [Prefix,Prefix,Prefix]}
|- !e0 e1 e2. ?! fn. (fn ONE = e0) /\ (fn TWO = e1) /\ (fn THREE = e2)
```

The theorem returned is a degenerate ‘primitive recursion’ theorem for the concrete type tri. An example of a recursive type that can be defined using define_type is a type of binary trees:

```
- define_type {type_spec = ‘btree = LEAF of ’a
                                   | NODE of btree => btree‘,
               name = "tree_DEF",
               fixities = [Prefix,Prefix]}
|- !f0 f1.
     ?! fn.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

The theorem returned by define_type in this case asserts the unique existence of functions defined by primitive recursion over labelled binary trees.

Note that the type being defined may not occur as a proper subtype in any of the types of the arguments of the constructors:

```
- define_type{type_spec = ‘ty = NUM of num | FUN of (ty -> ty)‘,
              name = "num_funcs", fixities = [Prefix, Prefix]};

Exception raised at Term.make_type_clause.check:
recursive occurrence of defined type is deeper than the first level
```

In this example, there is an error because ty occurs within the type expression (ty -> ty).

Comments

The ”=>” that may be used in type specifications is merely a delimiter that shows a constructor to be Curried. It must occur at the ”top-level” in the argument list to a

constructor. i.e., parsing of the type specification will fail if the "=>" occurs underneath an existing type constructor.

This function is deprecated. `Datatype.Hol_datatype` should be used instead. Types defined with `define_type` are not introduced into the `TypeBase`, and `define_type` doesn't handle nested or mutual recursion.

See also

`Datatype.Hol_datatype`, `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_induction_thm`, `Prim_rec.prove_rec_fn_exists`.

DefineSchema (TotalDefn)

```
DefineSchema : term quotation -> thm
```

Synopsis

Defines a recursion schema

Description

`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 schmatic 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 scheme 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.xDefineSchema, TotalDefn.Define, Defn.Hol_defn.`

definitions (DB)

```
definitions : string -> (string * thm) list
```

Synopsis
All the definitions stored in the named theory.

Description
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 =
    [("C_DEF", |- combin$C = (\f x y. f y x)),
     ("I_DEF", |- I = S K K),
     ("K_DEF", |- K = (\x y. x)),
     ("o_DEF", |- !f g. f o g = (\x. f (g x))),
     ("S_DEF", |- S = (\f g x. f x (g x))),
     ("W_DEF", |- W = (\f x. f x x))] : (string * thm) list
```

See also
`DB.thy, DB.fetch, DB.thms, DB.theorems, DB.axioms, DB.listDB.`

delete_binding (Theory)

```
delete_binding : string -> unit
```

Synopsis

Remove a stored value from the current theory segment.

Description

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)‘;
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

- current_theorems();
> val it =
    [("fact_def", |- fact x = (if x = 0 then 1 else x * fact (x - 1))),
     ("fact_ind", |- !P. (!x. (~(x = 0) ==> P (x - 1)) ==> P x) ==> !v. P v)]
  : (string * thm) list

- delete_binding "fact_ind";
> val it = () : unit

- current_theorems();
> val it =
    [("fact_def", |- 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)

```
delete_const : string -> unit
```

Synopsis

Remove a term constant from the current signature.

Description

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)‘;
Equations stored under "foo_def".
Induction stored under "foo_ind".
> val it = |- 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 = |- scratch$old->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.delete_axiom`, `Theory.delete_definition`, `Theory.delete_theorem`, `Theory.uptodate_type`, `Theory.uptodate_term`, `Theory.uptodate_thm`, `Theory.scrub`.

delete_type (Theory)

```
delete_type : string -> unit
```

Synopsis

Remove a type operator from the signature.

Description

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 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;
Exception raised at type_pp.pp_type:
old->foo<-old: no such type operator in grammar
```

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.delete_axiom`, `Theory.delete_definition`, `Theory.delete_theorem`, `Theory.uptodate_type`, `Theory.uptodate_term`, `Theory.uptodate_thm`, `Theory.scrub`.

delta (Lib)

```
type 'a delta
```

Synopsis

A type used for telling when a function has changed its argument.

Description

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)

```
delta : hol_type
```

Synopsis
Common type variable.

Description
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)

```
delta_apply : ('a -> 'a delta) -> 'a -> 'a
```

Synopsis
Apply a function to an argument, re-using the argument if possible.

Description
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)

```
delta_map : ('a -> 'a delta) -> 'a list -> 'a list delta
```

Synopsis

Apply a function to a list, sharing as much structure as possible.

Description
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)

```
delta_pair : ('a -> 'a delta) ->
             ('b -> 'b delta) ->
             'a * 'b -> ('a * 'b) delta
```

Synopsis
Apply two functions to the projections of a pair, sharing as much structure as possible.

Description
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.`

deprecate_int (intLib)

```
intLib.deprecate_int : unit -> unit
```

Synopsis
Makes the parser never consider integers as a numeric possibility.

Description
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

`numLib.deprecate_num`, `intLib.prefer_int`, `numLib.prefer_num`.

DEPTH_CONV (Conv)

```
DEPTH_CONV : conv -> conv
```

Synopsis

Applies a conversion repeatedly to all the sub-terms of a term, in bottom-up order.

Description

`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).

Uses

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.

dest_abs (Term)

```
dest_abs : term -> term * term
```

Synopsis
Breaks apart an abstraction into abstracted variable and body.

Description
`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.strip_abs.

dest_arb (boolSyntax)

```
dest_arb : term -> hol_type
```

Synopsis
Extract the type of an instance of the `ARB` constant.

Description
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)

```
dest_bool_case : term -> term * term * term
```

Synopsis
Destructs a case expression over bool.

Description
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)

```
dest_comb : term -> term * term
```

Synopsis
Breaks apart a combination (function application) into rator and rand.

Description
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)

```
dest_cond : term -> term * term * term
```

Synopsis
Breaks apart a conditional into the three terms involved.

Description
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)

```
dest_conj : term -> term * term
```

Synopsis
Term destructor for conjunctions.

Description
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)

```
dest_cons : term -> term * term
```

Synopsis
Breaks apart a 'CONS pair' into head and tail.

Description
`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)

```
dest_const : term -> string * hol_type
```

Synopsis
Breaks apart a constant into name and type.

Description
`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)

```
dest_disj : term -> term * term
```

Synopsis
Term destructor for disjunctions.

Description
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)

```
dest_eq : term -> term * term
```

Synopsis
Term destructor for equality.

Description
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)

```
dest_eq_ty : term -> term * term * hol_type
```

Synopsis
Term destructor for equality.

Description
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.

Uses
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)

```
dest_exists : term -> term * term
```

Synopsis
Breaks apart a existentially quantified term into quantified variable and body.

Description
If `M` has the form `?x. t`, then `dest_exists M` returns `(x,t)`.

Failure
Fails if `M` is not a existential quantification.

See also
`boolSyntax.mk_exists`, `boolSyntax.is_exists`, `boolSyntax.strip_exists`.

dest_exists1 (boolSyntax)

```
dest_exists1 : term -> term * term
```

Synopsis
Breaks apart a unique existence term into quantified variable and body.

Description
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)

```
dest_forall : term -> term * term
```

Synopsis
Breaks apart a universally quantified term into quantified variable and body.

Description
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)

```
dest_imp : term -> term * term
```

Synopsis
Breaks an implication or negation into antecedent and consequent.

Description
`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)

```
dest_imp_only : term -> term * term
```

Synopsis
Breaks an implication into antecedent and consequent.

Description
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)

```
dest_let : term -> term * term
```

Synopsis
Breaks apart a let-expression.

Description
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`);
> val it = (`\x. x \/ x`, `P /\ Q`) : term * term
```

Failure
Fails if `M` is not of the form `LET M N`.

See also
`boolSyntax.mk_let`, `boolSyntax.is_let`.

dest_list (listSyntax)

```
dest_list : term -> term list * hol_type
```

Synopsis
Iteratively breaks apart a list term.

Description
`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)

```
dest_neg : term -> term
```

Synopsis
Breaks apart a negation, returning its body.

Description
`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_pabs (pairSyntax)

```
dest_pabs : term -> term * term
```

Synopsis
Breaks apart a paired abstraction into abstracted pair and body.

Description
`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)

```
dest_pair : term -> term * term
```

Synopsis
Breaks apart a pair into two separate terms.

Description
`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)

```
dest_pexists : term -> term * term
```

Synopsis
Breaks apart paired existential quantifiers into the bound pair and the body.

Description
`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.mk_pexists`, `pairSyntax.is_pexists`, `pairSyntax.strip_pexists`.

dest_pforall (pairSyntax)

```
dest_pforall : term -> term * term
```

Synopsis
Breaks apart paired universal quantifiers into the bound pair and the body.

Description
`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.mk_pforall`, `pairSyntax.is_pforall`, `pairSyntax.strip_pforall`.

dest_prod (pairSyntax)

```
dest_prod : hol_type -> hol_type * hol_type
```

Synopsis
Breaks a product type into its two component types.

Description
`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)

```
dest_pselect : term -> term * term
```

Synopsis
Breaks apart a paired choice-term into the selected pair and the body.

Description
`dest_pselect` is a term destructor for paired choice terms. The application of `dest_select` 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.mk_pselect`, `pairSyntax.is_pselect`.

dest_res_abstract (res_quanLib)

```
dest_res_abstract : term -> (term # term # term)
```

Synopsis
Breaks apart a restricted abstract term into the quantified variable, predicate and body.

Description
`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)

```
dest_res_exists : term -> (term # term # term)
```

Synopsis
Breaks apart a restricted existentially quantified term into the quantified variable, predicate and body.

Description
`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)

```
dest_res_exists_unique : term -> (term # term # term)
```

Synopsis

Breaks apart a restricted unique existential quantified term into the quantified variable, predicate and body.

Description

`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)

```
dest_res_forall : term -> (term # term # term)
```

Synopsis

Breaks apart a restricted universally quantified term into the quantified variable, predicate and body.

Description

`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)

```
dest_res_select : term -> (term # term # term)
```

Synopsis

Breaks apart a restricted choice quantified term into the quantified variable, predicate and body.

Description

`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)

```
dest_select : term -> term * term
```

Synopsis
Breaks apart a choice term into selected variable and body.

Description
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_theory (DB)

```
dest_theory : string -> theory
```

Synopsis
Return the contents of a theory.

Description
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:
        sum
        one

    Type constants:
        option 1

    Term constants:
        option_case    :'b -> ('a -> 'b) -> 'a option -> 'b
        NONE    :'a option
        SOME    :'a -> 'a option
        IS_NONE    :'a option -> bool
        option_ABS    :'a + one -> 'a option
        IS_SOME    :'a option -> bool
        option_REP    :'a option -> 'a + one
        THE    :'a option -> 'a
        OPTION_JOIN    :'a option option -> 'a option
        OPTION_MAP    :('a -> 'b) -> 'a option -> 'b option

    Definitions:
        option_TY_DEF  |- ?rep. TYPE_DEFINITION (\x. T) rep
        option_REP_ABS_DEF
        |- (!a. option_ABS (option_REP a) = a) /\
           !r. (\x. T) r = (option_REP (option_ABS r) = r)
        SOME_DEF  |- !x. SOME x = option_ABS (INL x)
        NONE_DEF  |- NONE = option_ABS (INR ())
        option_case_def
        |- (!u f. case u f NONE = u) /\ !u f x. case u f (SOME x) = f x
        OPTION_MAP_DEF
        |- (!f x. OPTION_MAP f (SOME x) = SOME (f x)) /\
           !f. OPTION_MAP f NONE = NONE
        IS_SOME_DEF  |- (!x. IS_SOME (SOME x) = T) /\ (IS_SOME NONE = F)
        IS_NONE_DEF  |- (!x. IS_NONE (SOME x) = F) /\ (IS_NONE NONE = T)
        THE_DEF  |- !x. THE (SOME x) = x
        OPTION_JOIN_DEF
        |- (OPTION_JOIN NONE = NONE) /\ !x. OPTION_JOIN (SOME x) = x

    Theorems:
        option_Axiom  |- !e f. ?fn. (!x. fn (SOME x) = f x) /\ (fn NONE = e)
        option_induction  |- !P. P NONE /\ (!a. P (SOME a)) ==> !x. P x
        SOME_11  |- !x y. (SOME x = SOME y) = (x = y)
        NOT_NONE_SOME  |- !x. ~(NONE = SOME x)
        NOT_SOME_NONE  |- !x. ~(SOME x = NONE)
        option_nchotomy  |- !opt. (opt = NONE) \/ ?x. opt = SOME x
        option_CLAUSES
        |- (!x y. (SOME x = SOME y) = (x = y)) /\ (!x. THE (SOME x) = x) /\
           (!x. ~(NONE = SOME x)) /\ (!x. ~(SOME x = NONE)) /\
           (!x. IS_SOME (SOME x) = T) /\ (IS_SOME NONE = F) /\
```

via pattern matching.

See also
`DB.print_theory`, `Hol_pp.pp_theory`, `Hol_pp.theory`.

dest_thm (Thm)

```
dest_thm : thm -> term list * term
```

Synopsis
Breaks a theorem into assumption list and conclusion.

Description
`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)

```
dest_thy_const : term -> {Thy:string, Name:string, Ty:hol_type}
```

Synopsis
Breaks apart a constant into name, theory, and type.

Description
`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)

```
dest_thy_type
    : hol_type -> {Thy:string, Name:string,
                   Args:hol_type list}
```

Synopsis
Breaks apart a type (other than a variable type).

Description
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 (alpha --> bool);
> val it = {Args = [‘:’a‘, ‘:bool‘], Thy = "min", Tyop = "fun"} :

- try dest_thy_type alpha;

Exception raised at Type.dest_thy_type:
```

See also
`Type.mk_thy_type`, `Type.dest_type`, `Type.mk_type`, `Term.mk_thy_const`.

dest_type (Type)

```
dest_type : hol_type -> string * hol_type list
```

Synopsis
Breaks apart a non-variable type.

Description
If `ty` is a type constant, then `dest_type t` 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", [‘:’a‘, ‘: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)

```
dest_var : term -> string * hol_type
```

Synopsis
Breaks apart a variable into name and type.

Description
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)

```
dest_vartype : hol_type -> string
```

Synopsis
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 raised at Type.dest_vartype:
not a type variable
```

See also
`Type.mk_vartype`, `Type.is_vartype`, `Type.dest_type`.

DISCARD_TAC (Tactic)

```
DISCARD_TAC : thm_tactic
```

Synopsis
Discards a theorem already present in a goal's assumptions.

Description
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)

```
disch : ((term * term list) -> term list)
```

Synopsis
Removes those elements of a list of terms that are alpha equivalent to a given term.

Description
Given a pair `("t",tl)`, `disch` removes those elements of `tl` that are alpha equivalent to `"t"`.

Example

```
disch (Term`\x:bool.T`, [Term`A = T`,Term`B = 3`,Term`\y:bool.T`]);
[`A = T`,`B = 3`] : term list
```

See also
`Lib.filter`.

DISCH (Thm)

```
DISCH : (term -> thm -> thm)
```

Synopsis
Discharges an assumption.

Description

```
       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`, `Thm_cont.FILTER_DISCH_THEN`, `Drule.NEG_DISCH`, `Tactic.STRIP_TAC`, `Drule.UNDISCH`, `Drule.UNDISCH_ALL`, `Tactic.UNDISCH_TAC`.

DISCH_ALL (Drule)

```
DISCH_ALL : (thm -> thm)
```

Synopsis
Discharges all hypotheses of a theorem.

Description

```
        A1, ..., An |- t
  ---------------------------  DISCH_ALL
   |- A1 ==> ... ==> An ==> t
```

Failure
`DISCH_ALL` will not fail 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. Two or more alpha-convertible hypotheses will be discharged by a single implication; users should not rely on which hypothesis appears in the implication.

See also
Thm.DISCH, Tactic.DISCH_TAC, Thm_cont.DISCH_THEN, Drule.NEG_DISCH, Tactic.FILTER_DISCH_TAC, Thm_cont.FILTER_DISCH_THEN, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC.

DISCH_TAC (Tactic)

```
DISCH_TAC : tactic
```

Synopsis
Moves the antecedent of an implicative goal into the assumptions.

Description

```
    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.

Uses
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, Thm_cont.FILTER_DISCH_THEN, Drule.NEG_DISCH, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC.

DISCH_THEN (Thm_cont)

```
DISCH_THEN : (thm_tactic -> tactic)
```

Synopsis

Undischarges an antecedent of an implication and passes it to a theorem-tactic.

Description

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, Thm_cont.FILTER_DISCH_THEN, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC.

DISJ1 (Thm)

```
DISJ1 : thm -> term -> thm
```

Synopsis

Introduces a right disjunct into the conclusion of a theorem.

Description

```
     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)

```
DISJ1_TAC : tactic
```

Synopsis
Selects the left disjunct of a disjunctive goal.

Description

```
   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)

```
DISJ2 : term -> thm -> thm
```

Synopsis
Introduces a left disjunct into the conclusion of a theorem.

Description

```
      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)

```
DISJ2_TAC : tactic
```

Synopsis

Selects the right disjunct of a disjunctive goal.

Description

```
    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)

```
DISJ_CASES : (thm -> thm -> thm -> thm)
```

Synopsis

Eliminates disjunction by cases.

Description

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:

```
th = |- (m = 0) \/ (?n. m = SUC n)
```

Using two additional theorems, each having one disjunct as a hypothesis:

```
th1 = (m = 0 |- (PRE m = m) = (m = 0))
th2 = (?n. m = SUC n" |- (PRE m = m) = (m = 0))
```

a new theorem can be derived:

```
- DISJ_CASES 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

`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)

```
DISJ_CASES_TAC : thm_tactic
```

Synopsis

Produces a case split based on a disjunctive theorem.

Description

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
```

Uses

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)

```
DISJ_CASES_THEN : thm_tactical
```

Synopsis

Applies a theorem-tactic to each disjunct of a disjunctive theorem.

Description

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)
```

Uses

Building cases tactics. For example, `DISJ_CASES_TAC` could be defined by:

```
let 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.STRIP_THM_THEN`, `Thm_cont.CHOOSE_THEN`, `Thm_cont.CONJUNCTS_THEN`, `Thm_cont.CONJUNCTS_THEN2`, `Tactic.DISJ_CASES_TAC`, `Thm_cont.DISJ_CASES_THEN2`, `Thm_cont.DISJ_CASES_THENL`.

DISJ_CASES_THEN2 (Thm_cont)

```
DISJ_CASES_THEN2 : (thm_tactic -> thm_tactical)
```

Synopsis

Applies separate theorem-tactics to the two disjuncts of a theorem.

Description

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)
```

Uses

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)

```
DISJ_CASES_THENL : (thm_tactic list -> thm_tactic)
```

Synopsis

Applies theorem-tactics in a list to the corresponding disjuncts in a theorem.

Description

If the theorem-tactics f1...fn applied to the ASSUMEd 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.

Uses

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)

```
DISJ_CASES_UNION : thm -> thm -> thm -> thm
```

Synopsis

Makes an inference for each arm of a disjunct.

Description

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)

```
DISJ_IMP : (thm -> thm)
```

Synopsis

Converts a disjunctive theorem to an equivalent implicative theorem.

Description

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`.

disjunction (boolSyntax)

```
disjunction : term
```

Synopsis
Constant denoting logical disjunction.

Description
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`.

dom_rng (Type)

```
dom_rng : hol_type -> hol_type * hol_type
```

Synopsis
Breaks a function type into domain and range types.

Description
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‘, ‘:’a‘) : hol_type * hol_type

- try dom_rng bool;

Exception raised at Type.dom_rng:
not a function type
```

See also
`Type.-->`, `Type.dest_type`, `Type.dest_thy_type`.

e (goalstackLib)

```
e : tactic -> goalstack
```

Synopsis
Applies a tactic to the current goal, stacking the resulting subgoals.

Description
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`.

Uses
Doing a step in an interactive goal-directed proof.

See also
`goalstackLib.b`, `goalstackLib.backup`, `backup_limit`, `goalstackLib.expand`, `goalstackLib.expandf`, `goalstackLib.g`, `get_state`, `goalstackLib.p`, `print_state`, `goalstackLib.r`, `rotate`, `save_top_thm`, `goalstackLib.set_goal`, `set_state`, `goalstackLib.top_goal`, `goalstackLib.top_thm`, `VALID`.

el (Lib)

```
el : int -> 'a list -> 'a
```

Synopsis
Extracts a specified element from a list.

Description
`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`.

emit_ERR (Feedback)

```
emit_ERR : bool ref
```

Synopsis
Flag controlling output of `HOL_ERR` exceptions.

Description
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 raised at Module.function:
message
! Uncaught exception:
! HOL_ERR

- emit_ERR := false;
> val it = () : unit

- Raise (mk_HOL_ERR "Module" "function" "message");
! Uncaught exception:
! HOL_ERR
```

See also

Feedback, Feedback.Raise, Feedback.emit_MESG, Feedback.emit_WARNING.

emit_MESG (Feedback)

```
emit_MESG : bool ref
```

Synopsis

Flag controlling output of HOL_MESG function.

Description

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.>>

- 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)

```
emit_WARNING : bool ref
```

Synopsis

Flag controlling output of HOL_WARNING function.

Description

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)

```
empty_rewrites: rewrites
```

Synopsis
The empty database of rewrite rules.

Uses
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)

```
empty_tmset : term set
```

Synopsis
Empty set of terms.

Description
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)

```
empty_varset : term set
```

Synopsis
Empty set of term variables.

Description
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.`

end_itlist (Lib)

```
end_itlist : ('a -> 'a -> 'a) -> 'a list -> 'a)
```

Synopsis
List iteration function. Applies a binary function between adjacent elements of a list.

Description
`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)

```
end_time : Timer.cpu_timer -> unit
```

Synopsis
Check a running timer, and print out how long it has been running.

Description
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 = <cpu_timer> : cpu_timer

- use "foo.sml";
> ... output omitted ...

- end_time clock;
runtime: 525.996s,    gctime: 0.000s,     systime: 525.996s.
> 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`.

enumerate (Lib)

```
enumerate : int -> 'a list -> (int * 'a) list
```

Synopsis
Number each element of a list, in ascending order.

Description
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")]
```

EQ_IMP_RULE (Thm)

```
EQ_IMP_RULE : thm -> thm * thm
```

Synopsis
Derives forward and backward implication from equality of boolean terms.

Description
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_MP (Thm)

```
EQ_MP : thm -> thm -> thm
```

Synopsis
Equality version of the Modus Ponens rule.

Description

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_TAC (Tactic)

```
EQ_TAC : tactic
```

Synopsis

Reduces goal of equality of boolean terms to forward and backward implication.

Description

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.

See also

`Thm.EQ_IMP_RULE`, `Drule.IMP_ANTISYM_RULE`.

EQF_ELIM (Drule)

```
EQF_ELIM : (thm -> thm)
```

Synopsis
Replaces equality with `F` by negation.

Description

```
 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)

```
EQF_INTRO : (thm -> thm)
```

Synopsis
Converts negation to equality with `F`.

Description

```
  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)

```
EQT_ELIM : (thm -> thm)
```

Synopsis
Eliminates equality with `T`.

Description

```
    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)

```
EQT_INTRO : (thm -> thm)
```

Synopsis
Introduces equality with `T`.

Description

```
      A |- tm
  -------------  EQF_INTRO
    A |- tm = T
```

Failure
Never fails.

See also
`Drule.EQT_ELIM`, `Drule.EQF_ELIM`, `Drule.EQF_INTRO`.

equal (Lib)

```
equal : ''a -> ''a -> bool
```

Synopsis
Curried form of ML equality

Description
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)

```
equality : term
```

Synopsis
Constant denoting logical equality.

Description
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)

```
ERR_outstream : TextIO.outstream ref
```

Synopsis

Reference to output stream used when printing HOL_ERR

Description

The value of reference cell ERR_outstream controls where Raise prints its argument.

The default value of ERR_outstream is TextIO.stdErr.

Example

```
- val ostrm = TextIO.openOut "foo";
> val ostrm = <outstream> : outstream

- ERR_outstream := ostrm;
> val it = () : unit

- Raise (mk_HOL_ERR "Foo" "bar" "incomprehensible input");
! Uncaught exception:
! HOL_ERR

- TextIO.closeOut ostrm;
> val it = () : unit

- val istrm = TextIO.openIn "foo";
> val istrm = <instream> : instream

- print (TextIO.inputAll istrm);

Exception raised at Foo.bar:
incomprehensible input
```

See also

Feedback, Feedback.HOL_ERR, Feedback.Raise, Feedback.MESG_outstream, Feedback.WARNING_outstream.

ERR_to_string (Feedback)

```
ERR_to_string : (error_record -> string) ref
```

Synopsis

Alterable function for formatting HOL_ERR

Description

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"];

- ERR_to_string := alt_ERR_report;

- Raise (HOL_ERR {origin_structure = "Foo",
                  origin_function = "bar",
                  message = "incomprehensible input"});

This just in from bar at Foo : incomprehensible input
! Uncaught exception:
! HOL_ERR
```

See also

Feedback, Feedback.error_record, Feedback.HOL_ERR, Feedback.Raise, Feedback.MESG_to_string, Feedback.WARNING_to_string.

error_record (Feedback)

```
type error_record = {origin_structure : string,
                     origin_function  : string,
                     message          : string}
```

Synopsis

Type abbreviation for HOL exceptions in Feedback module.

Description

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 (Thm)

```
Eta : thm -> thm
```

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

Description

```
    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 = (\x. 1 + x) : thm

- Eta INC_DEF;
> val it = |- INC = $+ 1 : thm
```

See also
Thm.Beta, Term.eta_conv, Thm.ETA_CONV.

eta_conv (Term)

```
eta_conv : term -> term
```

Synopsis
Performs one step of eta-reduction.

Description

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

`Thm.ETA_CONV`.

ETA_CONV (Thm)

```
ETA_CONV : conv
```

Synopsis

Performs a toplevel eta-conversion.

Description

`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

`Thm.BETA_CONV`, `Thm.ALPHA`, `Term.eta_conv`.

etyvar (Type)

```
etyvar : hol_type
```

Synopsis
Common type variable.

Description
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)

```
EVAL : conv
```

Synopsis
Evaluate a term by deduction.

Description
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 (\x. 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, computeLib.the_compset, computeLib.add_funs, computeLib.add_persistent_funs, computeLib.add_convs, bossLib.Define.

EVAL_RULE (bossLib)

```
EVAL_RULE : thm -> thm
```

Synopsis
Evaluate conclusion of a theorem.

Description
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])

- EVAL_RULE th;
> val it =  [.] |- x = [3628800; 362880; 40320; 5040; 720; 120; 24; 6; 2; 1]

- hyp it;
> val it = [`x = MAP FACT (REVERSE [1; 2; 3; 4; 5; 6; 7; 8; 9; 10])`]
```

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)

```
EVAL_TAC : tactic
```

Synopsis
Evaluate a goal deductively.

Description
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.

Uses
Symbolic evaluation.

See also
`bossLib.EVAL`.

EVERY (Tactical)

```
EVERY : (tactic list -> tactic)
```

Synopsis
Sequentially applies all the tactics in a given list of tactics.

Description
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)

```
EVERY_ASSUM : (thm_tactic -> tactic)
```

Synopsis
Sequentially applies all tactics given by mapping a function over the assumptions of a goal.

Description
When applied to a theorem-tactic `f` and a goal `({A1,...,An} ?- C)`, the `EVERY_ASSUM` tactical maps `f` over a list of `ASSUME`d 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)

```
EVERY_CONJ_CONV : conv -> conv
```

Synopsis
Applies a conversion to every top-level conjunct in a term.

Description
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 = |- (\x. x /\ y) p = p /\ y : thm
- EVERY_CONJ_CONV BETA_CONV (Term`(\y. y /\ p) q /\ (\z. z) r`);
> val it = |- (\y. y /\ p) q /\ (\z. z) r = (q /\ p) /\ r : thm
```

Uses
Useful for applying a conversion to all of the “significant” sub-terms within a term with-

out 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_CONV (Conv)

```
EVERY_CONV : (conv list -> conv)
```

Synopsis
Applies in sequence all the conversions in a given list of conversions.

Description
`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"` returns `|- t=t`.

Failure
`EVERY_CONV [c1;...;cn] "t"` fails if any one of the conversions `c1`, ..., `cn` fails when applied in sequence as specified above.

See also
Conv.THENC.

EVERY_DISJ_CONV (Conv)

```
EVERY_DISJ_CONV : conv -> conv
```

Synopsis
Applies a conversion to every top-level disjunct in a term.

Description
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 =
    |- (\x. x /\ p) q \/ (\x. x) r \/ (\y. s /\ y) u = q /\ p \/ r \/ s /\ u
    : thm
- EVERY_DISJ_CONV REDUCE_CONV ``3 < x \/ 2 < 3 \/ 2 EXP 1000 < 10``;
> val it = |- 3 < x \/ 2 < 3 \/ 2 EXP 1000 < 10 = T : thm
```

Uses

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`.

EVERY_TCL (Thm_cont)

```
EVERY_TCL : (thm_tactical list -> thm_tactical)
```

Synopsis

Composes a list of theorem-tacticals.

Description

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.

EXISTENCE (Conv)

```
EXISTENCE : (thm -> thm)
```

Synopsis

Deduces existence from unique existence.

Description

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)

```
existential : term
```

Synopsis
Constant denoting existential quantification.

Description
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)

```
exists : ('a -> bool) -> 'a list -> bool
```

Synopsis
Check if a predicate holds somewhere in a list

Description
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];
! Uncaught exception:
! Fail  ""
```

See also

`Lib.all`, `Lib.first`, `Lib.tryfind`.

EXISTS (Thm)

```
EXISTS : term * term -> thm -> thm
```

Synopsis

Introduces existential quantification given a particular witness.

Description

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`, `Tactic.EXISTS_TAC`.

existential (boolSyntax)

```
exists1 : term
```

Synopsis
Constant denoting the unique existence quantifier.

Description
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)

```
EXISTS_AND_CONV : conv
```

Synopsis
Moves an existential quantification inwards through a conjunction.

Description

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.LEFT_AND_EXISTS_CONV`, `Conv.RIGHT_AND_EXISTS_CONV`.

EXISTS_EQ (Drule)

```
EXISTS_EQ : (term -> thm -> thm)
```

Synopsis

Existentially quantifies both sides of an equational theorem.

Description

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_IMP (Drule)

```
EXISTS_IMP : (term -> thm -> thm)
```

Synopsis
Existentially quantifies both the antecedent and consequent of an implication.

Description
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)

```
EXISTS_IMP_CONV : conv
```

Synopsis
Moves an existential quantification inwards through an implication.

Description

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_NOT_CONV (Conv)

```
EXISTS_NOT_CONV : conv
```

Synopsis

Moves an existential quantification inwards through a negation.

Description

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)

```
EXISTS_OR_CONV : conv
```

Synopsis
Moves an existential quantification inwards through a disjunction.

Description
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)

```
EXISTS_TAC : (term -> tactic)
```

Synopsis
Reduces existentially quantified goal to one involving a specific witness.

Description
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)

```
exists_tyvar : (hol_type -> bool) -> hol_type -> bool
```

Synopsis
Checks if a type variable satisfying a given condition exists in a type.

Description
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)

```
EXISTS_UNIQUE_CONV : conv
```

Synopsis
Expands with the definition of unique existence.

Description
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)

```
exn_to_string : exn -> string
```

Synopsis
Map an exception into a string

Description
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;
> Interrupted.

- exn_to_string Div;
> val it = "Div" : string

- print
   (exn_to_string (mk_HOL_ERR "Foo" "bar" "incomprehensible input"));

Exception raised at Foo.bar:
incomprehensible input
> val it = () : unit
```

See also

Feedback, Feedback.HOL_ERR, Feedback.ERR_to_string.

expand (goalstackLib)

```
expand : tactic -> goalstack
```

Synopsis

Applies a tactic to the current goal, stacking the resulting subgoals.

Description

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

```
- expand CONJ_TAC;
- expand CONJ_TAC;
OK..
NO_PROOFS! Uncaught exception:
! NO_PROOFS

- g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])

     : proofs

- expand CONJ_TAC;
OK..
2 subgoals:
> val it =
    TL [1; 2; 3] = [2; 3]


    HD [1; 2; 3] = 1

     : goalstack

- expand (REWRITE_TAC[listTheory.HD]);
OK..

Goal proved.
|- HD [1; 2; 3] = 1

Remaining subgoals:
> val it =
    TL [1; 2; 3] = [2; 3]

     : goalstack

- 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]) : goalstack
```

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: 2 proofs.
    2. Completed: |- (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])
    1. Incomplete:
         Initial goal:
         1 = 2

     : proofs
- expand (REWRITE_TAC[ASSUME (Term `1=2`)]);
OK..

Exception raised at Tactical.VALID:
Invalid tactic
! Uncaught exception:
! HOL_ERR
```

Uses

Doing a step in an interactive goal-directed proof.

See also

goalstackLib.b, goalstackLib.backup, goalstackLib.e, goalstackLib.expandf, goalstackLib.g, goalstackLib.set_goal, goalstackLib.p, goalstackLib.r, goalstackLib.rotate, goalstackLib.top_goal, goalstackLib.top_thm, Tactical.VALID.

expandf (goalstackLib)

```
expandf : (tactic -> unit)
```

Synopsis

Applies a tactic to the current goal, stacking the resulting subgoals.

Description

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.

Uses

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

`goalstackLib.b`, `goalstackLib.backup`, `backup_limit`, `goalstackLib.e`, `goalstackLib.expand`, `goalstackLib.g`, `get_state`, `goalstackLib.p`, `print_state`, `goalstackLib.r`, `rotate`, `save_top_thm`, `goalstackLib.set_goal`, `set_state`, `goalstackLib.top_goal`, `goalstackLib.top_thm`, `VALID`.

export_rewrites (BasicProvers)

```
export_rewrites : string list -> unit
```

Synopsis
Exports theorems so that they merge with the "stateful" rewriter's `simpset`.

Description
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()` when the theory generated by the script file is loaded.

The theory is also augmented with an element in its signature of the form `<thyname>_rwts` of type `simpLib.ssdata`. This value is the collection of all the theorems specified in the previous call to `export_rewrites`.

Failure
Never fails directly. However, if a string in the parameter does not correspond to a theorem exported by the theory (using `save_thm`, say), then the theory file resulting from the execution of the script that includes the call to `export_rewrites` will not compile.

See also
`bossLib.augment_srw_ss`, `bossLib.srw_ss`, `bossLib.SRW_TAC`.

export_theory (Theory)

```
export_theory : unit -> unit
```

Synopsis
Write a theory segment to disk.

Description
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 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 (Term `x:bool`));
> val it = |- x = x : thm

- export_theory();
Exporting theory "scratch" ... done.
> val it = () : unit
```

Comments

Note that `export_theory` exports the state of the theory, and not that of the ML environment. If one wants to restore the state of the ML environment in existence at the time `export_theory()` is invoked, special steps have to be taken; see `adjoin_to_theory`.

See also

`Theory.new_theory`, `Theory.adjoin_to_theory`, `Theory.set_MLname`, `Holmake`.

EXT (Drule)

```
EXT : thm -> thm
```

Synopsis

Derives equality of functions from extentional equivalence.

Description

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`, `Thm.ETA_CONV`, `Conv.FUN_EQ_CONV`.

F (boolSyntax)

```
F : term
```

Synopsis

Constant denoting falsity.

Description

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)

```
fail : unit -> 'a
```

Synopsis
Raise a HOL_ERR.

Description
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 raised at ??.??:
fail
! Uncaught exception:
! HOL_ERR
```

See also
Feedback, Feedback.failwith, Feedback.Raise, Feedback.HOL_ERR.

FAIL_TAC (Tactical)

```
FAIL_TAC : (string -> tactic)
```

Synopsis
Tactic which always fails, with the supplied string.

Description
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:

```
?- x => T | 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)

```
failwith : string -> 'a
```

Synopsis

Raise a `HOL_ERR`.

Description

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 raised at ??.failwith:
foo
! Uncaught exception:
! HOL_ERR
```

See also

`Feedback`, `Feedback.fail`, `Feedback.Raise`, `Feedback.HOL_ERR`.

Feedback

```
structure Feedback
```

Synopsis
Module for messages, warnings, errors, and tracing of HOL functions.

Description
The `Feedback` structure provides facilities for raising and viewing HOL errors, and also for monitoring tools as they run.

fetch (DB)

```
fetch : string -> string -> thm
```

Synopsis
Fetch a theorem by theory and name.

Description
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)

```
filter : ('a -> bool) -> 'a list -> 'a list)
```

Synopsis
Filters a list to the sublist of elements satisfying a predicate.

Description
`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 `gather`.

See also
`Lib.gather, Lib.mapfilter, Lib.partition.`

FILTER_ASM_REWRITE_RULE (Rewrite)

```
FILTER_ASM_REWRITE_RULE : ((term -> bool) -> thm list -> thm -> thm)
```

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

Description
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.

Uses
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)

```
FILTER_ASM_REWRITE_TAC : ((term -> bool) -> thm list -> tactic)
```

Synopsis

Rewrites a goal including built-in rewrites and some of the goal's assumptions.

Description

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.

Uses

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_DISCH_TAC (Tactic)

```
FILTER_DISCH_TAC : (term -> tactic)
```

Synopsis

Conditionally moves the antecedent of an implicative goal into the assumptions.

Description

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, Thm_cont.FILTER_DISCH_THEN, Drule.NEG_DISCH, Tactic.STRIP_TAC, Drule.UNDISCH, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC.

FILTER_DISCH_THEN (Thm_cont)

```
FILTER_DISCH_THEN : (thm_tactic -> term -> tactic)
```

Synopsis

Conditionally gives to a theorem-tactic the antecedent of an implicative goal.

Description

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.

Uses

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)

```
FILTER_GEN_TAC : (term -> tactic)
```

Synopsis

Strips off a universal quantifier, but fails for a given quantified variable.

Description

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, Drule.GENL, Drule.GEN_ALL, Thm.SPEC, Drule.SPECL, Drule.SPEC_ALL, Tactic.SPEC_TAC, Tactic.STRIP_TAC.

FILTER_ONCE_ASM_REWRITE_RULE (Rewrite)

```
FILTER_ONCE_ASM_REWRITE_RULE : ((term -> bool) -> thm list -> thm -> thm)
```

Synopsis

Rewrites a theorem once including built-in rewrites and some of its assumptions.

Description

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 `basic_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.

Uses

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)

```
FILTER_ONCE_ASM_REWRITE_TAC : ((term -> bool) -> thm list -> tactic)
```

Synopsis

Rewrites a goal once including built-in rewrites and some of its assumptions.

Description

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 `basic_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.

Uses

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)

```
FILTER_PGEN_TAC : (term -> tactic)
```

Synopsis

Strips off a paired universal quantifier, but fails for a given quantified pair.

Description

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, PGEN_ALL, PairRules.PSPEC, PairRules.PSPECL, PairRules.PSPEC_ALL, PairRules.PSPEC_TAC, PairRules.PSTRIP_TAC.

FILTER_PSTRIP_TAC (PairRules)

```
FILTER_PSTRIP_TAC : (term -> tactic)
```

Synopsis

Conditionally strips apart a goal by eliminating the outermost connective.

Description

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).

Uses

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)

```
FILTER_PSTRIP_THEN : (thm_tactic -> term -> tactic)
```

Synopsis

Conditionally strips a goal, handing an antecedent to the theorem-tactic.

Description

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.

Uses

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, Thm_cont.FILTER_STRIP_THEN, PairRules.PSTRIP_TAC, PairRules.FILTER_PSTRIP_TAC.

FILTER_PURE_ASM_REWRITE_RULE (Rewrite)

```
FILTER_PURE_ASM_REWRITE_RULE : ((term -> bool) -> thm list -> thm ->thm)
```

Synopsis
Rewrites a theorem with some of the theorem's assumptions.

Description
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.

Uses
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)

```
FILTER_PURE_ASM_REWRITE_TAC : ((term -> bool) -> thm list -> tactic)
```

Synopsis
Rewrites a goal with some of the goal's assumptions.

Description
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.

Uses
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)

```
FILTER_PURE_ONCE_ASM_REWRITE_RULE : ((term -> bool) -> thm list -> thm -> thm)
```

Synopsis
Rewrites a theorem once using some of its assumptions.

Description

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.

Uses

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)

```
FILTER_PURE_ONCE_ASM_REWRITE_TAC : ((term -> bool) -> thm list -> tactic)
```

Synopsis

Rewrites a goal once using some of its assumptions.

Description

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.

Uses

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)

```
FILTER_STRIP_TAC : (term -> tactic)
```

Synopsis

Conditionally strips apart a goal by eliminating the outermost connective.

Description

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 ?- v1 \/ ... \/ vn ==> v
============================       ================================
    A u {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"
```

Uses

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, Thm_cont.FILTER_DISCH_THEN, Tactic.FILTER_GEN_TAC, Tactic.STRIP_ASSUME_TAC, Tactic.STRIP_TAC.

FILTER_STRIP_THEN (Thm_cont)

```
FILTER_STRIP_THEN : (thm_tactic -> term -> tactic)
```

Synopsis
Conditionally strips a goal, handing an antecedent to the theorem-tactic.

Description
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
```

Uses
`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`, `Thm_cont.FILTER_DISCH_THEN`, `Tactic.FILTER_GEN_TAC`, `Tactic.FILTER_STRIP_TAC`, `Tactic.STRIP_ASSUME_TAC`, `Tactic.STRIP_GOAL_THEN`.

find (DB)

```
find : string -> data list
```

Synopsis
Search for theory element by name fragment.

Description
An invocation `DB.find s` returns a list of theory elements which have been stored with a name in which `s` occurs as a proper substring, ignoring case distinctions. All currently loaded theory segments are searched.

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)),
     (("bool", "BOOL_EQ_DISTINCT"), (|- ~(T = F) /\ ~(F = T), Thm)),
     (("list", "list_distinct"), (|- !a1 a0. ~([] = a0::a1), Thm)),
     (("sum", "sum_distinct"), (|- !x y. ~(INL x = INR y), Thm)),
     (("sum", "sum_distinct1"), (|- !x y. ~(INR y = INL x), Thm))]
  : ((string * string) * (thm * class)) list
```

Uses
Finding theorems in interactive proof sessions.

See also
`DB.match`, `DB.apropos`, `DB.thy`, `DB.theorems`.

find (hol88Lib)

```
find : ('a -> bool) -> 'a list -> 'a
```

Synopsis
Returns the first list element that satisfies a predicate.

Description
Identical to `Lib.first`.

See also
`Lib.first`, `Lib.tryfind`, `Lib.mem`, `Lib.exists`, `hol88Lib.forall`, `Lib.assoc`, `Lib.rev_assoc`.

find (Lib)

```
find : ('a -> bool) -> 'a list -> 'a
```

Synopsis
Returns the first list element that satisfies a predicate.

Description
An invocation `find P [x1,...,xn]` returns the first `xi` in `[x1,...,xn]` such that `(P xi)` is `true`.

Failure
Fails if no element satisfies the predicate. This will always be the case if the list is empty. Also fails if `P xi` fails for any element `xi` of the list.

Comments
`Lib.first` is equivalent.

See also
`Lib.first, Lib.tryfind, Lib.mem, Lib.exists, Lib.assoc, Lib.rev_assoc.`

first (Lib)

```
first : ('a -> bool) -> 'a list -> 'a
```

Synopsis
Return first element in list that predicate holds of.

Description
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];
! Uncaught exception:
! HOL_ERR

- first (fn _ => raise Fail "") [1];
! Uncaught exception:
! Fail  ""
```

See also

Lib.exists, Lib.tryfind, Lib.all, Lib.exists.

FIRST (Tactical)

```
FIRST : (tactic list -> tactic)
```

Synopsis

Applies the first tactic in a tactic list which succeeds.

Description

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)

```
FIRST_ASSUM : (thm_tactic -> tactic)
```

Synopsis
Maps a theorem-tactic over the assumptions, applying first successful tactic.

Description
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 (\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.

See also
`Tactical.ASSUM_LIST`, `Tactical.EVERY`, `Tactical.EVERY_ASSUM`, `Tactical.FIRST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`.

FIRST_CONV (Conv)

```
FIRST_CONV : (conv list -> conv)
```

Synopsis
Apply the first of the conversions in a given list that succeeds.

Description
`FIRST_CONV [c1;...;cn] "t"` returns the result of applying to the term `"t"` the first conversion `ci` that succeeds 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`.

FIRST_TCL (Thm_cont)

```
FIRST_TCL : (thm_tactical list -> thm_tactical)
```

Synopsis
Applies the first theorem-tactical in a list which succeeds.

Description
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
```

Synopsis
Maps a theorem-tactic over the assumptions, applying first successful tactic and removing the assumption that gave rise to the successful tactic.

Description

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 `ASSUME`d 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.

See also

`Tactical.ASSUM_LIST`, `Tactical.EVERY`, `Tactical.PAT_ASSUM`, `Tactical.EVERY_ASSUM`, `Tactical.FIRST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`, `Thm_cont.UNDISCH_THEN`.

flatten (Lib)

```
flatten : 'a list list -> 'a list
```

Synopsis
Removes one level of bracketing from a list.

Description
An invocation `flatten [[x11,...,x1k1],...,[xn1,...,xnkn]]` yields the list `[x1,...,x1k1,...,xn1,...,xn`

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
```

for (Lib)

```
for : int -> int -> (int -> 'a) -> 'a list
```

Synopsis
Functional 'for' loops.

Description
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)

```
for_se : int -> int -> (int -> unit) -> unit
```

Synopsis
Side-effecting 'for' loops.

Description
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 (print o Char.toString o curry Array.sub A)
   ; print "\n"
  end;

abcdefghijklmnopqrstuvwxyz
> val it = () : unit
```

See also
`Lib.for`.

FORALL_AND_CONV (Conv)

```
FORALL_AND_CONV : conv
```

Synopsis
Moves a universal quantification inwards through a conjunction.

Description

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_EQ (Drule)

```
FORALL_EQ : (term -> thm -> thm)
```

Synopsis

Universally quantifies both sides of an equational theorem.

Description

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_IMP_CONV (Conv)

```
FORALL_IMP_CONV : conv
```

Synopsis
Moves a universal quantification inwards through an implication.

Description
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)

```
FORALL_NOT_CONV : conv
```

Synopsis
Moves a universal quantification inwards through a negation.

Description
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)

```
FORALL_OR_CONV : conv
```

Synopsis
Moves a universal quantification inwards through a disjunction.

Description
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.

FORK_CONV (Conv)

```
FORK_CONV : (conv * conv) -> conv
```

Synopsis
Applies a pair of conversions to the arguments of a binary operator.

Description
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,REDUCE_CONV) (Term`(\x. x + 1)y * (10 DIV 3)`);
> val it = |- (\x. x + 1) y * (10 DIV 3) = (y + 1) * 3 : thm
```

See also
`Conv.BINOP_CONV`, `Conv.LAND_CONV`, `Conv.RAND_CONV`, `Conv.RATOR_CONV`.

format_ERR (Feedback)

```
format_ERR : error_record -> string
```

Synopsis
Maps argument record of `HOL_ERR` to a string

Description
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 raised at Foo.bar:
incomprehensible input
> val it = () : unit
```

See also

Feedback, Feedback.ERR_to_string, Feedback.format_MESG, Feedback.format_WARNING.

format_MESG (Feedback)

```
format_MESG : string -> string
```

Synopsis

Maps argument of HOL_MESG to a string

Description

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

```
- print (format_MESG "Hello world.");
<<HOL message: Hello world.>>
```

See also

Feedback, Feedback.MESG_to_string, Feedback.format_ERR, Feedback.format_WARNING.

format_WARNING (Feedback)

```
format_WARNING : string -> string -> string -> string
```

Synopsis
Maps arguments of HOL_WARNING to a string

Description
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

```
- print (format_WARNING "Module" "function" "Gadzooks!");
<<HOL warning: Module.function: Gadzooks!>>
```

See also
Feedback, Feedback.WARNING_to_string, Feedback.format_ERR, Feedback.format_MESG.

free_in (Term)

```
free_in : term -> term -> bool
```

Synopsis
Tests if one term is free in another.

Description
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 all the free variables of M are not bound at some occurrence of M 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 (Term `SUC x`)
          (Term `!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 (Term `x:bool`)
          (Term `x /\ ?x. x=T`);
> val it = true : bool
```

See also

`Term.free_vars`, `Term.FVL`.

free_vars (Term)

```
free_vars : term -> term list
```

Synopsis

Returns the set of free variables in a term.

Description

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)

```
free_vars_lr : term -> term list
```

Synopsis
Returns the set of free variables in a term, in order.

Description
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.

Uses
`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)

```
free_varsl : term list -> term list
```

Synopsis
Returns the set of free variables in a list of terms.

Description
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
```

Synopsis
Returns a list of the variables which are free in a term.

Description
Found in the hol88 library. When applied to a term, `frees` returns a list of the free variables in that term. There are no repetitions in the list produced even if there are multiple free instances of some variables.

Failure
Never fails.

Example

Clearly in the following term, `x` and `y` are free, whereas `z` is bound:

```
- frees (--‘(x=1) /\ (y=2) /\ (!z. z >= 0)‘--);
> val it = [(--‘x‘--),(--‘y‘--)] : term list
```

Comments

The function `frees` is not in the standard hol98 kernel; the function `free_vars` is used instead. WARNING: the order of the list returned by `frees` and `free_vars` is different.

```
- val tm = (--‘x (y:num):bool‘--);
> val tm = (--‘x y‘--) : term
- free_vars tm
> val it = [(--‘y‘--),(--‘x‘--)] : term list
- frees tm;
> val it = [(--‘x‘--),(--‘y‘--)] : term list
```

It ought to be the case that the result of a call to `frees` (or `free_vars`) is treated as a set, that is, the order of the free variables should be immaterial. This is sometimes not possible; for example the result of `gen_all` (and hence the results of `GEN_ALL` and `new_axiom`) necessarily depends on the order of the variables returned from `frees`. The problem comes when users write code that depends on the order of quantification. For example, contrary to some expectations, it is not the case that (`tm` being a closed term already)

```
GEN_ALL (SPEC_ALL tm) = tm
```

where “=” is interpreted as identity or alpha-convertibility.

See also

`hol88Lib.freesl`, `Term.free_in`, `thm_frees`.

freesl (hol88Lib)

```
Compat.freesl : term list -> term list
```

Synopsis

Returns a list of the free variables in a list of terms.

Description

Found in the hol88 library. When applied to a list of terms, `freesl` returns a list of the variables which are free in any of those terms. There are no repetitions in the list produced even if several terms contain the same free variable.

Failure

Never fails, unless the hol88 library has not been loaded.

Example

In the following example there are two free instances each of `x` and `y`, whereas the only instances of `z` are bound:

```
- freesl [(--‘x+y=2‘--), (--‘!z. z >= (x-y)‘--)];
val it = [(--‘x‘--),(--‘y‘--)] : term list
```

Comments

`freesl` is not in hol90; use `free_varsl` instead. WARNING: One can not depend on the order of the list returned by `freesl` to be identical to that returned by `free_varsl`. They are coded in terms of `frees` and `free_vars`, and thus the discussion in the documentation for `frees` applies by extension.

See also

`hol88Lib.frees, Term.free_in, thm_frees.`

FREEZE_THEN (Tactic)

```
FREEZE_THEN : thm_tactical
```

Synopsis

‘Freezes’ a theorem to prevent instantiation of its free variables.

Description

`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:

```
 A ?- t
=========  f (w |- w)
 A ?- t1
```

then applying `FREEZE_THEN f (A1 |- w)` to the goal `(A ?- t)` produces the subgoal:

```
 A ?- t
=========  FREEZE_THEN f (A1 |- w)
 A ?- 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
```

Uses

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_CONJ_CONV (Drule)

```
FRONT_CONJ_CONV: term list -> term -> thm
```

Synopsis
Moves a specified conjunct to the beginning of a conjunction.

Description
Given a list of boolean terms `[t1,...,t,...,tn]` and a term `t` which occurs in the list, `FRONT_CONJ_CONV` returns:

```
|- (t1 /\ ... /\ t /\ ... /\ tn) = (t /\ t1 /\ ... /\ tn)
```

That is, `FRONT_CONJ_CONV` proves that `t` can be moved to the 'front' of a conjunction of several terms.

Failure
`FRONT_CONJ_CONV [t1,...,tn] t` fails if `t` does not occur in the list `[t1,...,tn]` or if any of `t1`, ..., `tn` do not have type `bool`.

Comments
This is not a true conversion, so perhaps it ought to be called something else.

front_last (Lib)

```
Lib.front_last : 'a list -> 'a list * 'a
```

Synopsis
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)

```
fst : ('a * 'b) -> 'a
```

Synopsis

Extracts the first component of a pair.

Description

`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", []);
! Toplevel input:
! fst (1, "foo", []);
!     ^^^^^^^^^^^^^^
! Type clash: expression of type
!   'g * 'h * 'i
! cannot have type
!   'j * 'k
! because the tuple has the wrong number of components

- fst (1, ("foo", []));
> val it = 1 : int
```

See also

`Lib.snd`.

ftyvar (Type)

```
ftyvar : hol_type
```

Synopsis

Common type variable.

Description

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_SIMP_TAC (bossLib)

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

Synopsis

Simplifies the goal (assumptions as well as conclusion) with the given simpset.

Description

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.

See also
bossLib.ASM_SIMP_TAC, bossLib.SIMP_CONV, bossLib.SIMP_RULE, bossLib.SIMP_TAC.

FULL_SIMP_TAC (simpLib)

```
FULL_SIMP_TAC : simpset -> thm list -> tactic
```

Synopsis
Simplify a term with the given simpset and theorems.

Description
bossLib.FULL_SIMP_TAC is identical to simpLib.FULL_SIMP_TAC.

See also
bossLib.FULL_SIMP_TAC.

FUN_EQ_CONV (Conv)

```
FUN_EQ_CONV : conv
```

Synopsis
Equates normal and extensional equality for two functions.

Description
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.

Uses
Used for proving equality of functions.

See also
`Drule.EXT`, `Conv.X_FUN_EQ_CONV`.

funpow (Lib)

```
funpow : int -> ('a -> 'a) -> 'a -> 'a
```

Synopsis
Iterates a function a fixed number of times.

Description
`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`.

FVL (Term)

```
FVL : term list -> term set -> term set
```

Synopsis
Efficient computation of the set of free variables in a list of terms.

Description
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 = <set> : 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 (goalstackLib)

```
g : term frag list -> proofs
```

Synopsis
Initializes the subgoal package with a new goal which has no assumptions.

Description

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: 1 proof.
    1. Incomplete:
         Initial goal:
         (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])



    : GoalstackPure.proofs
```

See also

`goalstackLib.b`, `goalstackLib.backup`, `backup_limit`, `goalstackLib.e`, `goalstackLib.expand`, `goalstackLib.expandf`, `get_state`, `goalstackLib.p`, `print_state`, `goalstackLib.r`, `rotate`, `save_top_thm`, `goalstackLib.set_goal`, `set_state`, `goalstackLib.top_goal`, `goalstackLib.top_thm`.

gamma (Type)

```
gamma : hol_type
```

Synopsis

Common type variable.

Description
The ML variable `Type.gamma` is bound to the type variable `'c`.

See also
`Type.alpha`, `Type.beta`, `Type.delta`, `Type.bool`.

gather (Lib)

```
gather : ('a -> bool) -> 'a list -> 'a list)
```

Synopsis
Filters a list to the sublist of elements satisfying a predicate.

Description
`gather 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 `filter`.

See also
`Lib.filter`, `Lib.mapfilter`, `Lib.partition`.

GEN (Thm)

```
GEN : term -> thm -> thm
```

Synopsis
Generalizes the conclusion of a theorem.

Description

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 (Term `x=T`);
> val t =  [.] |- (x :bool) = T : thm

- try (GEN (Term `x:bool`)) t;
Exception raised at Thm.GEN:
variable occurs free in hypotheses
! Uncaught exception:
! HOL_ERR
```

See also

Drule.GENL, Drule.GEN_ALL, Tactic.GEN_TAC, Thm.SPEC, Drule.SPECL, Drule.SPEC_ALL, Tactic.SPEC_TAC.

GEN_ALL (Drule)

```
Drule.GEN_ALL : thm -> thm
```

Synopsis

Generalizes the conclusion of a theorem over its own free variables.

Description

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

WARNING: hol90 `GEN_ALL` does not always return the same result as `GEN_ALL` in hol88. Sometimes people write code that depends on the order of the quantification. They shouldn't.

See also

`Thm.GEN`, `Drule.GENL`, `Drule.GEN_ALL`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`, `Tactic.SPEC_TAC`.

GEN_ALPHA_CONV (Drule)

```
GEN_ALPHA_CONV : term -> conv
```

Synopsis

Renames the bound variable of an abstraction, a quantified term, or other binder application.

Description

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_BETA_CONV (PairedLamda)

```
GEN_BETA_CONV : conv
```

Synopsis

Beta-reduces single or paired beta-redexes, creating a paired argument if needed.

Description

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_MESON_TAC (mesonLib)

```
GEN_MESON_TAC : int -> int -> int -> thm list -> tactic
```

Synopsis

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

Description

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.

Uses

The construction of tailored versions of MESON_TAC and ASM_MESON_TAC.

See also

mesonLib.ASM_MESON_TAC, mesonLib.MESON_TAC.

GEN_PALPHA_CONV (PairRules)

```
GEN_PALPHA_CONV : term -> conv
```

Synopsis

Renames the bound pair of a paired abstraction, quantified term, or other binder.

Description

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)

```
GEN_REWRITE_CONV : ((conv -> conv) -> thm list -> thm list -> conv)
```

Synopsis

Rewrites a term, selecting terms according to a user-specified strategy.

Description

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.

Uses
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) []  [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 basic_rewrites
```

which specifies that matches should be searched recursively starting from the whole term of the theorem, and basic_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)

```
GEN_REWRITE_RULE : ((conv -> conv) -> thm list -> thm list -> thm -> thm)
```

Synopsis
Rewrites a theorem, selecting terms according to a user-specified strategy.

Description
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.

Uses

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) []  [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 basic_rewrites
```

which specifies that matches should be searched recursively starting from the whole term of the theorem, and basic_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)

```
GEN_REWRITE_TAC : ((conv -> conv) -> thm list -> thm list -> tactic)
```

Synopsis

Rewrites a goal, selecting terms according to a user-specified strategy.

Description

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.

Uses

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) [] [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:

```
let ADD_REWRITE_TAC = GEN_REWRITE_TAC TOP_DEPTH_CONV
                          (ADD_CLAUSES . basic_rewrites) ;;
```

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)

```
GEN_TAC : tactic
```

Synopsis
Strips the outermost universal quantifier from the conclusion of a goal.

Description
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.

Uses
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`, `Drule.GENL`, `Drule.GEN_ALL`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`, `Tactic.SPEC_TAC`, `Tactic.STRIP_TAC`, `Tactic.X_GEN_TAC`.

gen_tyvar (Type)

```
gen_tyvar : unit -> hol_type
```

Synopsis
Generate a fresh type variable

Description
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%%1‘ : hol_type

- try Type ‘:%%gen_tyvar%%1‘;

Exception raised at Parse.hol_type parser:
Couldn’t make any sense with remaining input of "%%gen_tyvar%%1"

- try mk_vartype "%%gen_tyvar%%1";

Exception raised at Type.mk_vartype:
incorrect syntax
```

Comments
In general, the actual name returned by `gen_tyvar` should not be relied on.

Uses
Useful for coding some proof procedures.

See also
`Term.genvar, Term.variant.`

GENL (Drule)

```
GENL : term list -> thm -> thm
```

Synopsis
Generalizes zero or more variables in the conclusion of a theorem.

Description

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`.

genvar (Term)

```
genvar : type -> term
```

Synopsis

Returns a variable whose name has not been used previously.

Description

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.

Uses

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)

```
genvars : hol_type -> int -> term list
```

Synopsis

Generate a specified number of fresh variables.

Description

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 = [‘%%genvar%%1558‘, ‘%%genvar%%1559‘, ‘%%genvar%%1560‘] : term list
```

See also

Term.genvar, Term.mk_var.

genvarstruct (pairSyntax)

```
genvarstruct : hol_type -> term
```

Synopsis

Returns a pair structure of variables whose names have not been previously used.

Description

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
```

Uses

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.

GPSPEC (PairRules)

```
GPSPEC : (thm -> thm)
```

Synopsis
Specializes the conclusion of a theorem with unique pairs.

Description
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.

Uses
`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`, `PGEN_ALL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`, `PairRules.PSPEC_PAIR`.

GSPEC (Drule)

```
GSPEC : (thm -> thm)
```

Synopsis
Specializes the conclusion of a theorem with unique variables.

Description

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.

Uses

`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`, `Drule.GENL`, `Term.genvar`, `Drule.GEN_ALL`, `Tactic.GEN_TAC`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`, `Tactic.SPEC_TAC`.

GSUBST_TAC (Tactic)

```
GSUBST_TAC : ((term * term) list -> term -> term) -> thm list -> tactic
```

Synopsis

Makes term substitutions in a goal using a supplied substitution function.

Description

`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.

Uses

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)

```
GSYM : thm -> thm
```

Synopsis

Reverses the first equation(s) encountered in a top-down search.

Description

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.

HALF_MK_ABS (Drule)

```
HALF_MK_ABS : (thm -> thm)
```

Synopsis

Converts a function definition to lambda-form.

Description

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

Thm.ETA_CONV, Drule.MK_ABS, Thm.MK_COMB, Drule.MK_EXISTS.

HALF_MK_PABS (PairRules)

```
HALF_MK_PABS : (thm -> thm)
```

Synopsis
Converts a function definition to lambda-form.

Description
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 the variables in the quantified pair is free in that function.

See also
`Drule.HALF_MK_ABS`, `PairRules.PETA_CONV`, `PairRules.MK_PABS`, `PairRules.MK_PEXISTS`.

hash (Lib)

```
hash : int -> string -> int * int -> int
```

Synopsis
Hash function for strings.

Description
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);
> val it = 5 : int
```

Comments
For better results, the `i` parameter should be a prime.

This is probably not an industrial strength hash function.

hidden (Parse)

```
hidden : string -> bool
```

Synopsis
Checks to see if a given name has been hidden.

Description
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)

```
hide : string -> ({Name : string, Thy : string} list *
                  {Name : string, Thy : string} list)
```

Synopsis
Stops the quotation parser from recognizing a constant.

Description
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.

See also

`Parse.hidden`, `Parse.known_constants`, `Parse.reveal`, `Parse.set_known_constants`, `Parse.update_overload_maps`.

Hol_datatype (bossLib)

```
Hol_datatype : type quotation -> unit
```

Synopsis

Define a concrete datatype.

Description

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 `Hol_datatype ‘<spec>‘`, where `<spec>` should conform to the following

grammar:

```
spec    ::= [ <binding> ; ]* <binding>

binding ::= <ident> = [ <clause> | ]* <clause>
          |  <ident> = <| [ <ident> : <type> ; ]* <ident> : <type> |>

clause  ::= <ident>
          |  <ident> of [<type> => ]* <type>
```

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. 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

```
Hol_datatype `btree = Leaf of 'a
                    | Node of btree => 'b => btree`
```

The => notation in a HOL datatype description is intended to replace * in an ML datatype description, and highlights the fact that, in HOL, constructors are by default curried. Note also that any type parameters for the new type are not allowed; they are inferred (in an arbitrary order) from the right hand side of the binding.

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 `Hol_datatype`.

To start, enumerated types can be defined as in the following example:

```
Hol_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:

```
Hol_datatype `foo = N of num
                  | B of bool
                  | Fn of 'a -> 'b
                  | Pr of 'a # 'b`
```

Turning to recursive types, we can define a type of binary trees where the leaves are numbers.

```
- Hol_datatype `tree = Leaf of num
                     | Node of 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.

```
Hol_datatype `tree = Leaf of 'a
                   | Node of 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

- Hol_datatype `lambda = Var of string
                       | Const of 'a
                       | Comb of lambda => lambda
                       | Abs of lambda => lambda`
```

The syntax for 'de Bruijn' terms is roughly similar:

```
Hol_datatype `dB = Var of string
                 | Const of 'a
                 | Bound of num
                 | Comb  of dB => dB
                 | Abs   of 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.

```
Hol_datatype `ntree = Node of 'a => ntree list`
```

A type of 'first order terms' can be declared as follows:

```
Hol_datatype `term = Var of string
                  | Fnapp of 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.

```
Hol_datatype
     `atexp = var_exp of string
            | let_exp of dec => exp ;

        exp = aexp    of atexp
            | app_exp of exp => atexp
            | fn_exp  of match ;

      match = match  of rule
            | matchl of rule => match ;

       rule = rule of pat => exp ;

        dec = val_dec   of valbind
            | local_dec of dec => dec
            | seq_dec   of dec => dec ;

    valbind = bind  of pat => exp
            | bindl of pat => exp => valbind
            | rec_bind of valbind ;

        pat = wild_pat
            | var_pat of string`
```

Simple record types may be introduced using the `<| ... |>` notation.

```
Hol_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.

```
Hol_datatype
     `file = Text of string
           | Dir of directory
       ;
 directory = <| owner : string ;
                files : (string # file) list |>`
```

Failure

Now we address some types that cannot be declared with `Hol_datatype`. In some cases they cannot exist in HOL at all; in others, the type can be built in the HOL logic, but `Hol_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.

```
Hol_datatype ‘foo = A of foo‘
```

So called ‘nested types’, which are occasionally quite useful, cannot at present be built with `Hol_datatype`:

```
Hol_datatype ‘btree = Leaf of ’a
                    | Node of  (’a # ’a) btree‘
```

Co-inductive types may not currently be built with `Hol_datatype`:

```
Hol_datatype ‘lazylist = Nil
                       | Cons of ’a # (one -> lazylist)‘
```

This type can however 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):

```
Hol_datatype ‘lambda = Var of string
                     | Const of ’a
                     | Comb of lambda => lambda
                     | Abs of 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 `Hol_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

```
Hol_datatype `tree = Leaf of num
                   | Node of tree => tree`;
```

results in the definitions

```
tree_case_def =
  |- (!f f1 a. case f f1 (Leaf a) = f a) /\
      !f f1 a0 a1. case f f1 (Node a0 a1) = 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))
       ==>
       (case f f1 M = case f' f1' M')
```

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`.

Hol_defn (bossLib)

```
Hol_defn : string -> term quotation -> defn
```

Synopsis

General-purpose function definition facility.

Description

`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 prop-

erly 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 a 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 wellfounded 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 `ltree`s. 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`, `DefnBase.read_congs`, `DefnBase.write_congs`, `bossLib.Hol_datatype`.

Hol_defn (Defn)

```
Hol_defn : string -> term quotation -> thm
```

Synopsis
Function definition facility.

Description
`bossLib.Hol_defn` is identical to `Defn.Hol_defn`.

See also
`bossLib.Hol_defn`.

HOL_ERR (Feedback)

```
HOL_ERR : {message : string, origin_function : string,
           origin_structure : string} -> exn
```

Synopsis
Standard HOL exception

Description
`HOL_ERR` is the single exception that HOL functions are expected to raise when they encounter an anomalous situation.

Example
Building an application of `HOL_ERR` and binding it to an ML variable

```
val test_exn =
   HOL_ERR {origin_structure = "Foo",
            origin_function = "bar",
            message = "incomprehensible input"};
```

yields

```
val test_exn = HOL_ERR : exn
```

One can scrutinize the contents of an application of `HOL_ERR` by pattern matching:

```
- val HOL_ERR r = test_exn;

> val r = {message = "incomprehensible input",
           origin_function = "bar",
           origin_structure = "Foo"}
```

In current ML implementations supporting HOL, exceptions that propagate to the top

level without being handled do not print informatively:

```
- raise test_exn;
! Uncaught exception:
! HOL_ERR
```

In such cases, the functions `Raise` and `exn_to_string` can be used to obtain useful information:

```
- Raise test_exn;

Exception raised at Foo.bar:
incomprehensible input
! Uncaught exception:
! HOL_ERR

- print(exn_to_string test_exn);

Exception raised at Foo.bar:
incomprehensible input
> val it = () : unit
```

See also

`Feedback`, `Feedback.error_record`, `Feedback.mk_HOL_ERR`, `Feedback.Raise`, `Feedback.exn_to_string`.

HOL_MESG (Feedback)

```
HOL_MESG : string -> unit
```

Synopsis

Prints out a message in a special format.

Description

`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 (bossLib)

```
Hol_reln : term quotation -> (thm * thm * thm)
```

Synopsis
Defines inductive relations.

Description
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>)
<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_reln (IndDefLib)

```
Hol_reln : term quotation -> thm * thm * thm
```

Synopsis
Definition facility for inductive predicates.

Description
bossLib.Hol_reln is identical to IndDefLib.Hol_reln.

See also
bossLib.Hol_reln.

hol_type (Type)

```
eqtype hol_type
```

Synopsis
Type of HOL types.

Description
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)

```
HOL_WARNING : string -> string -> string -> unit
```

Synopsis
Prints out a message in a special format.

Description
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.>>
```

See also
`Feedback`, `Feedback.HOL_ERR`, `Feedback.Raise`, `Feedback.HOL_MESG`, `Feedback.WARNING_to_string`, `Feedback.format_WARNING`, `Feedback.WARNING_outstream`.

hyp (Thm)

```
hyp : thm -> term list
```

Synopsis
Returns the hypotheses of a theorem.

Description
When applied to a theorem `A |- t`, the function `hyp` returns `A`, the list of hypotheses of the theorem.

Failure
Never fails.

See also
`Thm.dest_thm`, `Thm.concl`.

I (Lib)

```
I : 'a -> 'a
```

Synopsis
Performs identity operation: `I x` = `x`.

Failure
Never fails.

See also
`Lib`, `Lib.##`, `Lib.A`, `Lib.B`, `Lib.C`, `Lib.K`, `o`, `Lib.S`, `Lib.W`.

IMP_ANTISYM_RULE (Drule)

```
IMP_ANTISYM_RULE : thm -> thm -> thm
```

Synopsis
Deduces equality of boolean terms from forward and backward implications.

Description
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)

```
IMP_CANON : (thm -> thm list)
```

Synopsis
Puts theorem into a 'canonical' form.

Description
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)

```
IMP_CONJ : (thm -> thm -> thm)
```

Synopsis
Conjoins antecedents and consequents of two implications.

Description
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_ELIM (Drule)

```
IMP_ELIM : (thm -> thm)
```

Synopsis
Transforms `|- s ==> t` into `|- ~s \/ t`.

Description
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)

```
IMP_RES_FORALL_CONV : conv
```

Synopsis
Converts an implication to a restricted universal quantification.

Description
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)

```
IMP_RES_TAC : thm_tactic
```

Synopsis
Enriches assumptions by repeatedly resolving an implication with them.

Description
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

`Thm_cont.IMP_RES_THEN`, `Drule.RES_CANON`, `Tactic.RES_TAC`, `Thm_cont.RES_THEN`.

IMP_RES_THEN (Thm_cont)

```
IMP_RES_THEN : thm_tactical
```

Synopsis

Resolves an implication with the assumptions of a goal.

Description

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)

```
IMP_TRANS : (thm -> thm -> thm)
```

Synopsis

Implements the transitivity of implication.

Description

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.`

implication (boolSyntax)

```
implication : term
```

Synopsis
Constant denoting logical implication.

Description
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.disjunction, boolSyntax.negation,`
`boolSyntax.conditional, boolSyntax.bool_case, boolSyntax.let_tm,`
`boolSyntax.arb.`

implicit_rewrites (Rewrite)

```
implicit_rewrites: unit -> rewrites
```

Synopsis
Contains a number of theorems used, by default, in rewriting.

Description
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`.

Uses
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`.

ind (Type)

```
ind : hol_type
```

Synopsis
Basic type constant.

Description
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 IndDefRules
```

Synopsis
Tom Melham's inference support for inductive definitions

Description
`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)

```
index : ('a -> bool) -> 'a list -> int
```

Synopsis
Finds index of first list element for which predicate holds.

Description
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 raised at Lib.index:
no such element
! Uncaught exception:
! HOL_ERR

- index (equal 3 o hd) [[1],[],[2,3]];
! Uncaught exception:
! Empty
```

See also
`Lib.el`.

Induct (bossLib)

```
Induct : tactic
```

Synopsis

Performs structural induction over the type of the goal's outermost universally quantified variable.

Description

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 exam-

ple, 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`, `goalstackLib.g`, `goalstackLib.e`.

Induct (SingleStep)

```
Induct : tactic
```

Synopsis

Induct on leading universally quantified variable in a goal.

Description

`bossLib.Induct` is identical to `SingleStep.Induct`.

See also

`bossLib.Induct`.

Induct_on (bossLib)

```
Induct_on : term -> tactic
```

Synopsis

Performs structural induction, using the type of the given term.

Description

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 datattype.

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`, `goalstackLib.g`, `goalstackLib.e`.

Induct_on (SingleStep)

```
Induct_on : term -> tactic
```

Synopsis

Induct on given term.

Description

`bossLib.Induct_on` is identical to `SingleStep.Induct_on`.

See also
bossLib.Induct_on.

INDUCT_TAC (numLib)

```
INDUCT_TAC : tactic
```

Synopsis
Performs tactical proof by mathematical induction on the natural numbers.

Description
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.

See also
INDUCT.

INDUCT_THEN (Prim_rec)

```
INDUCT_THEN : (thm -> thm_tactic -> tactic)
```

Synopsis
Structural induction tactic for automatically-defined concrete types.

Description

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 ith constructor to a sequence of variables, then if ttac is a function that maps the induction hypotheses hypi of the ith 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

`Datatype.define_type`, `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`.

bool_compset (computeLib)

```
bool_compset : unit -> compset
```

Synopsis
Creates a new simplification set to use with `computeLib.CBV_CONV` for basic computations.

Description
This function creates a new simplification set to use with the compute library performing computations about operations on primitive booleans and numerals (in binary representation) such as LET, conditional, implication, conjunction, disjunction, negation, `FST`, `SND`, addition, subtraction, multiplication, division, modulo, exponentiation, etc.

We assume here that the canonical representation of the naturals is the binary one. Therefore, defining function by pattern matching using SUC will not be recognized. For instance, defining the exponentaition function as

```
|- (n EXP 0 = 1) /\ (n EXP (SUC p) = n * n EXP p)
```

It is possible to make this definition work by using the following lemma:

```
|- (exp n p = if n = 0 then 1 else n * (exp n (p-1)))
```

Example

```
- CBV_CONV (bool_compset()) (--‘EVERY (\n. EVEN n) [4;6;8;10;12;14;16]‘--);
> val it = |- EVERY (\n. EVEN n) [4; 6; 8; 10; 12; 14; 16] = T : thm
```

See also
`computeLib.CBV_CONV`, `REDUCE_CONV`.

insert (Lib)

```
insert ''a -> ''a list -> ''a list
```

Synopsis
Add an element to a list if it is not already there.

Description
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. Serious implementations of sets may be found in the Standard ML Basis Library.

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`.

inst (Term)

```
inst : (hol_type,hol_type)subst -> term -> term
```

Synopsis

Performs type instantiations in a term.

Description

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 = ‘\(x’ :bool). (x :bool)‘ : term
```

See also
`Type.type_subst`, `Lib.|->`.

INST (Thm)

```
INST : (term,term) subst -> thm -> thm
```

Synopsis
Instantiates free variables in a theorem.

Description
`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_TY_TERM (Drule)

```
INST_TY_TERM :
(term,term)subst * (hol_type,hol_type)subst -> thm -> thm
```

Synopsis

Instantiates terms and types of a theorem.

Description

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.

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.

INST_TYPE (Thm)

```
INST_TYPE : (hol_type,hol_type) subst -> thm -> thm
```

Synopsis
Instantiates types in a theorem.

Description
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 variables becoming identical after the instantiation.

Failure
Never fails.

Uses
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.|->.

int_of_string (hol88Lib)

```
Compat.int_of_string : string -> int
```

Synopsis
Maps a string of numbers to the corresponding integer.

Description
Found in the hol88 library. Given a string representing an integer in standard decimal notation, possibly including a leading plus sign or minus sign and/or leading zeros, `int_of_string` returns the corresponding integer constant.

Failure
Fails unless the string is a valid decimal representation as specified above. It will not be found unless the hol88 library has been loaded.

Comments
Not found in hol90, since the author always got it backwards; use `string_to_int` instead. Likewise, `string_of_int` is not found in hol90; use `int_to_string`.

See also
`ascii`, `ascii_code`, `hol88Lib.string_of_int`, `int_to_string`, `string_to_int`.

int_sort (Lib)

```
int_sort : int list -> int list
```

Synopsis
Sorts a list of integers using the <= relation.

Description
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)

```
int_to_string : int -> string
```

Synopsis
Translates an integer into a string.

Description
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 ML Standard Basis Library function `Int.toString`.

See also
`Lib.string_to_int`.

intersect (Lib)

```
intersect : ''a list -> ''a list -> ''a list
```

Synopsis
Computes the intersection of two 'sets'.

Description
`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.

High performance finite set operations may be found in the ML Standard Basis Library.

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)

```
IPSPEC : (term -> thm -> thm)
```

Synopsis
Specializes a theorem, with type instantiation if necessary.

Description

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, match.

IPSPECL (PairRules)

```
IPSPECL : (term list -> thm -> thm)
```

Synopsis

Specializes a theorem zero or more times, with type instantiation if necessary.

Description

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, MATCH, Thm.SPEC, PairRules.PSPECL.

is_abs (Term)

```
is_abs : (term -> bool)
```

Synopsis
Tests a term to see if it is an abstraction.

Description
`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)

```
is_arb : term -> bool
```

Synopsis
Tests a term to see if it's an instance of `ARB`.

Description
Returns `true` if and only if `M` has the form `ARB`.

Uses
None known.

See also
boolSyntax.mk_arb, boolSyntax.dest_arb.

is_bool_case (boolSyntax)

```
is_bool_case : term -> bool
```

Synopsis
Tests a case expression over `bool`.

Description
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)

```
is_comb : term -> bool
```

Synopsis
Tests a term to see if it is a combination (function application).

Description
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)

```
is_cond : term -> bool
```

Synopsis
Tests a term to see if it is a conditional.

Description
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)

```
is_conj : term -> bool
```

Synopsis
Tests a term to see if it is a conjunction.

Description
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)

```
is_cons : (term -> bool)
```

Synopsis
Tests a term to see if it is an application of `CONS`.

Description
`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)

```
is_const : term -> bool
```

Synopsis
Tests a term to see if it is a constant.

Description
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)

```
is_disj : term -> bool
```

Synopsis
Tests a term to see if it is a disjunction.

Description
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_eq (boolSyntax)

```
is_eq : term -> bool
```

Synopsis
Tests a term to see if it is an equation.

Description
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)

```
is_exists : term -> bool
```

Synopsis
Tests a term to see if it is an existential quantification.

Description
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)

```
is_exists1 : term -> bool
```

Synopsis
Tests a term to see if it is a unique existence term.

Description
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)

```
is_forall : term -> bool
```

Synopsis
Tests a term to see if it is a universal quantification.

Description
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)

```
is_gen_tyvar : hol_type -> bool
```

Synopsis
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)

```
is_genvar : term -> bool
```

Synopsis
Tells if a variable has been built by invoking `genvar`.

Description
`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)

```
is_imp : term -> bool
```

Synopsis

Tests a term to see if it is an implication or a negation.

Description

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)

```
is_imp_only : term -> bool
```

Synopsis
Tests a term to see if it is an implication.

Description
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)

```
is_let : term -> bool
```

Synopsis
Tests a term to see if it is a `let`-expression.

Description
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`;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it = `LET f x` : term

- is_let it;
> val it = true : bool

- is_let (Term `let x = P /\ Q in x \/ x`);
> val it = true : bool
```

See also
`boolSyntax.mk_let`, `boolSyntax.dest_let`.

is_list (listSyntax)

```
is_list : (term -> bool)
```

Synopsis
Tests a term to see if it is a list.

Description
`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)

```
is_neg : term -> bool
```

Synopsis
Tests a term to see if it is a negation.

Description
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_pabs (pairSyntax)

```
is_pabs : term -> bool
```

Synopsis
Tests a term to see if it is a paired abstraction.

Description
`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)

```
is_pair : (term -> bool)
```

Synopsis
Tests a term to see if it is a pair.

Description
`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)

```
is_pexists : (term -> bool)
```

Synopsis
Tests a term to see if it as a paired existential quantification.

Description
`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.mk_pexists`, `pairSyntax.dest_pexists`.

is_pforall (pairSyntax)

```
is_pforall : (term -> bool)
```

Synopsis
Tests a term to see if it is a paired universal quantification.

Description
`is_pforall "!pair. t"` returns `true`. If the term is not a a paired universal quantification the result is `false`.

Failure
Never fails.

See also
`boolSyntax.is_forall`, `pairSyntax.mk_pforall`, `pairSyntax.dest_pforall`.

is_prod (pairSyntax)

```
is_prod : hol_type -> bool
```

Synopsis
Tests a type to see if it is a product type.

Description
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)

```
is_pselect : (term -> bool)
```

Synopsis
Tests a term to see if it is a paired choice-term.

Description
`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.mk_pselect`, `pairSyntax.dest_pselect`.

is_pvar (pairSyntax)

```
is_pvar : (term -> bool)
```

Synopsis
Tests a term to see if it is a paired structure of variables.

Description
`is_pvar "pvar"` returns `true` iff `pvar` is a paired structure of variables. For example, `((a:*,b:*),(d:*,e:*))` is a paired structure of variables, `(1,2)` is not.

Failure
Never fails.

See also
`Term.is_var`.

is_res_abstract (res_quanLib)

```
is_res_abstract : term -> bool
```

Synopsis
Tests a term to see if it is a restricted abstraction.

Description
`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)

```
is_res_exists : term -> bool
```

Synopsis
Tests a term to see if it is a restricted existential quantification.

Description
`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)

```
is_res_exists_unique : term -> bool
```

Synopsis
Tests a term to see if it is a restricted unique existential quantification.

Description
`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)

```
is_res_forall : term -> bool
```

Synopsis
Tests a term to see if it is a restricted universal quantification.

Description
`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)

```
is_res_select : term -> bool
```

Synopsis
Tests a term to see if it is a restricted choice quantification.

Description
`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)

```
is_select : (term -> bool)
```

Synopsis
Tests a term to see if it is a choice binding.

Description
`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)

```
is_type : hol_type -> bool
```

Synopsis
Tests whether a HOL type is not a type variable.

Description
`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)

```
is_var : term -> bool
```

Synopsis
Tests a term to see if it is a variable.

Description
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)

```
is_vartype : type -> bool
```

Synopsis
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)

```
isEmpty : tag -> bool
```

Synopsis

Tells if a tag is empty.

Description

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 = true : bool
```

See also

Thm.tag, Thm.mk_oracle_thm.

ISPEC (Drule)

```
ISPEC : (term -> thm -> thm)
```

Synopsis
Specializes a theorem, with type instantiation if necessary.

Description
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)

```
ISPECL : term list -> thm -> thm
```

Synopsis
Specializes a theorem zero or more times, with type instantiation if necessary.

Description
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)

```
type ('a,'b) istream
```

Synopsis
Type of imperative streams

Description
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)

```
itlist : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Synopsis
List iteration function. Applies a binary function between adjacent elements of a list.

Description
`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)

```
itlist2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
```

Synopsis
Applies a function to corresponding elements of 2 lists.

Description
`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)

```
K : 'a -> 'b -> 'a
```

Synopsis
Forms a constant function: `K x y = x`.

Failure
Never fails.

See also
`Lib.##, Lib.A, Lib.B, Lib.C, Lib.I, Lib.S, Lib.W.`

known_constants (Parse)

```
Parse.known_constants : unit -> string list
```

Synopsis
Returns the list of constants known to the parser.

Description
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.`

LAND_CONV (Conv)

```
LAND_CONV : conv -> conv
```

Synopsis
Applies a conversion to the left-hand argument of a binary operator.

Description

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, or if `tm` has this form but the conversion `c` fails when applied to the term `t2`. 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 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`.

last (Lib)

```
last : 'a list -> 'a
```

Synopsis

Computes the last element of a list.

Description

`last [x1,...,xn]` returns `xn`.

Failure

Fails if the list is empty.

See also

`Lib.butlast`, `Lib.el`, `Lib.front_last`.

LAST_EXISTS_CONV (Conv)

```
Conv.LAST_EXISTS_CONV : conv -> conv
```

Synopsis
Applies a conversion to the last existential quantifier (and its body) in a chain.

Description
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 term `t`, then the result of the whole will be ``` ``?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.STRIP_QUANT_CONV`.

LEFT_AND_EXISTS_CONV (Conv)

```
LEFT_AND_EXISTS_CONV : conv
```

Synopsis
Moves an existential quantification of the left conjunct outwards through a conjunction.

Description
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)

```
LEFT_AND_FORALL_CONV : conv
```

Synopsis
Moves a universal quantification of the left conjunct outwards through a conjunction.

Description
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)

```
LEFT_AND_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the left conjunct outwards through a conjunction.

Description
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)

```
LEFT_AND_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the left conjunct outwards through a conjunction.

Description
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)

```
LEFT_IMP_EXISTS_CONV : conv
```

Synopsis
Moves an existential quantification of the antecedent outwards through an implication.

Description
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)

```
LEFT_IMP_FORALL_CONV : conv
```

Synopsis
Moves a universal quantification of the antecedent outwards through an implication.

Description
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)

```
LEFT_IMP_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the antecedent outwards through an implication.

Description
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)

```
LEFT_IMP_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the antecedent outwards through an implication.

Description
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)

```
LEFT_LIST_PBETA : (thm -> thm)
```

Synopsis
Iteratively beta-reduces a top-level paired beta-redex on the left-hand side of an equation.

Description
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)

```
LEFT_OR_EXISTS_CONV : conv
```

Synopsis
Moves an existential quantification of the left disjunct outwards through a disjunction.

Description
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)

```
LEFT_OR_FORALL_CONV : conv
```

Synopsis
Moves a universal quantification of the left disjunct outwards through a disjunction.

Description
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)

```
LEFT_OR_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the left disjunct outwards through a disjunction.

Description
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)

```
LEFT_OR_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the left disjunct outwards through a disjunction.

Description
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)

```
LEFT_PBETA : (thm -> thm)
```

Synopsis

Beta-reduces a top-level paired beta-redex on the left-hand side of an equation.

Description

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.

let_tm (boolSyntax)

```
let_tm : term
```

Synopsis

Constant denoting let expressions.

Description
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`.

lhs (boolSyntax)

```
lhs : term -> term
```

Synopsis
Returns the left-hand side of an equation.

Description
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`.

doc.

Lib.doc

```
structure Lib
```

Synopsis
Collection of commonly used functions

Description
`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`.

LIST_BETA_CONV (Drule)

```
LIST_BETA_CONV : conv
```

Synopsis
Performs an iterated beta conversion.

Description
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 = |- (\x 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)

```
list_compare : ('a * 'a -> order) -> 'a list * 'a list -> order
```

Synopsis
Lifts a comparison function to a lexicographic ordering on lists.

Description

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)

```
LIST_CONJ : thm list -> thm
```

Synopsis

Conjoins the conclusions of a list of theorems.

Description

```
        A1 |- t1 ... An |- tn
   ---------------------------------  LIST_CONJ
    A1 u ... u An |- t1 /\ ... /\ tn
```

Failure

`LIST_CONJ` fails if applied to an empty list of theorems.

Comments

`LIST_CONJ` does not check for alpha-equivalence of assumptions when forming their union. If a particular assumption is duplicated within one of the input theorems assumption lists, then it may be duplicated in the resulting assumption list.

See also
`Drule.BODY_CONJUNCTS`, `Thm.CONJ`, `Thm.CONJUNCT1`, `Thm.CONJUNCT2`, `Drule.CONJUNCTS`, `Drule.CONJ_PAIR`, `Tactic.CONJ_TAC`.

list_mk_abs (boolSyntax)

```
list_mk_abs : term list * term -> term
```

Synopsis
Iteratively constructs abstractions.

Description
`list_mk_abs([x1,...,xn],t)` returns the term `\x1 ... xn.t`.

Failure
Fails if the terms in the list are not variables.

See also
`boolSyntax.strip_abs`, `Term.mk_abs`.

list_mk_abs (Term)

```
list_mk_abs : term list * term -> term
```

Synopsis
Performs a sequence of lambda binding operations.

Description
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 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 = `\v1 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_binder (Term)

```
list_mk_binder : term option -> term list * term -> term
```

Synopsis

Performs a sequence of variable binding operations on a term

Description

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)

```
list_mk_comb : term * term list -> term
```

Synopsis

Iteratively constructs combinations (function applications).

Description

`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 raised at Term.list_mk_comb:
incompatible types
```

See also

`boolSyntax.strip_comb`, `Term.mk_comb`.

list_mk_conj (boolSyntax)

```
list_mk_conj : term list -> term
```

Synopsis
Constructs the conjunction of a list of terms.

Description
`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 raised at boolSyntax.mk_conj:
Non-boolean argument

- list_mk_conj [mk_var("x",alpha)];
> val it = ‘x‘ : term
```

See also
`boolSyntax.strip_conj`, `boolSyntax.mk_conj`.

list_mk_disj (boolSyntax)

```
list_mk_disj : term list -> term
```

Synopsis
Constructs the conjunction of a list of terms.

Description
`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 raised at boolSyntax.mk_disj:
Non-boolean argument

- list_mk_disj [mk_var("x",alpha)];
> val it = ‘x‘ : term
```

See also

`boolSyntax.strip_disj`, `boolSyntax.mk_disj`.

list_mk_exists (boolSyntax)

```
list_mk_exists : term list * term -> term
```

Synopsis

Iteratively constructs an existential quantification.

Description

`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_EXISTS (Drule)

```
LIST_MK_EXISTS : (term list -> thm -> thm)
```

Synopsis
Multiply existentially quantifies both sides of an equation using the given variables.

Description
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_forall (boolSyntax)

```
list_mk_forall : term list * term -> term
```

Synopsis
Iteratively constructs a universal quantification.

Description
`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)

```
list_mk_fun : hol_type list * hol_type -> hol_type
```

Synopsis
Iteratively constructs function types.

Description
`list_mk_fun([ty1,...,tyn],ty)` returns `ty1 -> ( ... (tyn -> t)...)`.

Failure
Never fails.

Example

```
- list_mk_fun ([alpha,bool],beta);
> val it = ‘:’a -> bool -> ’b‘ : hol_type
```

See also
`boolSyntax.strip_fun`, `Type.mk_type`, `Type.-->`.

list_mk_imp (boolSyntax)

```
list_mk_imp : term list * term -> term
```

Synopsis
Iteratively constructs implications.

Description
`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));
evaluation failed     list_mk_imp

- list_mk_imp ([],mk_var("x",alpha));
> val it = ‘x‘ : term
```

See also

`boolSyntax.strip_imp`, `boolSyntax.mk_imp`.

list_mk_pabs (pairSyntax)

```
list_mk_pabs : term list * term -> term
```

Synopsis

Iteratively constructs paired abstractions.

Description

`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)

```
list_mk_pair : term list -> term
```

Synopsis

Constructs a tuple from a list of terms.

Description

`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)

```
LIST_MK_PEXISTS : (term list -> thm -> thm)
```

Synopsis

Multiply existentially quantifies both sides of an equation using the given pairs.

Description

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)

```
LIST_MK_PFORALL : (term list -> thm -> thm)
```

Synopsis
Multiply universally quantifies both sides of an equation using the given pairs.

Description
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)

```
list_mk_res_exists : ((term # term) list # term) -> term)
```

Synopsis
Iteratively constructs a restricted existential quantification.

Description

```
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)

```
list_mk_res_forall : (term # term) list # term) -> term
```

Synopsis
Iteratively constructs a restricted universal quantification.

Description

```
   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_MP (Drule)

```
LIST_MP : (thm list -> thm -> thm)
```

Synopsis
Performs a chain of Modus Ponens inferences.

Description

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)

```
LIST_PBETA_CONV : conv
```

Synopsis

Performs an iterated paired beta-conversion.

Description

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

```
- LIST_PBETA_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
```

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_ss (bossLib)

```
list_ss : simpset
```

Synopsis

Simplification set for lists.

Description

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)

```
listDB : unit -> data list
```

Synopsis
All theorems, axioms, and definitions in the currently loaded theory segments.

Description
An invocation `listDB()` returns everything that has been stored in all theory segments currently loaded.

Example

```
- length (listDB());
> val it = 736 : int
```

See also
DB.thy, DB.theorems, DB.definitions, DB.axioms, DB.find, DB.match.

map2 (Lib)

```
map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

Synopsis
Maps a function over two lists to create one new list.

Description
`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.

MAP_EVERY (Tactical)

```
MAP_EVERY : ((* -> tactic) -> * list -> tactic)
```

Synopsis

Sequentially applies all tactics given by mapping a function over a list.

Description

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)

```
MAP_FIRST : ((* -> tactic) -> * list -> tactic)
```

Synopsis
Applies first tactic that succeeds in a list given by mapping a function over a list.

Description
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)

```
mapfilter : ('a -> 'b) -> 'a list -> 'b list
```

Synopsis
Applies a function to every element of a list, returning a list of results for those elements for which application succeeds.

Failure
If `f x` raises `Interrupt` for some element `x` of `l`, then `mapfilter f l` fails.

Example

```
- mapfilter hd [[1,2,3],[4,5],[],[6,7,8],[]];
> val it = [1, 4, 6] : int list
```

See also

`Lib.filter`, `Lib.gather`.

match (DB)

```
match : string list -> term -> data list
```

Synopsis

Attempt to find matching theorems in the specified theories.

Description

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", "EQ_CLAUSES"),
      (|- !t.((T = t) = t) /\ ((t = T) = t) /\
             ((F = t) = ~t) /\ ((t = F) = ~t), Db.Thm)),
     (("bool", "EQ_EXPAND"),
      (|- !t1 t2. (t1 = t2) = t1 /\ t2 \/ ~t1 /\ ~t2, Db.Thm)),
     (("bool", "EQ_IMP_THM"),
      (|- !t1 t2. (t1 = t2) = (t1 ==> t2) /\ (t2 ==> t1), Db.Thm)),
     (("bool", "EQ_SYM_EQ"), (|- !x y. (x = y) = (y = x), Db.Thm)),
     (("bool", "FUN_EQ_THM"), (|- !f g. (f = g) = !x. f x = g x, Db.Thm)),
     (("bool", "OR_IMP_THM"), (|- !A B. (A = B \/ A) = B ==> A, Db.Thm)),
     (("bool", "REFL_CLAUSE"), (|- !x. (x = x) = T, Db.Thm)),
     (("pair", "CLOSED_PAIR_EQ"),
      (|- !x y a b. ((x,y) = (a,b)) = (x = a) /\ (y = b), Db.Thm)),
     (("pair", "CURRY_ONE_ONE_THM"),
      (|- (CURRY f = CURRY g) = (f = g), Db.Thm)),
     (("pair", "PAIR_EQ"), (|- ((x,y) = (a,b)) = (x = a) /\ (y = b), Db.Thm)),
     (("pair", "UNCURRY_ONE_ONE_THM"),
      (|- (UNCURRY f = UNCURRY g) = (f = g), Db.Thm))] :
  ((string * string) * (thm * class)) list
```

Comments

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

Uses

For locating theorems when doing interactive proof.

See also

DB.matcher, DB.matchp, DB.find, DB.theorems, Db.thy, Db.listDB.

MATCH_ACCEPT_TAC (Tactic)

```
MATCH_ACCEPT_TAC : thm_tactic
```

Synopsis

Solves a goal which is an instance of the supplied theorem.

Description

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
```

See also

`Tactic.ACCEPT_TAC`.

MATCH_MP (Drule)

```
MATCH_MP : thm -> thm -> thm
```

Synopsis

Modus Ponens inference rule with automatic matching.

Description

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'
```

Failure

Fails unless the first theorem is a (possibly repeatedly 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.

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

`Thm.EQ_MP`, `Tactic.MATCH_MP_TAC`, `Thm.MP`, `Tactic.MP_TAC`.

MATCH_MP_TAC (Tactic)

```
MATCH_MP_TAC : thm_tactic
```

Synopsis
Reduces the goal using a supplied implication, with matching.

Description
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...tm. 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'.

See also
Thm.EQ_MP, Drule.MATCH_MP, Thm.MP, Tactic.MP_TAC.

match_term (Term)

```
match_term : term -> term -> (term,term) subst * (hol_type,hol_type) subst
```

Synopsis
Finds instantiations to match one term to another.

Description
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.

Example

The following shows how `match_term` could be used to match the conclusion of a theorem to a term.

```
- val th = REFL (Term `x:'a`);
 val th = |- x = x : thm

- match_term (concl th) (Term `1 = 1`);
val it = ([{redex = `x`, residue = `1`}],
          [{redex = `:'a`, residue = `:num`}])
  : term subst * hol_type subst

- INST_TY_TERM it th;
val it = |- 1 = 1
```

Comments

For instantiating theorems `PART_MATCH` is usually easier to use.

See also

`Type.match_type`, `Drule.INST_TY_TERM`, `Drule.PART_MATCH`.

match_terml (Term)

```
match_terml
     : hol_type list -> term set -> term -> term
       -> (term,term) subst * (hol_type,hol_type) subst
```

Synopsis

Match two terms while restricting some instantiations.

Description

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 `redex`es 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 (S,T) = match_terml [] empty_varset
                  (Term `\x:'a. x = f (y:'b)`)
                  (Term `\a.    a = ~p`);
> val S = [{redex = `(f :'b -> 'a)`, residue = `$~`},
           {redex = `(y :'b)`,       residue = `(p :bool)`}] : ...

  val T = [{redex = `:'b`, residue = `:bool`},
           {redex = `:'a`, residue = `:bool`}] : ...

- match_terml [alpha] empty_varset  (* forbid instantiation of 'a *)
          (Term `\x:'a. x = f (y:'b)`)
          (Term `\a.    a = ~p`);
! Uncaught exception:
! HOL_ERR

- match_terml [] (HOLset.add(empty_varset,mk_var("y",beta)))
          (Term `\x:'a. x = f (y:'b)`)
          (Term `\a.    a = ~p`);
! Uncaught exception:
! HOL_ERR
```

See also

`Term.match_term`, `Term.raw_match`, `Term.subst`, `Term.inst`, `Type.match_typel`, `Type.type_subst`.

match_type (Type)

```
match_type : hol_type -> hol_type -> hol_type subst
```

Synopsis

Calculates a substitution `theta` such that instantiating the first argument with `theta` equals the second argument.

Description

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 = ‘:’a‘, residue = ‘:num‘}] : hol_type 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);
! Uncaught exception:
! HOL_ERR
```

See also

`Term.match_term`, `HolKernel.ho_match_term`, `Type.type_subst`.

match_typel (Type)

```
match_typel : hol_type list -> hol_type -> hol_type
        -> (hol_type, hol_type) subst
```

Synopsis

Match types with restrictions.

Description

An invocation `match_typel away pat ty` matches `pat` to `ty` in the same way as `match_type`, but prohibits any of the type variables in `away` from being instantiated. In effect, the elements of `away`, although type variables, are treated as constants in `pat` during the matching process.

Failure

An invocation of `match_typel away pat ty` will fail if `match_type pat ty` would fail. It will also fail if `match_type pat ty` would succeed giving a substitution `[{redex_1,residue_1},...,`
where one or more of the `redex_i` are members of `away`.

Example

In the first example, we perform a normal match operation

```
- match_typel [] (alpha --> beta --> gamma)
                  (bool --> ind --> delta);
> val it = [{redex = ':'c', residue = ':'d'},
            {redex = ':'b', residue = ':ind'},
            {redex = ':'a', residue = ':bool'}] : ...
```

Now we require that `gamma`, although a type variable in the pattern, not be instantiable. In the first try, the match succeeds because `'c` is mapped only to itself. In the second, it

fails because an association is made between `'c` and `'d`.

```
- match_typel [gamma] (alpha --> beta --> gamma)
                      (bool --> ind --> gamma);
> val it = [{redex = ‘:'b‘, residue = ‘:ind‘},
            {redex = ‘:'a‘, residue = ‘:bool‘}] : ...

- match_typel [gamma] (alpha --> beta --> gamma)
                      (bool --> ind --> delta);
! Uncaught exception:
! HOL_ERR
```

Comments

The use of `away` allows matching to take account of type variables that are 'frozen' (by occurring in the hypotheses of a theorem, for example). This allows certain fruitless proof attempts to be avoided at the matching stage.

See also

`Type.match_type`, `Term.match_term.`, `HolKernel.ho_match_term`, `Type.type_subst`.

matcher (DB)

```
matcher : (term -> term -> 'a) -> string list -> term -> data list
```

Synopsis

All theory elements matching a given term.

Description

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", "RC_def"), (|- !R x y. RC R x y = (x = y) \/ R x y, Def)),
     (("relation", "RTC_CASES1"),
      (|- !R x y. RTC R x y = (x = y) \/ ?u. R x u /\ RTC R u y, Thm)),
     (("relation", "RTC_CASES2"),
      (|- !R x y. RTC R x y = (x = y) \/ ?u. RTC R x u /\ R u y, Thm)),
     (("relation", "RTC_TC_RC"),
      (|- !R x y. RTC R x y ==> RC R x y \/ TC R x y, Thm)),
     (("relation", "TC_CASES1"),
      (|- !R x z. TC R x z ==> R x z \/ ?y. R x y /\ TC R y z, Thm)),
     (("relation", "TC_CASES2"),
      (|- !R x z. TC R x z ==> R x z \/ ?y. TC R x y /\ R y z, Thm))] :
  ((string * string) * (thm * class)) list

- 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)),
     (("bool", "EXISTS_OR_THM"),
      (|- !P Q. (?x. P x \/ Q x) = (?x. P x) \/ ?x. Q x, Thm)),
     (("bool", "LEFT_OR_EXISTS_THM"),
      (|- !P Q. (?x. P x) \/ Q = ?x. P x \/ Q, Thm)),
     (("bool", "RIGHT_OR_EXISTS_THM"),
      (|- !P Q. P \/ (?x. Q x) = ?x. P \/ Q x, Thm)),
     (("sum", "IS_SUM_REP"),
      (|- !f.
            IS_SUM_REP f =
            ?v1 v2.
              (f = (\b x y. (x = v1) /\ b)) \/ (f = (\b x y. (y = v2) /\ ~b)),
       Def))] : ((string * string) * (thm * class)) list
```

Comments

Usually, `pm` will be a pattern-matcher, but it need not be.

See also

`DB.match`, `DB.apropos`, `DB.matchp`, `DB.find`.

matchp (DB)

```
matchp : (thm -> bool) -> string list -> data list
```

Synopsis

All theory elements satisfying a predicate.

Description

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", "CLOSED_PAIR_EQ"),
      (|- !x y a b. ((x,y) = (a,b)) = (x = a) /\ (y = b), Thm)),
     (("pair", "COMMA_DEF"), (|- !x y. (x,y) = ABS_prod (MK_PAIR x y), Def)),
     (("pair", "CURRY_DEF"), (|- !f x y. CURRY f x y = f (x,y), Def)),
     (("pair", "CURRY_ONE_ONE_THM"), (|- (CURRY f = CURRY g) = (f = g), Thm)),
     (("pair", "CURRY_UNCURRY_THM"), (|- !f. CURRY (UNCURRY f) = f, Thm)),
     (("pair", "ELIM_PEXISTS"),
      (|- (?p. P (FST p) (SND p)) = ?p1 p2. P p1 p2, Thm)),
     (("pair", "ELIM_PFORALL"),
      (|- (!p. P (FST p) (SND p)) = !p1 p2. P p1 p2, Thm)),
     (("pair", "ELIM_UNCURRY"),
      (|- !f. UNCURRY f = (\x. f (FST x) (SND x)), Thm)),
     (("pair", "EXISTS_PROD"), (|- (?p. P p) = ?p_1 p_2. P (p_1,p_2), Thm)),
     (("pair", "FORALL_PROD"), (|- (!p. P p) = !p_1 p_2. P (p_1,p_2), Thm)),
     (("pair", "FST"), (|- !x y. FST (x,y) = x, Thm)),
     (("pair", "IS_PAIR_DEF"),
      (|- !P. IS_PAIR P = ?x y. P = MK_PAIR x y, Def)),
     (("pair", "LAMBDA_PROD"),
      (|- !P. (\p. P p) = (\(p1,p2). P (p1,p2)), Thm)),
     (("pair", "LET2_RAND"),
      (|- !P M N. P (let (x,y) = M in N x y) = (let (x,y) = M in P (N x y)),
       Thm)),
     (("pair", "LET2_RATOR"),
      (|- !M N b. (let (x,y) = M in N x y) b = (let (x,y) = M in N x y b),
       Thm)),
     (("pair", "LEX_DEF"),
      (|- !R1 R2. R1 LEX R2 = (\(s,t) (u,v). R1 s u \/ (s = u) /\ R2 t v),
       Def)),
     (("pair", "MK_PAIR_DEF"),
      (|- !x y. MK_PAIR x y = (\a b. (a = x) /\ (b = y)), Def)),
     (("pair", "PAIR"), (|- !x. (FST x,SND x) = x, Def)),
     (("pair", "pair_case_def"), (|- case = UNCURRY, Def)),
     (("pair", "pair_case_thm"), (|- case f (x,y) = f x y, Thm)),
     (("pair", "PAIR_EQ"), (|- ((x,y) = (a,b)) = (x = a) /\ (y = b), Thm)),
     (("pair", "PAIR_MAP"),
      (|- !f g p. (f ## g) p = (f (FST p),g (SND p)), Def)),
     (("pair", "PAIR_MAP_THM"),
      (|- !f g x y. (f ## g) (x,y) = (f x,g y), Thm)),
     (("pair", "PEXISTS_THM"), (|- !P. (?x y. P x y) = ?(x,y). P x y, Thm)),
     (("pair", "PFORALL_THM"), (|- !P. (!x y. P x y) = !(x,y). P x y, Thm)),
     (("pair", "RPROD_DEF"),
      (|- !R1 R2. RPROD R1 R2 = (\(s,t) (u,v). R1 s u /\ R2 t v), Def)),
     (("pair", "SND"), (|- !x y. SND (x,y) = y, Thm)),
     (("pair", "UNCURRY"), (|- !f v. UNCURRY f v = f (FST v) (SND v), Def)),
     (("pair", "UNCURRY_CURRY_THM"), (|- !f. UNCURRY (CURRY f) = f, Thm)),
     (("pair", "UNCURRY_DEF"), (|- !f x y. UNCURRY f (x,y) = f x y, Thm)),
     (("pair", "UNCURRY_ONE_ONE_THM"),
      (|- (UNCURRY f = UNCURRY g) = (f = g), Thm)),
     (("pair", "UNCURRY_VAR"),
```

max_print_depth (Globals)

```
max_print_depth : int ref
```

Synopsis
Sets depth bound on prettyprinting.

Description
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 (iZ (n + m))) /\
       (!n. 0 * n = 0) /\ (!n. n * 0 = 0) /\
       (!n m. ... ... * ... ... = NUMERAL (... * ...)) /\
       (!n. ... - ... = 0) /\ (!n. ... = ...) /\ (!... .... ...) /\ ... /\ ...
     : thm
```

measureInduct_on (bossLib)

```
measureInduct_on : term quotation -> tactic
```

Synopsis
Perform complete induction with a supplied measure function.

Description
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)

```
mem : ''a -> ''a list -> bool
```

Synopsis
Tests whether a list contains a certain member.

Description
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.

High performance finite set operations may be found in the ML Standard Basis Library.

See also
`Lib.op_mem`, `Lib.insert`, `Lib.find`, `Lib.tryfind`, `Lib.exists`, `Lib.all`, `Lib.assoc`, `Lib.rev_assoc`.

merge (Tag)

```
merge : tag -> tag -> tag
```

Synopsis
Combine two tags into one.

Description
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 = Kerneltypes.TAG(["bar", "foo"], []) : tag

- Tag.merge it (Tag.read "foo");
> val it = Kerneltypes.TAG(["bar", "foo"], []) : 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)

```
MESG_outstream : TextIO.outstream ref
```

Synopsis
Reference to output stream used when printing HOL_MESG

Description
The value of reference cell MESG_outstream controls where HOL_MESG prints its argument.

The default value of MESG_outstream is TextIO.stdOut.

Example

```
- val ostrm = TextIO.openOut "foo";
> val ostrm = <outstream> : outstream

- MESG_outstream := ostrm;
> val it = () : unit

- HOL_MESG "Nattering nabobs of negativity.";
> val it = () : unit

- TextIO.closeOut ostrm;
> val it = () : unit

- val istrm = TextIO.openIn "foo";
> val istrm = <instream> : instream

- print (TextIO.inputAll istrm);
<<HOL message: Nattering nabobs of negativity.>>
```

See also
Feedback, Feedback.HOL_MESG, Feedback.ERR_outstream, Feedback.WARNING_outstream, Feedback.emit_MESG.

MESG_to_string (Feedback)

```
MESG_to_string : (string -> string) ref
```

Synopsis
Alterable function for formatting HOL_MESG

Description
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)

```
MESON_TAC : thm list -> tactic
```

Synopsis

Performs first order proof search to prove the goal, using the given theorems as additional assumptions in the search.

Description

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.

Uses

`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`.

MK_ABS (Drule)

```
MK_ABS : (thm -> thm)
```

Synopsis

Abstracts both sides of an equation.

Description

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, Drule.HALF_MK_ABS, Thm.MK_COMB, Drule.MK_EXISTS.

mk_abs (Term)

```
mk_abs : term * term -> term
```

Synopsis
Constructs an abstraction.

Description
`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_arb (boolSyntax)

```
mk_arb : hol_type -> term
```

Synopsis
Creates a type instance of the `ARB` constant.

Description
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_bool_case (boolSyntax)

```
mk_bool_case : term * term * term -> term
```

Synopsis
Constructs a case expression over bool.

Description
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`);
<<HOL message: inventing new type variable names: 'a, 'b>>

> val it = `case x of T -> f x || F -> b` : term
```

See also
boolSyntax.dest_bool_case, boolSyntax.is_bool_case.

mk_comb (Term)

```
mk_comb : term * term -> term
```

Synopsis
Constructs a combination (function application).

Description
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)

```
MK_COMB : thm * thm -> thm
```

Synopsis
Proves equality of combinations constructed from equal functions and operands.

Description
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)

```
MK_COMB_TAC : tactic
```

Synopsis
Breaks an equality between applications into two equality goals: one for the functions, and other for the arguments.

Description
`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_THM_TAC`.

mk_cond (boolSyntax)

```
mk_cond : term * term * term -> term
```

Synopsis
Constructs a conditional term.

Description
`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)

```
mk_conj : term * term -> term
```

Synopsis
Constructs a conjunction.

Description
`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)

```
mk_cons : {hd :term, tl :term} -> term
```

Synopsis
Constructs a `CONS` pair.

Description
`mk_cons{hd = t, tl = ‘[t1;...;tn]‘}` returns `‘[t;t1;...;tn]‘`.

Failure
Fails if `tl` is not a list or if `hd` 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)

```
mk_const : string * hol_type -> term
```

Synopsis

Constructs a constant.

Description

If `n` is a string that has been previously declared to be a constant with type `ty` and 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)

```
mk_disj : term * term -> term
```

Synopsis

Constructs a disjunction.

Description

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)

```
mk_eq : term * term -> term
```

Synopsis

Constructs an equation.

Description
`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 (boolSyntax)

```
mk_exists : term * term -> term
```

Synopsis
Term constructor for existential quantification.

Description
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_EXISTS (Drule)

```
MK_EXISTS : (thm -> thm)
```

Synopsis
Existentially quantifies both sides of a universally quantified equational theorem.

Description

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_exists1 (boolSyntax)

```
mk_exists1 : term * term -> term
```

Synopsis

Term constructor for unique existence.

Description

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)

```
mk_forall : term * term -> term
```

Synopsis

Term constructor for universal quantification.

Description

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.strip_forall`.

mk_HOL_ERR (Feedback)

```
mk_HOL_ERR : string -> string -> string -> exn
```

Synopsis

Creates an application of `HOL_ERR`.

Description

`mk_HOL_ERR` provides a curried interface to the standard `HOL_ERR` exception; experience has shown that this is often more convenient.

Failure

Never fails.

Example

```
- mk_HOL_ERR "Module" "function" "message"

> val it = HOL_ERR : exn

- print(exn_to_string it);

Exception raised at Module.function:
message
> val it = () : unit
```

See also

`Feedback`, `Feedback.HOL_ERR`, `Feedback.error_record`.

mk_imp (boolSyntax)

```
mk_imp : term * term -> term
```

Synopsis
Constructs an implication.

Description
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)

```
mk_istream : ('a -> 'a) -> 'a -> ('a -> 'b) -> ('a,'b) istream
```

Synopsis
Create a stream.

Description
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)

```
mk_let : term * term -> term
```

Synopsis
Constructs a `let` term.

Description
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, pairLib.`

mk_list (listSyntax)

```
mk_list : {els : term list, ty : hol_type} -> term
```

Synopsis
Constructs an object-level (HOL) list from an ML list of terms.

Description
`mk_list{els = [t1, ..., tn], ty = 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)

```
mk_neg : (term -> term)
```

Synopsis
Constructs a negation.

Description
`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_oracle_thm (Thm)

```
mk_oracle_thm : tag -> term list * term -> thm
```

Synopsis
Construct a theorem without proof, and tag it.

Description

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`.

Example

In the following, we construct a tag and then make a rogue rule of inference.

```
- val tag = Tag.read "SimonSays";
> val tag = Kerneltypes.TAG(["SimonSays"], []) : tag

- 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 (Tag.read "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.mk_thm`, `Tag.read`, `Thm.tag`, `Theory.mk_axiom`.

MK_PABS (PairRules)

```
MK_PABS : (thm -> thm)
```

Synopsis

Abstracts both sides of an equation.

Description

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)

```
mk_pabs : term * term -> term
```

Synopsis

Constructs a paired abstraction.

Description

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)

```
MK_PAIR : thm -> thm -> thm
```

Synopsis
Proves equality of pairs constructed from equal components.

Description
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)

```
mk_pair : term * term -> term
```

Synopsis
Constructs object-level pair from a pair of terms.

Description
`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)

```
MK_PEXISTS : (thm -> thm)
```

Synopsis
Existentially quantifies both sides of a universally quantified equational theorem.

Description

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)

```
MK_PFORALL : (thm -> thm)
```

Synopsis

Universally quantifies both sides of a universally quantified equational theorem.

Description

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)

```
mk_primed_var : string * hol_type -> term
```

Synopsis
Primes a variable name sufficiently to make it distinct from all constants.

Description
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`, `Globals.priming`.

mk_prod (pairSyntax)

```
mk_prod : hol_type * hol_type -> hol_type
```

Synopsis
Constructs a product type from two constituent types.

Description
`mk_prod(ty1, ty2)` returns `ty1 # t2`.

Failure
Never fails.

See also
pairSyntax.is_prod, pairSyntax.dest_prod.

MK_PSELECT (PairRules)

```
MK_PSELECT : (thm -> thm)
```

Synopsis
Quantifies both sides of a universally quantified equational theorem with the choice quantifier.

Description
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_res_abstract (res_quanLib)

```
mk_res_abstract : (term # term # term) -> term
```

Synopsis
Term constructor for restricted abstraction.

Description
`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)

```
mk_res_exists : ((term # term # term) -> term)
```

Synopsis
Term constructor for restricted existential quantification.

Description
`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)

```
mk_res_exists_unique : (term # term # term) -> term
```

Synopsis
Term constructor for restricted unique existential quantification.

Description
`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)

```
mk_res_forall : (term # term # term) -> term
```

Synopsis
Term constructor for restricted universal quantification.

Description
`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)

```
mk_res_select : (term # term # term) -> term
```

Synopsis
Term constructor for restricted choice quantification.

Description
`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)

```
mk_select : term * term -> term
```

Synopsis
Constructs a choice-term.

Description
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)

```
mk_set : ''a list -> ''a list
```

Synopsis
Transforms a list into one with distinct elements.

Description
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. Serious implementations of sets may be found in the Standard ML Basis Library.

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 : ssdata list -> simpset
```

Synopsis

Creates a simpset by combining a list of `ssdata` values.

Failure

Never fails.

Uses

Creates simpsets, which are a necessary argument to any simplification function.

See also

`simpLib.++`, `simpLib.rewrites`, `simpLib.SIMP_CONV`.

mk_thm (Thm)

```
mk_thm : term list * term -> thm
```

Synopsis
Creates an arbitrary theorem (dangerous!)

Description
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)

```
mk_thy_const : {Thy:string, Name:string, Ty:hol_type} -> term
```

Synopsis
Constructs a constant.

Description
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)

```
mk_thy_type
       : {Thy:string, Name:string, Args:hol_type list} -> hol_type
```

Synopsis
Constructs a type.

Description

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 = ‘:'a -> bool‘ : hol_type

- try mk_thy_type {Tyop="bar", Thy="foo", Args = []};

Exception raised at Type.mk_thy_type:
"foo$bar" not found
```

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, decls, op_arity.

mk_type (Type)

```
mk_type : string * hol_type list -> hol_type
```

Synopsis

Constructs a compound type.

Description

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 = ‘:'a -> 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_var (Term)

```
mk_var : string * hol_type -> term
```

Synopsis
Constructs a variable of given name and type.

Description
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)

```
mk_vartype : string -> hol_type
```

Synopsis
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";

Exception raised at Type.mk_vartype:
incorrect syntax
```

See also
`Type.dest_vartype`, `Type.is_vartype`, `Type.mk_type`.

mlquote (Lib)

```
mlquote : string -> string
```

Synopsis
Put quotation marks around a string.

Description
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

```
- print (quote "foo\nbar" ^ "\n");
"foo
bar"
> val it = () : unit

- print (mlquote "foo\nbar" ^ "\n");
"foo\nbar"
> val it = () : unit
```

See also
`Lib.quote`.

monitoring (computeLib)

```
monitoring : (term -> bool) option ref
```

Synopsis
Monitoring support for evaluation

Description
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);

- EVAL (Term ‘FACT 4‘);
FACT 4 = (if 4 = 0 then 1 else 4 * FACT (PRE 4))
FACT 3 = (if 3 = 0 then 1 else 3 * FACT (PRE 3))
FACT 2 = (if 2 = 0 then 1 else 2 * FACT (PRE 2))
FACT 1 = (if 1 = 0 then 1 else 1 * FACT (PRE 1))
FACT 0 = (if 0 = 0 then 1 else 0 * FACT (PRE 0))
> val it = |- FACT 4 = 24 : thm
```

See also

`computeLib.RESTR_EVAL_CONV`, `Term.decls`.

MP (Thm)

```
MP : thm -> thm -> thm
```

Synopsis

Implements the Modus Ponens inference rule.

Description

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
```

Failure

Fails unless the first theorem is an implication whose antecedent is the same as the conclusion of the second theorem (up to alpha-conversion).

See also

`Thm.EQ_MP`, `Drule.LIST_MP`, `Drule.MATCH_MP`, `Tactic.MATCH_MP_TAC`, `Tactic.MP_TAC`.

MP_TAC (Tactic)

```
MP_TAC : thm_tactic
```

Synopsis
Reduces a goal to implication from a known theorem.

Description
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`.

NEG_DISCH (Drule)

```
NEG_DISCH : term -> thm -> thm
```

Synopsis
Discharges an assumption, transforming `|- s ==> F` into `|- ~s`.

Description
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`.

negation (boolSyntax)

```
negation : term
```

Synopsis
Constant denoting logical negation.

Description
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`.

new_axiom (Theory)

```
new_axiom : string * term -> thm
```

Synopsis
Install a new axiom in the current theory.

Description
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)

```
new_binder : string * hol_type -> unit
```

Synopsis
Sets up a new binder in the current theory.

Description
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";
() : unit

- new_binder ("!!", (bool-->bool)-->bool);;
() : 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)

```
new_binder_definition : string * term -> thm
```

Synopsis
Defines a new constant, giving it the syntactic status of a binder.

Description
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))‘);

> val it = |- $?! = (\P. $? P /\ (!x y. P x /\ P y ==> (x = y))) : thm
```

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)

```
new_constant : string * hol_type -> unit
```

Synopsis
Declares a new constant in the current theory.

Description
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)

```
new_definition : string * term -> thm
```

Synopsis
Declare a new constant and install a definitional axiom in the current theory.

Description
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)

```
new_infix : string * hol_type * int -> unit
```

Synopsis

Declares a new infix constant in the current theory.

Description

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`, `precedence`, `Theory.constants`, `infixes`, `binders`, `is_constant`, `Theory.new_constant`, `boolSyntax.new_binder`, `Definition.new_definition`, `new_infix_definition`, `boolSyntax.new_binder_definition`.

new_infixl_definition (boolSyntax)

```
new_infixl_definition : string * term * int -> thm
```

Synopsis

Declares a new left associative infix constant and installs a definition in the current theory.

Description

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)

```
new_infixr_definition : string * term * int -> thm
```

Synopsis

Declares a new right associative infix constant and installs a definition in the current theory.

Description

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)

```
new_recursive_definition :
{name:string,def:term,rec_axiom:thm} -> thm
```

Synopsis

Defines a primitive recursive function over a concrete recursive type.

Description

`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 ith 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
```

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

Hol_datatype, Prim_rec.prove_rec_fn_exists, TotalDefn.Define, Parse.set_fixity.

new_specification (Definition)

```
new_specification :
   {name : string,
    sat_thm : thm,
    consts : {const_name:string, fixity:fixity} list} -> thm
```

Synopsis

Introduces a constant or constants satisfying a given property.

Description

The ML function new_specification implements the primitive rule of constant specifi-

cation for the HOL logic. Evaluating:

```
new_specification {name=name, sat_thm = |- ?x1...xn. t,
                   consts = [{const_name = "c1", fixity = f1}, ...,
                             {const_name = "cn", fixity = fn}]}
```

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` The fixities `f1`, ..., `fn` are values which determine the parsing status of the new constants. Typical fixity values are `Prefix`, `Binder`, `Infixl n`, `Infixr n`, `Suffix n`, `TruePrefix n` or `Closefix`.

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. Finally, failure occurs if some `ci` does not contain all the type variables that occur in the term `?x1...xn. t`.

Uses

`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:

```
- val DIVISION =
     new_specification
         {name = "DIVISION",
          consts = [{fixity = Infixl 650, const_name = "MOD"},
                    {fixity = Infixl 600, const_name = "DIV"}],
          sat_thm = th};
```

This introduces the constants `MOD` and `DIV` with the defining property shown above.

See also

`Definition.new_definition`, `boolSyntax.new_binder_definition`, `boolSyntax.new_infixl_definition`, `boolSyntax.new_infixr_definition`, `TotalDefn.Define`.

new_theory (Theory)

```
new_theory : string -> unit
```

Synopsis

Creates a new theory segment.

Description

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. 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.

Uses

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, DB.print_theory, Theory.save_thm, Theory.export_theory, Theory.after_new_theory.

new_type (Theory)

```
new_type : string * int -> unit
```

Synopsis

Declares a new type or type constructor.

Description

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)

```
new_type_definition : string * thm -> thm
```

Synopsis

Defines a new type constant or type operator.

Description

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 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`. Failure also occurs if the type of `t` is not of the form `ty->bool`.

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 repre-

sentations 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
   {name = "THREE",
    sat_thm = PROVE [abs_distinct]
                    (Term‘?x y z:three. ~(x=y) /\ ~(y=z) /\ ~(z=x)‘),
    consts = [{const_name="e1", fixity=Prefix},
              {const_name="e2", fixity=Prefix},
              {const_name="e3", fixity=Prefix}]};

> 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)

```
next : ('a,'b) istream -> ('a,'b) istream
```

Synopsis

Move to the next element in the stream.

Description

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 (concat "gsym" o int_to_string);
> val it = <istream> : (int, string) istream

- next istrm;
> val it = <istream> : (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)

```
NO_CONV : conv
```

Synopsis
Conversion that always fails.

Failure
`NO_CONV` always fails.

See also
`Conv.ALL_CONV`.

NO_TAC (Tactical)

```
NO_TAC : tactic
```

Synopsis
Tactic which always fails.

Description
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)

```
NO_THEN : thm_tactical
```

Synopsis
Theorem-tactical which always fails.

Description
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!)

Uses
Writing compound tactics or tacticals.

See also
Tactical.ALL_TAC, Thm_cont.ALL_THEN, Tactical.FAIL_TAC, Tactical.NO_TAC.

norm_subst (Term)

```
norm_subst : (hol_type,hol_type) subst
        -> (term,term) subst -> (term,term)subst
```

Synopsis
Instantiate term substitution by a type substitution.

Description

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,(T,Id))`. The `Id` component can be ignored. The `S` component is a substitution for term variables, but it has to be instantiated by `T` in order to be suitable for use by `INST_TY_TERM`. In this case, one uses `norm_subst T S`. Thus a suitable input for `INST_TY_TERM` would be `(norm_subst T S, T)`.

Failure

Never fails.

Example

```
- val (S,(T,_)) = raw_match [] empty_varset
                    (Term `\x:'a. x = f (y:'b)`)
                    (Term `\a.    a = ~p`) ([],([],[]));
> val S = [{redex = `(f :'b -> 'a)`, residue = `$~`},
           {redex = `(y :'b)`,       residue = `(p :bool)`}] : ...

  val T = [{redex = `:'b`, residue = `:bool`},
           {redex = `:'a`, residue = `:bool`}] : ...

- norm_subst T S;
> val it =
    [{redex = `(y :bool)`, residue = `(p :bool)`},
     {redex = `(f :bool -> bool)`, residue = `$~`}]
  : {redex : term, residue : term} list
```

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`.

NOT_ELIM (Thm)

```
NOT_ELIM : thm -> thm
```

Synopsis

Transforms `|- ~t` into `|- t ==> F`.

Description

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)

```
NOT_EQ_SYM : (thm -> thm)
```

Synopsis

Swaps left-hand and right-hand sides of a negated equation.

Description

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)

```
NOT_EXISTS_CONV : conv
```

Synopsis
Moves negation inwards through an existential quantification.

Description
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)

```
NOT_FORALL_CONV : conv
```

Synopsis
Moves negation inwards through a universal quantification.

Description
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)

```
NOT_INTRO : (thm -> thm)
```

Synopsis
Transforms `|- t ==> F` into `|- ~t`.

Description
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)

```
NOT_PEXISTS_CONV : conv
```

Synopsis
Moves negation inwards through a paired existential quantification.

Description
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)

```
NOT_PFORALL_CONV : conv
```

Synopsis
Moves negation inwards through a paired universal quantification.

Description
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`.

null_intersection (Lib)

```
null_intersection : ''a list -> ''a list -> bool
```

Synopsis
Tells if two lists have no common elements.

Description
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

High performance finite set operations may be found in the ML Standard Basis Library.

See also

`Lib.intersect`, `Lib.union`, `Lib.U`, `Lib.mk_set`, `Lib.mem`, `Lib.insert`, `Lib.set_eq`, `Lib.set_diff`.

occs_in (pairSyntax)

```
occs_in : (term -> term -> bool)
```

Synopsis

Occurrence check for bound variables.

Description

When applied to two terms `p` and `t`, where `p` is a paired structure of variables, the function `occs_in` returns `true` if and of the constituent variables of `p` occurs free in `t`, and `false` otherwise.

Failure

Fails of `p` is not a paired structure of variables.

See also

`Term.free_in`, `hol88Lib.frees`, `hol88Lib.freesl`, `thm_frees`.

ONCE_ASM_REWRITE_RULE (Rewrite)

```
ONCE_ASM_REWRITE_RULE : (thm list -> thm -> thm)
```

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

Description
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.

Uses
This tactic is used when rewriting with the hypotheses of a theorem (as well as a given list of theorems and basic_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)

```
ONCE_ASM_REWRITE_TAC : (thm list -> tactic)
```

Synopsis
Rewrites a goal once including built-in rewrites and the goal's assumptions.

Description
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)
```

Uses

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_CONV (Conv)

```
ONCE_DEPTH_CONV : (conv -> conv)
```

Synopsis

Applies a conversion once to the first suitable sub-term(s) encountered in top-down order.

Description

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.

Uses

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)

```
ONCE_REWRITE_CONV : (thm list -> conv)
```

Synopsis

Rewrites a term, including built-in tautologies in the list of rewrites.

Description

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 basic_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.

Uses

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)

```
ONCE_REWRITE_RULE : (thm list -> thm -> thm)
```

Synopsis
Rewrites a theorem, including built-in tautologies in the list of rewrites.

Description
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 basic_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.

Uses
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)

```
ONCE_REWRITE_TAC : (thm list -> tactic)
```

Synopsis
Rewrites a goal only once with basic_rewrites and the supplied list of theorems.

Description
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)
```

Uses

`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`, `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`.

op_arity (Type)

```
op_arity : {Thy:string, Tyop:string} -> int option
```

Synopsis

Return the arity of a type operator.

Description

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)

```
op_insert ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list
```

Synopsis
Add an element to a list if it is not already there.

Description
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`, `\z. z /\ y`, `F`] : term list

- op_insert aconv (Term `\x. x /\ y`)
                  [T, Term `\z. z /\ a`, F];
> val it = [`\x. x /\ y`, `T`, `\z. z /\ a`, `F`] : term list
```

Comments
There is no requirement that `eq` be recognizable as a kind of equality (it could be imple-

mented by an order relation, for example).

One should not write code that depends on the arrangement of elements in the result.

High performance finite set operations may be found in the ML Standard Basis Library.

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)

```
op_intersect : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

Synopsis

Computes the intersection of two 'sets'.

Description

`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 = [`\x. x`] : term list
```

Comments

The order of items in the list returned by `op_intersect` is not dependable.

High performance finite set operations may be found in the ML Standard Basis Library.

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)

```
op_mem : ('a -> 'a -> bool) -> 'a -> 'a list -> bool
```

Synopsis
Tests whether a list contains a certain element.

Description
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
High performance finite set operations may be found in the ML Standard Basis Library.

See also
`Lib.mem`, `Lib.op_insert`, `Lib.find`, `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)

```
op_mk_set : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Synopsis
Transforms a list into one with elements that are distinct modulo the supplied relation.

Description
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 = [`\z a. z /\ a`] : term list
```

Comments
The order of items in the list returned by `op_mk_set` is not dependable.

Serious implementations of sets may be found in the Standard ML Basis Library.

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)

```
op_set_diff : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

Synopsis
Computes the set-theoretic difference of two 'sets', modulo a supplied relation.

Description
`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.

High performance set operations may be found in the ML Standard Basis Library.

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)

```
op_U : ('a -> 'a -> bool) -> 'a list list -> 'a list
```

Synopsis

Takes the union of a list of sets, modulo the supplied relation.

Description

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.

High performance set operations may be found in the ML Standard Basis Library.

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`.

union (Lib)

```
union : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

Synopsis

Computes the union of two 'sets'.

Description

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).

High performance set operations may be found in the ML Standard Basis Library.

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_EXISTS_CONV (Conv)

```
OR_EXISTS_CONV : conv
```

Synopsis
Moves an existential quantification outwards through a disjunction.

Description
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)

```
OR_FORALL_CONV : conv
```

Synopsis
Moves a universal quantification outwards through a disjunction.

Description
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)

```
OR_PEXISTS_CONV : conv
```

Synopsis

Moves a paired existential quantification outwards through a disjunction.

Description

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)

```
OR_PFORALL_CONV : conv
```

Synopsis

Moves a paired universal quantification outwards through a disjunction.

Description

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 ORELSE : tactic * tactic -> tactic
```

Synopsis
Applies first tactic, and if it fails, applies the second instead.

Description
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.THEN`.

ORELSE_TCL (Thm_cont)

```
$ORELSE_TCL : (thm_tactical -> thm_tactical -> thm_tactical)
```

Synopsis
Applies a theorem-tactical, and if it fails, tries a second.

Description
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)

```
$ORELSEC : (conv -> conv -> conv)
```

Synopsis
Applies the first of two conversions that succeeds.

Description
`(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"`.

Failure
`(c1 ORELSEC c2) "t"` fails both `c1` and `c2` fail when applied to `"t"`.

See also
Conv.FIRST_CONV.

overload_on (Parse)

```
Parse.overload_on : string * term -> unit
```

Synopsis
Establishes a constant as one of the overloading possibilities for a string.

Description
Calling `overload_on(name,tm)` establishes `tm` as a possible resolution of the overloaded `name`. The term `tm` must be a constant. 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.

Finally, 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.

Failure

Fails if the term argument is not a constant.

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
```

Comments

Overloading with abandon can lead to input that is very hard to make sense of, and so should be used with caution.

See also

Parse.clear_overloads_on.

p (goalstackLib)

```
p : unit -> goalstack
```

Synopsis

Prints the top levels of the subgoal package goal stack.

Description

The function p is part of the subgoal package. It is an abbreviation for the function print_state. For a description of the subgoal package, see set_goal.

Failure

Never fails.

Uses

Examining the proof state during an interactive proof session.

See also

goalstackLib.b, goalstackLib.backup, backup_limit, goalstackLib.e, goalstackLib.expand, goalstackLib.expandf, goalstackLib.g, get_state, print_state, goalstackLib.r, rotate, save_top_thm, goalstackLib.set_goal, set_state, goalstackLib.top_goal, goalstackLib.top_thm.

P_FUN_EQ_CONV (PairRules)

```
P_FUN_EQ_CONV : (term -> conv)
```

Synopsis
Performs extensionality conversion for functions (function equality).

Description
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)

```
P_PCHOOSE_TAC : (term -> thm_tactic)
```

Synopsis
Assumes a theorem, with existentially quantified pair replaced by a given witness.

Description
`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)

```
P_PCHOOSE_THEN : (term -> thm_tactical)
```

Synopsis

Replaces existentially quantified pair with given witness, and passes it to a theorem-tactic.

Description

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)

```
P_PGEN_TAC : (term -> tactic)
```

Synopsis

Specializes a goal with the given paired structure of variables.

Description

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, PGEN_ALL, PairRules.PSPEC, PairRules.PSPECL, PairRules.PSPEC_ALL, PairRules.PSPEC_TAC.

P_PSKOLEM_CONV (PairRules)

```
P_PSKOLEM_CONV : (term -> conv)
```

Synopsis

Introduces a user-supplied Skolem function.

Description

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)

```
PABS : (term -> thm -> thm)
```

Synopsis

Paired abstraction of both sides of an equation.

Description

```
      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

```
- PABS (Term `(x:'a,y:'b)`) (REFL (Term `(x:'a,y:'b)`));
> val it = |- (\(x,y). (x,y)) = (\(x,y). (x,y)) : thm
```

See also

Thm.ABS, PairRules.PABS_CONV, PairRules.PETA_CONV, PairRules.PEXT, PairRules.MK_PABS.

PABS_CONV (PairRules)

```
PABS_CONV : conv -> conv
```

Synopsis

Applies a conversion to the body of a paired abstraction.

Description

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

```
- PABS_CONV SYM_CONV (Term `\(x,y). (1,2) = (x,y)`);
> val it = |- (\(x,y). (1,2) = (x,y)) = (\(x,y). (x,y) = (1,2)) : thm
```

See also

Conv.ABS_CONV, PairRules.PSUB_CONV.

paconv (pairSyntax)

```
paconv : (term -> term -> bool)
```

Synopsis
Tests for alpha-equivalence of terms.

Description
When applied to a pair of terms `t1` and `t2`, `paconv` returns true if the terms are alpha-equivalent.

Failure
Never fails.

Comments
`paconv` is implemented as `curry (can (uncurry PALPHA))`.

See also
`PairRules.PALPHA`, `Term.aconv`.

pair (Lib)

```
pair : 'a -> 'b -> 'a * 'b
```

Synopsis
Makes two values into a pair.

Description
`pair x y` returns `(x,y)`.

Failure
Never fails.

See also
`Lib.fst`, `Lib.snd`, `Lib.curry`, `Lib.uncurry`.

PAIR_CONV (PairRules)

```
PAIR_CONV : (conv -> conv)
```

Synopsis
Applies a conversion to all the components of a pair structure.

Description
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`.

PAIRED_BETA_CONV (PairedLambda)

```
PAIRED_BETA_CONV : conv
```

Synopsis
Performs generalized beta conversion for tupled beta-redexes.

Description
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`, `PairedLambda.GEN_BETA_CONV`, `Drule.LIST_BETA_CONV`, `Drule.RIGHT_BETA`, `Drule.RIGHT_LIST_BETA`.

PAIRED_ETA_CONV (PairedLambda)

```
PAIRED_ETA_CONV : conv
```

Synopsis

Performs generalized eta conversion for tupled eta-redexes.

Description

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

Thm.ETA_CONV.

PALPHA (PairRules)

```
PALPHA : term -> term -> thm
```

Synopsis

Proves equality of paired alpha-equivalent terms.

Description

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

```
- PALPHA (Term `\(x:'a,y:'a). (x,y)`) (Term`\xy:'a#'a. xy`);
> val it = |- (\(x,y). (x,y)) = (\xy. 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)

```
PALPHA_CONV : term -> conv
```

Synopsis

Renames the bound variables of a paired lambda-abstraction.

Description

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)

```
parents : string -> string list
```

Synopsis
Lists the parent theories of a named theory.

Description
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 = "scratch" : string

- parents "-";
> val it = ["list", "option"] : string list
```

See also
`Theory.ancestry`, `Theory.current_theory`.

parse_from_grammars (Parse)

```
parse_from_grammars :
  (parse_type.grammar * term_grammar.grammar) ->
  ((hol_type frag list -> hol_type) * (term frag list -> term))
```

Synopsis
Returns parsing functions based on the supplied grammars.

Description

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 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
```

Uses

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.Term, Type.

parse_in_context (Parse)

```
Parse.parse_in_context : term list -> term quotation -> term
```

Synopsis

Parses a quotation into a term, using the terms as typing context.

Description

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 >>
```

Uses

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)

```
PART_MATCH : (term -> term) -> thm -> term -> thm
```

Synopsis

Instantiates a theorem by matching part of it to a term.

Description

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.

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

`Thm.INST_TYPE`, `Drule.INST_TY_TERM`, `Drule.HO_PART_MATCH`, `Term.match_term`.

PART_PMATCH (PairRules)

```
PART_PMATCH : ((term -> term) -> thm -> term -> thm)
```

Synopsis

Instantiates a theorem by matching part of it to a term.

Description

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)

```
partial : exn -> ('a -> 'b option) -> 'a -> 'b
```

Synopsis
Converts a total function to a partial function.

Description
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";
! Uncaught exception:
! Fail  "not convertable"

- (total o partial (Fail "not convertable")) Int.fromString "foo";
> val it = NONE : int option
```

See also
`Lib.total`.

partition (Lib)

```
partition : ('a -> bool) -> 'a list -> 'a list * 'a list
```

Synopsis
Split a list by a predicate

Description
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];
! Uncaught exception:
! Fail  ""
```

See also
`Lib.split_after`, `Lib.pluck`.

PAT_ASSUM (Tactical)

```
PAT_ASSUM : term -> thm_tactic -> tactic
```

Synopsis
Finds the first assumption that matches the term argument, applies the theorem tactic to it, and removes this assumption.

Description
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`. Unless there is just one match otherwise, 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 (Term‘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, and would leave an equality on the assumption list of the form `x = x`.

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-indepedent 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.

See also

`Tactical.ASSUM_LIST`, `Tactical.EVERY`, `Tactical.PAT_ASSUM`, `Tactical.EVERY_ASSUM`, `Tactical.FIRST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`, `Thm_cont.UNDISCH_THEN`, `Term.match_term`, `ho_match_term`.

PBETA_CONV (PairRules)

```
PBETA_CONV : conv
```

Synopsis
Performs a general beta-conversion.

Description
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)

```
PBETA_RULE : (thm -> thm)
```

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

Description
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)

```
PBETA_TAC : tactic
```

Synopsis
Beta-reduces all the paired beta-redexes in the conclusion of a goal.

Description
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)

```
pbody : (term -> term)
```

Synopsis
Returns the body of a paired abstraction.

Description
`pbody "\pair. t"` returns `"t"`.

Failure
Fails unless the term is a paired abstraction.

See also
`Term.body`, `pairSyntax.bndpair`, `pairSyntax.dest_pabs`.

PCHOOSE (PairRules)

```
PCHOOSE : term * thm -> thm -> thm
```

Synopsis
Eliminates paired existential quantification using deduction from a particular witness.

Description

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)

```
PCHOOSE_TAC : thm_tactic
```

Synopsis

Adds the body of a paired existentially quantified theorem to the assumptions of a goal.

Description

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)

```
PCHOOSE_THEN : thm_tactical
```

Synopsis
Applies a tactic generated from the body of paired existentially quantified theorem.

Description
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`.

PETA_CONV (PairRules)

```
PETA_CONV : conv
```

Synopsis
Performs a top-level paired eta-conversion.

Description
`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)

```
PEXISTENCE : (thm -> thm)
```

Synopsis
Deduces paired existence from paired unique existence.

Description
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)

```
PEXISTS : term * term -> thm -> thm)
```

Synopsis
Introduces paired existential quantification given a particular witness.

Description
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)

```
PEXISTS_AND_CONV : conv
```

Synopsis
Moves a paired existential quantification inwards through a conjunction.

Description

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)

```
PEXISTS_CONV : conv
```

Synopsis

Eliminates paired existential quantifier by introducing a paired choice-term.

Description

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.PSELECT_INTRO, PairRules.PSELECT_ELIM.

PEXISTS_EQ (PairRules)

```
PEXISTS_EQ : (term -> thm -> thm)
```

Synopsis

Existentially quantifies both sides of an equational theorem.

Description

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)

```
PEXISTS_IMP : (term -> thm -> thm)
```

Synopsis

Existentially quantifies both the antecedent and consequent of an implication.

Description

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)

```
PEXISTS_IMP_CONV : conv
```

Synopsis

Moves a paired existential quantification inwards through an implication.

Description

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)

```
PEXISTS_NOT_CONV : conv
```

Synopsis

Moves a paired existential quantification inwards through a negation.

Description

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)

```
PEXISTS_OR_CONV : conv
```

Synopsis
Moves a paired existential quantification inwards through a disjunction.

Description
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)

```
PEXISTS_RULE : (thm -> thm)
```

Synopsis
Introduces a paired existential quantification in place of a paired choice.

Description
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)

```
PEXISTS_TAC : (term -> tactic)
```

Synopsis

Reduces paired existentially quantified goal to one involving a specific witness.

Description

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)

```
PEXISTS_UNIQUE_CONV : conv
```

Synopsis
Expands with the definition of paired unique existence.

Description
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)

```
PEXT : (thm -> thm)
```

Synopsis
Derives equality of functions from extensional equivalence.

Description
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

```
- 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)

```
PFORALL_AND_CONV : conv
```

Synopsis

Moves a paired universal quantification inwards through a conjunction.

Description

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)

```
PFORALL_EQ : (term -> thm -> thm)
```

Synopsis

Universally quantifies both sides of an equational theorem.

Description

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)

```
PFORALL_IMP_CONV : conv
```

Synopsis

Moves a paired universal quantification inwards through an implication.

Description

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)

```
PFORALL_NOT_CONV : conv
```

Synopsis

Moves a paired universal quantification inwards through a negation.

Description

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)

```
PFORALL_OR_CONV : conv
```

Synopsis
Moves a paired universal quantification inwards through a disjunction.

Description
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)

```
PGEN : (term -> thm -> thm)
```

Synopsis
Generalizes the conclusion of a theorem.

Description

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`, `PGEN_ALL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`.

PGEN_TAC (PairRules)

```
PGEN_TAC : tactic
```

Synopsis

Strips the outermost paired universal quantifier from the conclusion of a goal.

Description

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`, `PGEN_ALL`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`, `PairRules.PSTRIP_TAC`, `PairRules.P_PGEN_TAC`.

PGENL (PairRules)

```
PGENL : (term list -> thm -> thm)
```

Synopsis
Generalizes zero or more pairs in the conclusion of a theorem.

Description
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
`Drule.GENL`, `PairRules.PGEN`, `PGEN_ALL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`.

pluck (Lib)

```
pluck : ('a -> bool) -> 'a list -> 'a * 'a list
```

Synopsis
Pull an element out of a list.

Description
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 `1 <= i < k`.

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 x = 2 : int
  val rst = [1, 3] : int list
```

See also
Lib.first, Lib.gather, Lib.filter, Lib.mapfilter, Lib.assoc1, Lib.assoc2, Lib.assoc, Lib.rev_assoc.

++ (bossLib)

```
op ++ : simpset * ssdata -> simpset
```

Synopsis
Infix operator for augmenting simpsets with ssdata values.

Description
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 ssdata 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, boolSimps.UNWIND_ss.

PMATCH_MP (PairRules)

```
PMATCH_MP : (thm -> thm -> thm)
```

Synopsis
Modus Ponens inference rule with automatic matching.

Description
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)

```
PMATCH_MP_TAC : thm_tactic
```

Synopsis
Reduces the goal using a supplied implication, with matching.

Description

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)

```
polymorphic : hol_type -> bool
```

Synopsis

Checks if there is a type variable in a type

Description

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)

```
POP_ASSUM : thm_tactic -> tactic
```

Synopsis

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

Description

When applied to a theorem-tactic and a goal, `POP_ASSUM` applies the theorem-tactic to the `ASSUME`d 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 (\th. SUBST1_TAC (SYM th))
```

one might use

```
DISCH_THEN (SUBST1_TAC o SYM)
```

Example

The goal:

```
{4 = SUC x} ?- x = 3
```

can be solved by:

```
POP_ASSUM
 (fn th => REWRITE_TAC[REWRITE_RULE[num_CONV (Term‘4‘, INV_SUC_EQ] th]])
```

Uses

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, Rewrite.REWRITE_TAC.

POP_ASSUM_LIST (Tactical)

```
POP_ASSUM_LIST : (thm list -> tactic) -> tactic
```

Synopsis

Generates a tactic from the assumptions, discards the assumptions and applies the tactic.

Description

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
```

Uses

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.

pp_tag (Tag)

```
pp_tag : ppstream -> tag -> unit
```

Synopsis
Prettyprinter for tags.

Description
An invocation `pp_tag ppstrm t` will place a representation of tag `t` on prettyprinting stream `ppstrm`.

Failure
Never fails.

Example

```
- val ppstrm = PP.mk_ppstream (Portable.defaultConsumer());
> val ppstrm = <ppstream> : ppstream

- Tag.pp_tag ppstrm (Tag.read "fooble");
> val it = () : unit

- (PP.flush_ppstream ppstrm; print "\n");
[oracles: fooble] [axioms: ]
> val it = () : unit
```

Comments
In MoscowML, `Meta.installPP` won't install `pp_tag` in the top-level loop.

See also
`Hol_pp.pp_thm`.

prefer_form_with_tok (Parse)

```
prefer_form_with_tok : {term_name : string, tok : string} -> unit
```

Synopsis
Sets a grammar rule's preferred flag, causing it to be preferentially printed.

Description
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
The initially preferred rule for conditional expressions causes them to print using the `if-then-else` syntax. If the user prefers the "traditional" syntax with `=>-|`, this change can be brought about as follows:

```
- prefer_form_with_tok {term_name = "COND", tok = "=>"};
> val it = () : unit

- Term‘if p then q else r‘;
<<HOL message: inventing new type variable names: 'a.>>
> val it = ‘p => q | r‘ : 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.

See also
`Parse.clear_prefs_for_term`.

prefer_int (intLib)

```
intLib.prefer_int : unit -> unit
```

Synopsis
Makes the parser favour integer possibilities in ambiguous terms.

Description
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, numLib.deprecate_num, Parse.overload_on, numLib.prefer_num.

prim_mk_const (Term)

```
prim_mk_const : {Thy:string, Name:string} -> term
```

Synopsis
Build a constant.

Description
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 = ‘:’a -> ’a -> 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)

```
prim_variant : term list -> term -> term
```

Synopsis
Rename a variable to be different from any in a list.

Description
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)

```
prime : string -> string
```

Synopsis
Attach a prime mark to a string.

Description
A call `prime s` is equal to `s ^ "'"`.

Failure
Never fails.

See also
`Term.variant`.

priming (Globals)

```
priming : string option ref
```

Synopsis
Controls how variables get renamed.

Description
The flag `Globals.priming` controls how certain system function perform renaming of variables. When `priming` has the value `NONE`, renaming is achieved by concatenation of primes (`'`). When `priming` has the value `SOME s`, renaming is achieved by incrementing a counter.

The default value of `priming` is `NONE`.

Example

```
- mk_primed_var ("T",bool);
> val it = ‘T’‘ : term

- with_flag (priming,SOME "_") mk_primed_var ("T",bool);
> val it = ‘T_1‘ : term
```

Comments
Proofs should be re-run in the same priming regime as they were originally performed in, since different styles of renaming can break proofs.

See also
`Term.variant`, `Term.subst`, `Term.inst`, `Term.mk_primed_var`, `Lib.with_flag`.

print_term (Parse)

```
Parse.print_term : term -> unit
```

Synopsis
Prints a term to the screen (standard out).

Description

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_theory (DB)

```
print_theory : string -> unit
```

Synopsis

Print a theory on the standard output.

Description

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:
    bool

Term constants:
    C    :('a -> 'b -> 'c) -> 'b -> 'a -> 'c
    I    :'a -> 'a
    K    :'a -> 'b -> 'a
    S    :('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
    W    :('a -> 'a -> 'b) -> 'a -> 'b
    o    :('c -> 'b) -> ('a -> 'c) -> 'a -> 'b

Definitions:
    K_DEF  |- K = (\x y. x)
    S_DEF  |- S = (\f g x. f x (g x))
    I_DEF  |- I = S K K
    C_DEF  |- combin$C = (\f x y. f y x)
    W_DEF  |- W = (\f x. f x x)
    o_DEF  |- !f g. f o g = (\x. f (g x))

Theorems:
    o_THM  |- !f g x. (f o g) x = f (g x)
    o_ASSOC  |- !f g h. f o g o h = (f o g) o h
    K_THM  |- !x y. K x y = x
    S_THM  |- !f g x. S f g x = f x (g x)
    C_THM  |- !f x y. combin$C f x y = f y x
    W_THM  |- !f x. W f x = f x x
    I_THM  |- !x. I x = x
    I_o_ID  |- !f. (I o f = f) /\ (f o I = f)
> val it = () : unit
```

See also

DB.dest_theory, DB.thy.

PROVE (BasicProvers)

```
PROVE : thm list -> term -> thm
```

Synopsis
Prove a theorem with use of supplied lemmas.

Description
`bossLib.PROVE` is identical to `BasicProvers.PROVE`.

See also
`bossLib.PROVE`.

PROVE (bossLib)

```
PROVE : thm list -> term -> thm
```

Synopsis
Prove a theorem with use of supplied lemmas.

Description
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 (hol88Lib)

```
PROVE : term * tactic -> thm
```

Synopsis
Attempts to prove a boolean term using the supplied tactic.

Description
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.

Failure
Fails if the term is not of type bool (and so cannot possibly be the conclusion of a theorem), or if the tactic cannot solve the goal. Also fails if the hol88 library has not been loaded.

See also
Tactical.TAC_PROOF, Tactical.prove, hol88Lib.prove_thm, Tactical.VALID.

prove (Tactical)

```
prove : term * tactic -> thm
```

Synopsis
Attempts to prove a boolean term using the supplied tactic.

Description
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.

Failure

Fails if the term is not of type `bool` (and so cannot possibly be the conclusion of a theorem), or if the tactic cannot solve the goal.

Comments

The function `PROVE` provides almost identical functionality, and will also list unsolved goals if the tactic fails. It is therefore preferable for most purposes.

See also

`BasicProvers.PROVE`, `hol88Lib.prove_thm`, `Tactical.TAC_PROOF`, `VALID`.

prove_abs_fn_one_one (Drule)

```
prove_abs_fn_one_one : thm -> thm
```

Synopsis

Proves that a type abstraction function is one-to-one (injective).

Description

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`, `Prim_rec.prove_abs_fn_onto`, `Drule.prove_rep_fn_one_one`, `Drule.prove_rep_fn_onto`.

prove_abs_fn_one_one (Prim_rec)

```
prove_abs_fn_one_one : thm -> thm
```

Synopsis
Proves that a type abstraction function is one-to-one (injective).

Description
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`, `Prim_rec.prove_abs_fn_onto`, `Drule.prove_rep_fn_one_one`, `Drule.prove_rep_fn_onto`.

prove_abs_fn_onto (Drule)

```
prove_abs_fn_onto : thm -> thm
```

Synopsis
Proves that a type abstraction function is onto (surjective).

Description

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`, `Prim_rec.prove_abs_fn_one_one`, `Drule.prove_rep_fn_one_one`, `Drule.prove_rep_fn_onto`.

prove_abs_fn_onto (Prim_rec)

```
prove_abs_fn_onto : thm -> thm
```

Synopsis

Proves that a type abstraction function is onto (surjective).

Description

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, Prim_rec.prove_abs_fn_one_one, Drule.prove_rep_fn_one_one, Drule.prove_rep_fn_onto.

prove_cases_thm (Prim_rec)

```
prove_cases_thm : (thm -> thm)
```

Synopsis

Proves a structural cases theorem for an automatically-defined concrete type.

Description

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

Datatype.define_type, 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)

```
prove_constructors_distinct : (thm -> thm)
```

Synopsis

Proves that the constructors of an automatically-defined concrete type yield distinct values.

Description

`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

`Datatype.define_type`, `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)

```
prove_constructors_one_one : (thm -> thm)
```

Synopsis

Proves that the constructors of an automatically-defined concrete type are injective.

Description

`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

`Datatype.define_type`, `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_HYP (Drule)

```
PROVE_HYP : thm -> thm -> thm
```

Synopsis
Eliminates a provable assumption from a theorem.

Description
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)

```
prove_induction_thm : (thm -> thm)
```

Synopsis
Derives structural induction for an automatically-defined concrete type.

Description
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

Datatype.define_type, 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)

```
prove_rec_fn_exists : thm -> term -> thm
```

Synopsis

Proves the existence of a primitive recursive function over a concrete recursive type.

Description

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 theorem of the form returned by define_type, 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
  (Term `(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

`Datatype.define_type`, `Prim_rec.new_recursive_definition`.

prove_rep_fn_one_one (Drule)

```
prove_rep_fn_one_one : thm -> thm
```

Synopsis

Proves that a type representation function is one-to-one (injective).

Description

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`, `Prim_rec.prove_abs_fn_one_one`, `Prim_rec.prove_abs_fn_onto`, `Drule.prove_rep_fn_onto`.

prove_rep_fn_onto (Drule)

```
prove_rep_fn_onto : thm -> thm
```

Synopsis

Proves that a type representation function is onto (surjective).

Description

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, Prim_rec.prove_abs_fn_one_one, Prim_rec.prove_abs_fn_onto, Drule.prove_rep_fn_one_one.

PROVE_TAC (BasicProvers)

```
PROVE_TAC : thm list -> tactic
```

Synopsis

Solve a goal with use of hypotheses and supplied lemmas.

Description

bossLib.PROVE_TAC is identical to BasicProvers.PROVE_TAC.

See also

bossLib.PROVE_TAC.

PROVE_TAC (bossLib)

```
PROVE_TAC : thm list -> tactic
```

Synopsis

Solve a goal with use of hypotheses and supplied lemmas.

Description

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.

prove_thm (hol88Lib)

```
Compat.prove_thm : (string * term * tactic) -> thm
```

Synopsis
Attempts to prove a boolean term using the supplied tactic, then save the theorem.

Description
Found in the hol88 library. When applied to a triple (s,tm,tac), giving the name to save the theorem under, the term to prove (with no assumptions) and the tactic to perform the proof, the function prove_thm attempts to prove the goal ?- tm, that is, the term tm with no assumptions, using the tactic tac. If prove_thm succeeds, it attempts to save the resulting theorem in the current theory segment, and if this succeeds, the saved theorem is returned.

Failure
Fails if the term is not of type bool (and so cannot possibly be the conclusion of a theorem), or if the tactic cannot solve the goal. In addition, prove_thm will fail if the theorem cannot be saved, e.g. because there is already a theorem of that name in the current theory segment, or if the resulting theorem has assumptions; clearly this can only happen if the tactic was invalid, so this gives some measure of validity checking. The function is not available unless the hol88 library has been loaded.

Comments
In hol90, use store_thm instead; the cognitive dissonance between prove, PROVE, and prove_thm proved to be too much for the author, so in hol90 PROVE doesn't exist: there is only prove; and prove_thm doesn't exist: it has been replaced by store_thm.

See also
Tactical.prove, BasicProvers.PROVE, Tactical.TAC_PROOF, VALID.

PSELECT_CONV (PairRules)

```
PSELECT_CONV : conv
```

Synopsis
Eliminates a paired epsilon term by introducing a existential quantifier.

Description
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)

```
PSELECT_ELIM : thm -> term * thm -> thm
```

Synopsis
Eliminates a paired epsilon term, using deduction from a particular instance.

Description
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, SELECT_AX, PairRules.PSELECT_CONV, PairRules.PSELECT_INTRO, PairRules.PSELECT_RULE.

PSELECT_EQ (PairRules)

```
PSELECT_EQ : (term -> thm -> thm)
```

Synopsis

Applies epsilon abstraction to both terms of an equation.

Description

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)

```
PSELECT_INTRO : (thm -> thm)
```

Synopsis
Introduces an epsilon term.

Description
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, SELECT_AX, PairRules.PSELECT_CONV, PairRules.PSELECT_ELIM, PairRules.PSELECT_RULE.

PSELECT_RULE (PairRules)

```
PSELECT_RULE : (thm -> thm)
```

Synopsis
Introduces a paired epsilon term in place of a paired existential quantifier.

Description

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, SELECT_AX, PairRules.PSELECT_CONV, PairRules.PEXISTS_CONV, PairRules.PSELECT_ELIM, PairRules.PSELECT_INTRO.

PSKOLEM_CONV (PairRules)

```
PSKOLEM_CONV : conv
```

Synopsis

Proves the existence of a pair of Skolem functions.

Description

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)

```
PSPEC : (term -> thm -> thm)
```

Synopsis

Specializes the conclusion of a theorem.

Description

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, PSPEC_VAR, PairRules.PGEN, PairRules.PGENL.

PSPEC_ALL (PairRules)

```
PSPEC_ALL : (thm -> thm)
```

Synopsis

Specializes the conclusion of a theorem with its own quantified pairs.

Description

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, PGEN_ALL, PairRules.PGEN_TAC, PairRules.PSPEC, PairRules.PSPECL, PairRules.PSPEC_TAC.

PSPEC_PAIR (PairRules)

```
PSPEC_PAIR : thm -> term * thm
```

Synopsis
Specializes the conclusion of a theorem, returning the chosen variant.

Description
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)

```
PSPEC_TAC : term * term -> tactic
```

Synopsis
Generalizes a goal.

Description

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

- e (PSPEC_TAC (Term`(1,2)`, Term`(x:num,y:num)`));
OK..
1 subgoal:
> val it =
    !(x,y). x + y = y + x

     : goalstack
```

Uses

Removing unnecessary speciality in a goal, particularly as a prelude to an inductive proof.

See also

`PairRules.PGEN`, `PairRules.PGENL`, `PGEN_ALL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPECL`, `PairRules.PSPEC_ALL`, `PairRules.PSTRIP_TAC`.

PSPECL (PairRules)

```
PSPECL : (term list -> thm -> thm)
```

Synopsis

Specializes zero or more pairs in the conclusion of a theorem.

Description

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`, `PGEN_ALL`, `PairRules.PGEN_TAC`, `PairRules.PSPEC`, `PairRules.PSPEC_ALL`, `PairRules.PSPEC_TAC`.

PSTRIP_ASSUME_TAC (PairRules)

```
PSTRIP_ASSUME_TAC : thm_tactic
```

Synopsis

Splits a theorem into a list of theorems and then adds them to the assumptions.

Description

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.

Uses

`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)

```
PSTRIP_GOAL_THEN : (thm_tactic -> tactic)
```

Synopsis

Splits a goal by eliminating one outermost connective, applying the given theorem-tactic to the antecedents of implications.

Description

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.

Uses

`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)

PSTRIP_TAC : tactic

Synopsis

Splits a goal by eliminating one outermost connective.

Description

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.

Uses

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)

```
PSTRIP_THM_THEN : thm_tactical
```

Synopsis

PSTRIP_THM_THEN applies the given theorem-tactic using the result of stripping off one outer connective from the given theorem.

Description

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 dis-

juncts, 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`.

Uses

`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)

```
PSTRUCT_CASES_TAC : thm_tactic
```

Synopsis
Performs very general structural case analysis.

Description
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.

Uses
Generating a case split from the axioms specifying a structure.

See also
`Tactic.STRUCT_CASES_TAC`.

PSUB_CONV (PairRules)

```
PSUB_CONV : (conv -> conv)
```

Synopsis
Applies a conversion to the top-level subterms of a term.

Description
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

```
Psyntax : Psyntax_sig
```

Synopsis
A structure that provides a tuple-style environment for term manipulation.

Description

A lot of the familiar term construction and decomposition functions from hol88 have different types in hol90. For those longing for the good old days, Psyntax provides hol88-style types. The functions provided by Psyntax return exactly the same results as their hol90 counterparts.

Each function in the Psyntax structure has a corresponding function 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;
open Psyntax
  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_pair = fn : term * term -> term
  val mk_let = fn : term * term -> term
  val mk_cons = fn : term * term -> term
  val mk_list = fn : term list * hol_type -> term
  val mk_pabs = fn : term * term -> term
  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_pair = fn : term -> term * term
  val dest_let = fn : term -> term * term
  val dest_cons = fn : term -> term * term
  val dest_list = fn : term -> term list * term
  val dest_pabs = fn : term -> term * term
  val mk_type = fn : string * hol_type list -> hol_type
  val dest_type = fn : hol_type -> string * hol_type list
  val subst = fn : (term * term) list -> term -> term
  val subst_occs = fn : int list list -> (term * term) list -> term -> term
  val inst = fn : term list -> (hol_type * hol_type) list -> term -> term
  val INST = fn : (term * term) list -> thm -> thm
  val match_type = fn : hol_type -> hol_type -> (hol_type * hol_type) list
  val match_term = fn
    : term -> term -> (term * term) list * (hol_type * hol_type) list
  val SUBST = fn : (thm * term) list -> term -> thm -> thm
  val SUBST_CONV = fn : (thm * term) list -> term -> term -> thm
  val INST_TYPE = fn : (hol_type * hol_type) list -> thm -> thm
  val INST_TY_TERM = fn
    : (term * term) list * (hol_type * hol_type) list -> thm -> thm
  val new_type = fn : int -> string -> unit
```

PURE_ASM_REWRITE_RULE (Rewrite)

```
PURE_ASM_REWRITE_RULE : (thm list -> thm -> thm)
```

Synopsis
Rewrites a theorem including the theorem's assumptions as rewrites.

Description
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 `basic_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)

```
PURE_ASM_REWRITE_TAC : (thm list -> tactic)
```

Synopsis
Rewrites a goal including the goal's assumptions as rewrites.

Description
`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.

Uses
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_ONCE_ASM_REWRITE_RULE (Rewrite)

```
PURE_ONCE_ASM_REWRITE_RULE : (thm list -> thm -> thm)
```

Synopsis
Rewrites a theorem once, including the theorem's assumptions as rewrites.

Description
PURE_ONCE_ASM_REWRITE_RULE excludes the basic tautologies in basic_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)

```
PURE_ONCE_ASM_REWRITE_TAC : (thm list -> tactic)
```

Synopsis
Rewrites a goal once, including the goal's assumptions as rewrites.

Description
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.

Uses
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)

```
PURE_ONCE_REWRITE_CONV : (thm list -> conv)
```

Synopsis
Rewrites a term once with only the given list of rewrites.

Description
PURE_ONCE_REWRITE_CONV generates rewrites from the list of theorems supplied by the user, without including the tautologies given in basic_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)

```
PURE_ONCE_REWRITE_RULE : (thm list -> thm -> thm)
```

Synopsis
Rewrites a theorem once with only the given list of rewrites.

Description
PURE_ONCE_REWRITE_RULE generates rewrites from the list of theorems supplied by the user, without including the tautologies given in basic_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)

```
PURE_ONCE_REWRITE_TAC : (thm list -> tactic)
```

Synopsis
Rewrites a goal using a supplied list of theorems, making one rewriting pass over the goal.

Description
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.

Uses
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)

```
PURE_REWRITE_CONV : (thm list -> conv)
```

Synopsis
Rewrites a term with only the given list of rewrites.

Description
This conversion provides a method for rewriting a term with the theorems given, and excluding simplification with tautologies in basic_rewrites. Matching subterms are found recursively, until no more matches are found. For more details on rewriting see GEN_REWRITE_CONV.

Uses
PURE_REWRITE_CONV is useful when the simplifications that arise by rewriting a theorem with basic_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)

```
PURE_REWRITE_RULE : (thm list -> thm -> thm)
```

Synopsis
Rewrites a theorem with only the given list of rewrites.

Description
This rule provides a method for rewriting a theorem with the theorems given, and excluding simplification with tautologies in `basic_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`.

Uses
`PURE_REWRITE_RULE` is useful when the simplifications that arise by rewriting a theorem with `basic_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)

```
PURE_REWRITE_TAC : (thm list -> tactic)
```

Synopsis
Rewrites a goal with only the given list of rewrites.

Description
`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.

Uses

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

Synopsis

A simpset containing only the conditional rewrite generator and no additional rewrites.

Description

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
```

Uses

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 ssdata value. Its usefulness is questionable.

See also

boolSimps.bool_ss, Rewrite.PURE_REWRITE_TAC, simpLib.SIMP_CONV, simpLib.SIMP_TAC.

pvariant (pairSyntax)

```
pvariant : (term list -> term -> term)
```

Synopsis

Modifies variable and constant names in a paired structure to avoid clashes.

Description

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
```

Uses

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)

```
Q_TAC : (term -> tactic) -> term quotation -> tactic
```

Synopsis
A tactical that parses in the context of a goal, a la the Q library.

Description
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`.

QUANT_CONV (Conv)

```
QUANT_CONV : conv -> conv
```

Synopsis
Applies a conversion underneath a quantifier.

Description
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.

quote (Lib)

```
quote : string -> string
```

Synopsis

Put quotation marks around a string.

Description

An application `quote s` is equal to `"\"" ^ s ^ "\""`. This is often useful when printing messages.

Failure

Never fails

Example

```
- print "foo\n";
foo
> val it = () : unit

- print (quote "foo" ^ "\n");
"foo"
> val it = () : unit
```

See also

Lib.mlquote.

r (goalstackLib)

```
r : int -> unit
```

Synopsis
Reorders the subgoals on top of the subgoal package goal stack.

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

Failure
As for `rotate`.

Uses
Proving subgoals in a different order to that generated by the subgoal package.

See also
`goalstackLib.b`, `goalstackLib.backup`, `backup_limit`, `goalstackLib.e`, `goalstackLib.expand`, `goalstackLib.expandf`, `goalstackLib.g`, `get_state`, `goalstackLib.p`, `print_state`, `rotate`, `save_top_thm`, `goalstackLib.set_goal`, `set_state`, `goalstackLib.top_goal`, `goalstackLib.top_thm`.

Raise (Feedback)

```
Raise : exn -> 'a
```

Synopsis
Print an exception before re-raising it

Description
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 raised at Foo.bar:
incomprehensible input
! Uncaught exception:
! HOL_ERR
```

See also

`Feedback`, `Feedback.ERR_to_string`, `Feedback.ERR_outstream`, `Lib.try`, `Lib.trye`.

rand (Term)

```
rand : term -> term
```

Synopsis

Returns the operand from a combination (function application).

Description

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)

```
RAND_CONV : (conv -> conv)
```

Synopsis

Applies a conversion to the operand of an application.

Description

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`.

rator (Term)

```
rator : term -> term
```

Synopsis

Returns the operator from a combination (function application).

Description

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)

```
RATOR_CONV : (conv -> conv)
```

Synopsis
Applies a conversion to the operator of an application.

Description
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 = |- (\x y. x + y)1 2 = (\y. 1 + y) 2 : thm
```

See also
`Conv.ABS_CONV`, `Conv.RAND_CONV`, `Conv.SUB_CONV`.

raw_match (Term)

```
raw_match : hol_type list -> term set
    -> term -> term
    -> (term,term) subst *
       ((hol_type,hol_type) subst * hol_type list)
    -> (term,term) subst *
       ((hol_type,hol_type) subst * hol_type list)
```

Synopsis
Primitive term matcher.

Description

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 (S,T) such that

```
aconv (subst S' (inst T pat)) ob.
```

where S' is S instantiated by T. 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. (S,T) must agree with (tmS,tyS). This means that if there is a {redex,residue} in S 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 T 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 (S,T) includes (tmS,tyS).

Failure

raw_match 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 raw_match will also fail.

Example

We first perform a match that requires type instantitations, and also alpha-convertibility.

```
- val (S,T) = raw_match [] empty_varset
                 (Term `\x:'a. x = f (y:'b)`)
                 (Term `\a.    a = ~p`) ([],([],[]));
> val S =
    [{redex = `(f :'b -> 'a)`, residue = `$~`},
     {redex = `(y :'b)`,       residue = `(p :bool)`}] : ...

  val T =
    ([{redex = `:'b`, residue = `:bool`},
      {redex = `:'a`, residue = `:bool`}], []) : ...
```

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 (S,T) to \x:'a. x = f (y:'b), type instantiation with T would be applied first, yielding \x:bool. x = f (y:bool). Then substitution with S 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 S by T to get S' before applying (S',T) to the pattern term. This can be achieved by using norm_subst. However, raw_match exposes this level of detail to the programmer.

The returned type substitution T has two components (T1,T2). T1 is a substitution, and T2 is a list of type variables, encountered in the matching process, which have matched to themselves. These identity matches are held in the separate list T2 for obscure reasons. Once matching is finished, they can be ignored (which is why they are held on a separate list).

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)

```
raw_match_type
  : hol_type list
       -> hol_type -> hol_type
       -> (hol_type,hol_type) subst * hol_type list
       -> (hol_type,hol_type) subst * hol_type list
```

Synopsis

Primitive type matching algorithm.

Description

An invocation raw_match_type away pat ty (S,Id) performs matching, just as match_tyl, 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 away pat ty (S,Id) will fail when match_typel away 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  (alpha --> bool) ([],[]);
> val it = ([{redex = ':'a', residue = ':'a -> bool'}], []) : ...

- raw_match_type [] (alpha --> beta --> gamma)
                    ((alpha --> bool) --> beta --> ind) res1;
> val it =([{redex = ':'c', residue = ':ind'},
            {redex = ':'a', residue = ':'a -> bool'}], [':'b']) : ....
```

Comments

Probably exposes too much internal state of the matching algorithm.

See also

`Type.match_type`, `Type.match_typel`.

read (Tag)

```
read : string -> tag
```

Synopsis

Make a tag suitable for use by `mk_oracle_thm`.

Description

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 = Kerneltypes.TAG(["Shamboozled"], []) : tag
```

See also

`Thm.mk_oracle_thm`, `Thm.tag`.

recInduct (bossLib)

```
recInduct : thm -> tactic
```

Synopsis
Performs recursion induction.

Description
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 bet-

ter. `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`, `goalstackLib.g`, `goalstackLib.e`.

recInduct (SingleStep)

```
recInduct : thm -> tactic
```

Synopsis
Induct with supplied recursion induction scheme.

Description
`bossLib.recInduct` is identical to `SingleStep.recInduct`.

See also
`bossLib.recInduct`.

REDEPTH_CONV (Conv)

```
REDEPTH_CONV : (conv -> conv)
```

Synopsis
Applies a conversion bottom-up to all subterms, retraversing changed ones.

Description
`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.

REFINE_EXISTS_TAC (Q)

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

Synopsis
Attacks existential goals, making the existential variable more concrete.

Description
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)
```

Uses
`Q.REFINE_EXISTS_TAC` is useful if it is clear that a 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
`Tactic.EXISTS_TAC`.

REFL (Thm)

```
REFL : conv
```

Synopsis
Returns theorem expressing reflexivity of equality.

Description
`REFL` maps any term `t` to the corresponding theorem `|- t = t`.

Failure
Never fails.

See also
`Conv.ALL_CONV`, `Tactic.REFL_TAC`.

REFL_TAC (Tactic)

```
REFL_TAC : tactic
```

Synopsis
Solves a goal which is an equation between alpha-equivalent terms.

Description
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
```

Synopsis
Registers a trace variable for a boolean reference.

Description
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_ftrace (Feedback)

```
register_ftrace :
  (string * ((unit -> int) * (int -> unit)) * int) -> unit
```

Synopsis
Registers a trace that is accessed by a set/get pair of functions.

Description
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)

```
register_trace : (string * int ref * int) -> unit
```

Synopsis
Registers a new tracing variable.

Description
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`.

remove_ovl_mapping (Parse)

```
remove_ovl_mapping: string -> {Name:string,Thy:string} -> unit
```

Synopsis
Removes an overloading mapping between the string and constant specified.

Description

Each grammar maintains two maps internally. One is from strings to non-empty sets of `Thy-Name` pairs, and the other is from `Thy-Name` pairs to strings. (Each `Thy-Name` pair serves to specify a constant without needing to worry about different type instantiations of that constant.) The first map is used to resolve overloading when parsing. A string will eventually be turned into one of the constants in the set that it maps to. When printing a constant, the map in the opposite direction is used to turn a constant into a string.

A call to `remove_ovl_mapping s {Name,Thy}` removes the given pair from both maps.

Failure

Never fails. If the given pair is not in either map, the function silently does nothing.

Uses

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
```

Synopsis

Removes parsing/pretty-printing rules from the global grammar.

Description

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_termtok (Parse)

```
remove_termtok : {term_name : string, tok : string} -> unit
```

Synopsis

Removes a rule from the global grammar.

Description

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 second 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
```

Uses
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_user_printer (Parse)

```
remove_user_printer :
    {Thy:string, Tyop:string} -> term_pp_types.userprinter option
```

Synopsis
Removes a user-defined pretty-printing function for a given type.

Description
This removes a user-defined pretty-printing function for a given type (or family of types, generated by a type operator). If there is such a printer in the global grammar for the specified type, this is returned in the option type. If there is no printer, then `NONE` is returned.

Failure
Never fails.

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`.

rename_bvar (Term)

```
rename_bvar : string -> term -> term
```

Synopsis
Performs one step of alpha conversion.

Description

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 = `\x. x ==> w` : term

- rename_bvar "x" (Term `\y. y /\ x`);
> val it = `\x'. x' /\ x` : term
```

Comments

`rename_bvar` takes constant time in the current implementation.

See also

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

RENAME_VARS_CONV (Conv)

```
Conv.RENAME_VARS_CONV : string list -> term -> thm
```

Synopsis

Renames variables underneath a binder.

Description

`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 = |- (\x y. x /\ y) = (\a 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
```

Uses

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`, `SWAP_VARS_CONV`.

repeat (Lib)

```
repeat : ('a -> 'a) -> 'a -> 'a
```

Synopsis

Iteratively apply a function until it fails.

Description

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

val maxint = repeat incr 0;   (* takes some time *)
> val maxint = 1073741823 : 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)

```
REPEAT : (tactic -> tactic)
```

Synopsis
Repeatedly applies a tactic until it fails.

Description
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)

```
REPEAT_GTCL : (thm_tactical -> thm_tactical)
```

Synopsis
Applies a theorem-tactical until it fails when applied to a goal.

Description
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_TCL (Thm_cont)

```
REPEAT_TCL : (thm_tactical -> thm_tactical)
```

Synopsis
Repeatedly applies a theorem-tactical until it fails when applied to the theorem.

Description
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)

```
REPEATC : (conv -> conv)
```

Synopsis

Repeatedly apply a conversion (zero or more times) until it fails.

Description

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+1th 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.

Failure

Never fails, but can diverge if the supplied conversion never fails.

RES_CANON (Drule)

```
RES_CANON : (thm -> thm list)
```

Synopsis

Put an implication into canonical form for resolution.

Description

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 implica-

tion 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.

Uses

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_EXISTS_CONV : conv
```

Synopsis

Converts a restricted existential quantification to a conjunction.

Description

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_EXISTS_UNIQUE_CONV : conv
```

Synopsis

Converts a restricted unique existential quantification to a conjunction.

Description

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_FORALL_AND_CONV : conv
```

Synopsis
Splits a restricted universal quantification across a conjunction.

Description
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_FORALL_CONV : conv
```

Synopsis
Converts a restricted universal quantification to an implication.

Description
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_FORALL_SWAP_CONV : conv
```

Synopsis
Changes the order of two restricted universal quantifications.

Description
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_SELECT_CONV : conv
```

Synopsis
Converts a restricted choice quantification to a conjunction.

Description
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)

```
RES_TAC : tactic
```

Synopsis

Enriches assumptions by repeatedly resolving them against each other.

Description

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 REPEATed. 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)

```
RES_THEN : (thm_tactic -> tactic)
```

Synopsis

Resolves all implicative assumptions against the rest.

Description

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)

```
reset : ('a,'b) istream -> ('a,'b) istream
```

Synopsis

Restart an istream.

Description

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))));
> val it = <istream> : (int, string) istream

- state it;
> val it = "gsym0" : string
```

Comments

Perhaps the type of `reset` should be `('a,'b) istream -> unit`.

See also

`Lib.mk_istream`, `Lib.next`, `Lib.state`.

reset_trace (Feedback)

```
reset_trace : string -> unit
```

Synopsis

Resets a tracing variable to its default value.

Description

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)

```
reset_traces : unit -> unit
```

Synopsis
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.`

RESQ_HALF_SPEC (res_quanLib)

```
RESQ_HALF_SPEC : thm -> thm
```

Synopsis
Strip a restricted universal quantification in the conclusion of a theorem.

Description
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.`

RESQ_REWR_CANON (res_quanLib)

```
RESQ_REWR_CANON : thm -> thm
```

Synopsis
Transform a theorem into a form accepted for rewriting.

Description
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.

RESQ_REWRITE1_CONV (res_quanLib)

```
RESQ_REWRITE1_CONV : thm list -> thm -> conv
```

Synopsis
Rewriting conversion using a restricted universally quantified theorem.

Description
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_CONV.

RESQ_REWRITE1_TAC (res_quanLib)

RESQ_REWRITE1_TAC : thm_tactic

Synopsis

Rewriting with a restricted universally quantified theorem.

Description

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_CONV.

RESQ_SPEC (res_quanLib)

```
RESQ_SPEC : term -> thm -> thm
```

Synopsis
Specializes the conclusion of a possibly-restricted universally quantified theorem.

Description
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.

See also
res_quanLib.RESQ_HALF_SPECL.

RESTR_EVAL_CONV (computeLib)

```
RESTR_EVAL_CONV : term list -> conv
```

Synopsis
Symbolically evaluate a term, except for specified constants.

Description

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
```

Uses

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)

```
RESTR_EVAL_RULE : term list -> thm -> thm
```

Synopsis

Symbolically evaluate a theorem, except for specified constants.

Description
This is a version of RESTR_EVAL_CONV that works on theorems.

Failure
As for RESTR_EVAL_CONV.

Uses
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)

```
RESTR_EVAL_TAC : term list -> tactic
```

Synopsis
Symbolically evaluate a theorem, except for specified constants.

Description
This is a tactic version of RESTR_EVAL_CONV.

Failure
As for RESTR_EVAL_CONV.

Uses
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)

```
rev_assoc : ''a -> ('b * ''a) list -> ('b * ''a)
```

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

Description
An invocation `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)];
> val it = (3, 2) : (int * int)
```

See also
`Lib.assoc`, `Lib.assoc1`, `Lib.assoc2`, `Lib.find`, `Lib.mem`, `Lib.tryfind`, `Lib.exists`, `Lib.all`.

rev_itlist (Lib)

```
rev_itlist : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Synopsis
Applies a binary function between adjacent elements of the reverse of a list.

Description
`rev_itlist f [x1,...,xn] b` returns `f xn ( ... (f x2 (f x1 y))...)`. 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)

```
rev_itlist2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
```

Synopsis
Applies a function to corresponding elements of 2 lists.

Description
`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)

```
reveal : string -> unit
```

Synopsis
Restores recognition of a constant by the quotation parser.

Description
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)

```
REVERSE : (tactic -> tactic)
```

Synopsis

Reverses the order of the generated subgoals.

Description

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 TO
THEN ...
```

if the first conjunct is easily dispatched with TO, and

```
REVERSE CONJ_TAC THEN1 TO
THEN ...
```

if it is the second conjunct that yields.

See also

Tactical.EVERY, Tactical.FIRST, Tactical.ORELSE, Tactical.THEN, Tactical.THEN1, Tactical.THENL.

REWR_CONV (Conv)

```
REWR_CONV : (thm -> conv)
```

Synopsis

Uses an instance of a given equation to rewrite a term.

Description

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`.

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

`Rewrite.REWRITE_CONV`.

REWRITE_CONV (Rewrite)

```
REWRITE_CONV : (thm list -> conv)
```

Synopsis

Rewrites a term including built-in tautologies in the list of rewrites.

Description

Rewriting a term using `REWRITE_CONV` utilizes as rewrites two sets of theorems: the tautologies in the ML list `basic_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 `basic_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.

Uses

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

basic_rewrites, Rewrite.GEN_REWRITE_CONV, Rewrite.ONCE_REWRITE_CONV, Rewrite.PURE_REWRITE_CONV, Conv.REWR_CONV, Drule.SUBST_CONV.

REWRITE_RULE (Rewrite)

```
REWRITE_RULE : (thm list -> thm -> thm)
```

Synopsis

Rewrites a theorem including built-in tautologies in the list of rewrites.

Description

Rewriting a theorem using REWRITE_RULE utilizes as rewrites two sets of theorems: the tautologies in the ML list basic_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 basic_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.

Uses
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, basic_rewrites, Rewrite.GEN_REWRITE_RULE, Rewrite.ONCE_REWRITE_RULE, Rewrite.PURE_REWRITE_RULE, Conv.REWR_CONV, Rewrite.REWRITE_CONV, Thm.SUBST.

REWRITE_TAC (Rewrite)

```
REWRITE_TAC : (thm list -> tactic)
```

Synopsis
Rewrites a goal including built-in tautologies in the list of rewrites.

Description
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 basic_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
```

Uses

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)

```
rewrites : thm list -> ssdata
```

Synopsis

Creates an `ssdata` 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 `ssdata` values containing a great number of rewrites. This is how the basic system's various `ssdata` values are constructed where those values consist only of rewrite theorems.

See also

bossLib.++, simpLib.mk_simpset, simpLib.SIMPSET, bossLib.SIMP_CONV.

rewrites (simpLib)

```
rewrites : thm list -> ssdata
```

Synopsis

Create an `ssdata` value consisting of the given theorems as rewrites.

Description
`bossLib.rewrites` is identical to `simpLib.rewrites`.

See also
`bossLib.rewrites`.

rhs (boolSyntax)

```
rhs : term -> term
```

Synopsis
Returns the right-hand side of an equation.

Description
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`.

RIGHT_AND_EXISTS_CONV (Conv)

```
RIGHT_AND_EXISTS_CONV : conv
```

Synopsis
Moves an existential quantification of the right conjunct outwards through a conjunction.

Description
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)

```
RIGHT_AND_FORALL_CONV : conv
```

Synopsis
Moves a universal quantification of the right conjunct outwards through a conjunction.

Description
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)

```
RIGHT_AND_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the right conjunct outwards through a conjunction.

Description

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)

```
RIGHT_AND_PFORALL_CONV : conv
```

Synopsis

Moves a paired universal quantification of the right conjunct outwards through a conjunction.

Description

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)

```
RIGHT_BETA : (thm -> thm)
```

Synopsis
Beta-reduces a top-level beta-redex on the right-hand side of an equation.

Description
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)

```
RIGHT_CONV_RULE : (conv -> thm -> thm)
```

Synopsis
Applies a conversion to the right-hand side of an equational theorem.

Description
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_IMP_EXISTS_CONV (Conv)

```
RIGHT_IMP_EXISTS_CONV : conv
```

Synopsis

Moves an existential quantification of the consequent outwards through an implication.

Description

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)

```
RIGHT_IMP_FORALL_CONV : conv
```

Synopsis
Moves a universal quantification of the consequent outwards through an implication.

Description
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)

```
RIGHT_IMP_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the consequent outwards through an implication.

Description
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)

```
RIGHT_IMP_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the consequent outwards through an implication.

Description
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_PEXISTS_CONV`.

RIGHT_LIST_BETA (Drule)

```
RIGHT_LIST_BETA : (thm -> thm)
```

Synopsis
Iteratively beta-reduces a top-level beta-redex on the right-hand side of an equation.

Description

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)

```
RIGHT_LIST_PBETA : (thm -> thm)
```

Synopsis

Iteratively beta-reduces a top-level paired beta-redex on the right-hand side of an equation.

Description

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)

```
RIGHT_OR_EXISTS_CONV : conv
```

Synopsis
Moves an existential quantification of the right disjunct outwards through a disjunction.

Description
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)

```
RIGHT_OR_FORALL_CONV : conv
```

Synopsis
Moves a universal quantification of the right disjunct outwards through a disjunction.

Description
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)

```
RIGHT_OR_PEXISTS_CONV : conv
```

Synopsis
Moves a paired existential quantification of the right disjunct outwards through a disjunction.

Description
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)

```
RIGHT_OR_PFORALL_CONV : conv
```

Synopsis
Moves a paired universal quantification of the right disjunct outwards through a disjunction.

Description

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)

```
RIGHT_PBETA : (thm -> thm)
```

Synopsis

Beta-reduces a top-level paired beta-redex on the right-hand side of an equation.

Description

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`.

Rsyntax

`Rsyntax`

Synopsis

A structure that restores a record-style environment for term manipulation.

Description

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
```

RULE_ASSUM_TAC (Tactic)

```
RULE_ASSUM_TAC : ((thm -> thm) -> tactic)
```

Synopsis
Maps an inference rule over the assumptions of a goal.

Description
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
`Tactical.ASSUM_LIST`, `Tactical.MAP_EVERY`, `Tactical.MAP_FIRST`, `Tactical.POP_ASSUM_LIST`.

RW_TAC (BasicProvers)

```
RW_TAC : simpset -> thm list -> tactic
```

Synopsis
Simplification with case-splitting and built-in knowledge of declared datatypes.

Description
`bossLib.RW_TAC` is identical to `BasicProvers.RW_TAC`.

See also
bossLib.RW_TAC.

RW_TAC (bossLib)

```
RW_TAC : simpset -> thm list -> tactic
```

Synopsis
Simplification with case-splitting and built-in knowledge of declared datatypes.

Description
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, Ho_Rewrite.REWRITE_TAC, bossLib.Hol_datatype.

S (Lib)

```
S : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
```

Synopsis
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.A`, `Lib.B`, `Lib.C`, `Lib.I`, `Lib.K`, `Lib.W`.

same_const (Term)

```
same_const : term -> term -> bool
```

Synopsis
Constant time equality check for constants.

Description
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`.

save_thm (Theory)

```
save_thm : string * thm -> thm
```

Synopsis
Stores a theorem in the current theory segment.

Description
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 foo = save_thm("foo", REFL (Term `x:bool`));
> val foo = |- x = x : thm

- current_theorems();
> val it = [("foo", |- x = x)] : (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.

Uses
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)

```
say : string -> unit
```

Synopsis
Print a string.

Description
An application `say s` prints the string `s` on the standard output.

Failure
Never fails.

Comments
The ML Standard Basis Library structure `TextIO` offers related functions.

scrub (Theory)

```
scrub : unit -> unit
```

Synopsis
Remove all out-of-date elements from the current theory segment.

Description
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
```

Uses

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, Theory.delete_axiom, Theory.delete_definition, Theory.delete_theorem.

select (boolSyntax)

```
select : term
```

Synopsis

Constant denoting Hilbert's choice operator.

Description

The ML variable boolSyntax.select is bound to the term min$@.

See also

boolSyntax.equality, boolSyntax.implication, 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.

SELECT_CONV (Conv)

```
SELECT_CONV : conv
```

Synopsis

Eliminates an epsilon term by introducing an existential quantifier.

Description

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
```

Uses

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)

```
SELECT_ELIM : thm -> term * thm -> thm
```

Synopsis

Eliminates an epsilon term, using deduction from a particular instance.

Description

SELECT_ELIM expects two arguments, a theorem th1, and a pair (v,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 variable v appears only in the assumption

`P v` 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 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`. If `v` appears in the conclusion of `th2`, the epsilon term will NOT be eliminated, and the conclusion will be `t[$@ P/v]`.

Failure

Fails if the first theorem is not of the form `A1 |- P($@ P)`, 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 (Term`g:num->num`, th2);
val it = |- ?h. ONE_ONE h /\ ~ONTO h : thm
```

Uses

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`, `SELECT_AX`, `Conv.SELECT_CONV`, `Drule.SELECT_INTRO`, `Drule.SELECT_RULE`.

SELECT_EQ (Drule)

```
SELECT_EQ : (term -> thm -> thm)
```

Synopsis
Applies epsilon abstraction to both terms of an equation.

Description
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)

```
SELECT_INTRO : (thm -> thm)
```

Synopsis
Introduces an epsilon term.

Description

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, SELECT_AX, Conv.SELECT_CONV, Drule.SELECT_ELIM, Drule.SELECT_RULE.

SELECT_RULE (Drule)

```
SELECT_RULE : thm -> thm
```

Synopsis

Introduces an epsilon term in place of an existential quantifier.

Description

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
```

Uses

May be used to introduce an epsilon term to permit rewriting with a constant defined using the epsilon operator.

See also

Thm.CHOOSE, SELECT_AX, Conv.SELECT_CONV, Drule.SELECT_ELIM, Drule.SELECT_INTRO.

set_backup (goalstackLib)

```
goalstackLib.set_backup : int -> unit
```

Synopsis

Limits the number of proof states saved on the subgoal package backup list.

Description

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;
()  unit

- g `(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])

     : proofs

- e CONJ_TAC;
OK..
2 subgoals:
> val it =
    TL [1; 2; 3] = [2; 3]


    HD [1; 2; 3] = 1

     : goalstack

- e (REWRITE_TAC[listTheory.HD]);
OK..

Goal proved.
|- HD [1; 2; 3] = 1

Remaining subgoals:
> val it =
    TL [1; 2; 3] = [2; 3]

     : goalstack

- b();
> val it =
    TL [1; 2; 3] = [2; 3]


    HD [1; 2; 3] = 1

     : goalstack

- b();
! Uncaught exception:
! CANT_BACKUP_ANYMORE
```

See also

goalstackLib.b, goalstackLib.backup, goalstackLib.e, goalstackLib.expand,

goalstackLib.expandf, goalstackLib.g, goalstackLib.p, goalstackLib.r, goalstackLib.rotate, goalstackLib.set_goal, goalstackLib.top_goal, goalstackLib.top_thm.

set_diff (Lib)

```
set_diff : ''a list -> ''a list -> ''a list
```

Synopsis
Computes the set-theoretic difference of two 'sets'.

Description
`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.

High performance set operations may be found in the ML Standard Basis Library.

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)

```
set_eq : ''a list -> ''a list -> bool
```

Synopsis
Tells whether two lists have the same elements.

Description
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
High performance finite set operations may be found in the ML Standard Basis Library.

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.op_set_eq`, `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
```

Synopsis
Allows the fixity of tokens to be updated.

Description
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 them appropriate fixities:

```
- 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.

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`.

set_goal (goalstackLib)

```
set_goal : term list * term -> unit
```

Synopsis

Initializes the subgoal package with a new goal.

Description

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: 1 proof.
    1. Incomplete:
         Initial goal:
         (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])

     : proofs
```

Uses

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. The subgoals of the current level may be considered in any order.

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

`goalstackLib.b`, `goalstackLib.backup`, `goalstackLib.set_backup`, `goalstackLib.e`, `goalstackLib.expand`, `goalstackLib.expandf`, `goalstackLib.g`, `goalstackLib.p`, `goalstackLib.r`, `goalstackLib.rotate`, `goalstackLib.top_goal`, `goalstackLib.top_thm`.

set_base_rewrites (Rewrite)

```
set_base_rewrites: rewrites -> unit
```

Synopsis

Allows the user to control the built-in database of simplifications used in rewriting.

Description

Uses

See also

`base_rewrites`, `add_base_rewrites`, `Rewrite.empty_rewrites`, `Rewrite.add_rewrites`.

set_known_constants (Parse)

```
Parse.set_known_constants : string list -> unit
```

Synopsis
Specifies the list of names that should be parsed as constants.

Description
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 =
    ["bool_case", "ARB", "TYPE_DEFINITION", "ONTO", "ONE_ONE", "COND",
     "LET", "?!", "~", "F", "\\/", "/\\", "!", "T", "?", "@",
     "==>", "="]
    : string list
- Term`p /\ q`;
> val it = `p /\ q` : term
- set_known_constants (Lib.subtract (known_constants()) ["/\\"]);
> val it = () : unit
- Term`p /\ q`;
<<HOL message: inventing new type variable names: 'a, 'b, 'c.>>
> val it = `p /\ q` : term
- strip_comb it;
> val it = (`$/\`, [`p`, `q`]) : term * term list
- dest_var (#1 it);
> val it = ("/\\", `:'a -> 'b -> 'c`) : string * hol_type
```

Uses
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_MLname (Theory)

```
set_MLname : string -> string -> unit
```

Synopsis

Change the name attached to an element of the current theory.

Description

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_trace (Feedback)

```
set_trace : string -> int -> unit
```

Synopsis
Set a tracing level for a registered trace.

Description
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`.

setify (hol88Lib)

```
Compat.setify : ''a list -> ''a list
```

Synopsis
`setify` makes a set out of an "eqtype" list.

Description
Found in the hol88 library. `setify l` removes repeated elements from `l`, leaving the last occurrence of each duplicate in the list.

Failure

Never fails. The function is not available unless the hol88 library has been loaded.

Example

```
- setify [1,2,3,1,4,3];
[2,1,4,3] : int list
```

Comments

Perhaps the first occurrence of each duplicate should be left in the list, not the last? However, other functions may rely on the ordering currently used. Included in `Compat` because `setify` is not found in hol90 (`mk_set` is used instead.)

See also

`distinct`.

show_numeral_types (Parse)

```
Globals.show_numeral_types : bool ref
```

Synopsis

A flag which causes numerals to be printed with suffix annotation when true.

Description

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

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

- Term‘~3‘;
> val it = ‘~3‘ : term

- show_numeral_types := true;
> val it = () : unit

- Term‘~3‘;
> val it = ‘~3i‘ : term
```

Uses

Can help to disambiguate terms involving numerals.

See also

`Parse.add_numeral_form`, `Globals.show_types`.

show_tags (Globals)

```
show_tags : bool ref
```

Synopsis

Flag for controlling display of tags in theorem prettyprinter.

Description

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: ] [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`, `Globals.show_axioms`.

show_types (Globals)

```
Globals.show_types : bool ref
```

Synopsis

Flag controlling printing of HOL types (in terms).

Description

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)

```
SIMP_CONV : simpset -> thm list -> conv
```

Synopsis
Applies a simpset and a list of rewrite rules to simplify a term.

Description
`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, and the way they can be combined to create simpsets is given in the reference entry for SIMPSET.

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 `ssdata` 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 subterms. (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`. Similarly, if one had a conversion for deciding equalities over a certain type `foo`, one would add the relevant conversion keyed on terms ``` ``x:foo = y`` ```.

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 ssdata` value. (These `ssdata` values 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 = |- (\x. x + 3) 4 = 7 : thm
```

Uses

`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.SIMPSET`, `bossLib.EVAL`.

SIMP_CONV (simpLib)

```
SIMP_CONV : simpset -> thm list -> conv
```

Synopsis
Simplify a term with the given simpset and theorems.

Description
`bossLib.SIMP_CONV` is identical to `simpLib.SIMP_CONV`.

See also
`bossLib.SIMP_CONV`.

SIMP_PROVE (simpLib)

```
simpLib.SIMP_PROVE : simpset -> thm list -> term -> thm
```

Synopsis
Like `SIMP_CONV`, but converts boolean terms to theorem with same conclusion.

Description
`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.

Uses
`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)

```
SIMP_RULE : simpset -> thm list -> thm -> thm
```

Synopsis
Simplifies the conclusion of a theorem according to the given simpset and theorem rewrites.

Description
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)

```
SIMP_RULE : simpset -> thm list -> thm -> thm
```

Synopsis
Simplify a term with the given simpset and theorems.

Description

bossLib.SIMP_RULE is identical to simpLib.SIMP_RULE.

See also

bossLib.SIMP_RULE.

SIMP_TAC (bossLib)

```
SIMP_TAC : simpset -> thm list -> tactic
```

Synopsis

Simplifies the goal, using the given simpset and the additional theorems listed.

Description

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.

Uses

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)

```
SIMP_TAC : simpset -> thm list -> tactic
```

Synopsis

Simplify a term with the given simpset and theorems.

Description

`bossLib.SIMP_TAC` is identical to `simpLib.SIMP_TAC`.

See also

`bossLib.SIMP_TAC`.

SIMPSET (simpLib)

```
SIMPSET : { 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 : (thm -> thm list) option,
            rewrs  : thm list } -> ssdata
```

Synopsis

Constructs `ssdata` values.

Description

The `ssdata` type is the way in which simplification components are packaged up and made available to the simplifier (though `ssdata` 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 `SIMPSET` as an argument has six 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 form of the associative theorem must be

```
|- x op (y op z) = (x op y) op z
```

and the commutative theorem (the second element of the pair) must be of the form

```
|- x op y = y op x
```

Note that neither theorem can have any universal quantification.

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 a 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.

The convs field is a list of conversions that the simplifier will apply. Each conversion added to an ssdata 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. The name and trace fields are only relevant to the debugging facilities of the simplifier.

The dprocs field of the record passed to SIMPSET is where decision procedures can be specified. The construction of values of type reducer will be described in other reference entries (some of which may not have been written yet).

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 rewrs field of the record is a list of rewrite theorems that are to be applied.

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 = SIMPSET {ac = [], congs = [],
                     convs = [{conv = K (K MUL_CONV),
                               key= SOME ([], Term`x * y`),
                               name = "MUL_CONV",
                               trace = 2}],
                     dprocs = [], filter = NONE, rewrs = []};
> val ssd =
    SIMPSET{ac = [], congs = [],
            convs =
              [{conv = fn, key = SOME([], `x * y`), name = "MUL_CONV",
                trace = 2}], dprocs = [], filter = NONE, rewrs = []}
    : ssdata
- 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 = SIMPSET {ac = [(SPEC_ALL ADD_ASSOC, SPEC_ALL ADD_SYM)],
                      congs = [], convs = [], dprocs = [],
                      filter = NONE, rewrs = []};
> val ssd2 =
    SIMPSET{ac = [(|- m + n + p = (m + n) + p, |- m + n = n + m)],
            congs = [], convs = [], dprocs = [], filter = NONE,
            rewrs = []}
    : ssdata
- 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
```

Comments

`SIMPSET` is not the right name for something that creates an `ssdata` value.

See also

`simpLib.++`, `boolSimps.bool_ss`, `simpLib.mk_simpset`, `simpLib.rewrites`, `simpLib.SIMP_CONV`.

SKOLEM_CONV (Conv)

```
SKOLEM_CONV : conv
```

Synopsis
Proves the existence of a Skolem function.

Description
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)

```
snd : ('a * 'b) -> 'b
```

Synopsis
Extracts the second component of a pair.

Description
`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`.

sort (Lib)

```
sort : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Synopsis

Sorts a list using a given transitive 'ordering' relation.

Description

The call `sort opr list` where `opr` is a curried transitive relation on the elements of `list`, will topologically sort the list, i.e., will permute it 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 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`.

SPEC (Thm)

```
SPEC : term -> thm -> thm
```

Synopsis

Specializes the conclusion of a theorem.

Description

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, Drule.GENL, Drule.GEN_ALL.

SPEC_ALL (Drule)

```
SPEC_ALL : thm -> thm
```

Synopsis

Specializes the conclusion of a theorem with its own quantified variables.

Description

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, Drule.GENL, Drule.GEN_ALL, Tactic.GEN_TAC, Thm.SPEC, Drule.SPECL, Tactic.SPEC_TAC.

SPEC_TAC (Tactic)

```
SPEC_TAC : term * term -> tactic
```

Synopsis

Generalizes a goal.

Description

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`.

Uses

Removing unnecessary speciality in a goal, particularly as a prelude to an inductive proof.

See also

Thm.GEN, Drule.GENL, Drule.GEN_ALL, Tactic.GEN_TAC, Thm.SPEC, Drule.SPECL, Drule.SPEC_ALL, Tactic.STRIP_TAC.

SPEC_VAR (Drule)

```
SPEC_VAR : thm -> term * thm
```

Synopsis
Specializes the conclusion of a theorem, returning the chosen variant.

Description
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`, `Drule.GENL`, `Drule.GEN_ALL`, `Tactic.GEN_TAC`, `Thm.SPEC`, `Drule.SPECL`, `Drule.SPEC_ALL`.

Specialize (Thm)

```
Specialize : term -> thm -> thm
```

Synopsis
Specializes the conclusion of a universal theorem.

Description
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, Drule.GENL, Drule.GEN_ALL.

SPECL (Drule)

```
SPECL : term list -> thm -> thm
```

Synopsis

Specializes zero or more variables in the conclusion of a theorem.

Description

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, Drule.GENL, Drule.GEN_ALL, Tactic.GEN_TAC, Thm.SPEC, Drule.SPEC_ALL,

Tactic.SPEC_TAC.

spine_pair (pairSyntax)

```
spine_pair : term -> term list
```

Synopsis
Breaks a paired structure into its constituent pieces.

Example

```
- 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.

split (Lib)

```
split : ('a * 'b) list -> ('a list * 'b list)
```

Synopsis
Converts a list of pairs into a pair of lists.

Description
`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)

```
split_after : int -> 'a list -> 'a list * 'a list
```

Synopsis
Breaks a list in two at a specified index.

Description
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];
! Uncaught exception:
! HOL_ERR

- split_after 0 ([]:int list);
> val it = ([], []) : int list * int list
```

See also
`Lib.partition`, `Lib.pluck`.

SPOSE_NOT_THEN (bossLib)

```
SPOSE_NOT_THEN : (thm -> tactic) -> tactic
```

Synopsis
Initiate proof by contradiction.

Description
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.

srw_ss (BasicProvers)

```
srw_ss : unit -> simpset
```

Synopsis
Implicit `simpset`.

Description
`bossLib.srw_ss` is identical to `BasicProvers.srw_ss`.

See also
`bossLib.srw_ss`.

srw_ss (bossLib)

```
srw_ss : unit -> simpset
```

Synopsis
Returns the "stateful rewriting" system's underlying `simpset`.

Description
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. First, 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)

```
SRW_TAC : simpset -> thm list -> tactic
```

Synopsis
A version of `RW_TAC` with an implicit `simpset`.

Description
`bossLib.SRW_TAC` is identical to `BasicProvers.SRW_TAC`.

See also
`bossLib.SRW_TAC`.

SRW_TAC (bossLib)

```
SRW_TAC : ssdata list -> thm list -> tactic
```

Synopsis
A version of `RW_TAC` with an implicit `simpset`.

Description
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.SIMPSET, simpLib.ssdata.

start_time (Lib)

```
start_time : unit -> Timer.cpu_timer
```

Synopsis

Set a timer running.

Description

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 = <cpu_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 ML Standard Basis Library structures `Timer` and `Time`.

See also

`Lib.end_time`, `Lib.time`.

state (Lib)

```
state : ('a,'b) istream -> 'b
```

Synopsis

Project the state of an istream.

Description

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 (concat "gsym" o int_to_string);
> val it = <istream> : (int, string) istream

- state istrm;
> val it = "gsym0" : string

- next (next istrm);
> val it = <istream> : (int, string) istream

- state istrm;
> val it = "gsym2" : string
```

See also

`Lib.mk_istream`, `Lib.next`, `Lib.reset`.

std_ss (bossLib)

```
std_ss : simpset
```

Synopsis

Basic simplification set.

Description

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)
```

Uses

times 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)

```
store_thm : string * term * tactic -> thm
```

Synopsis

Proves and then stores a theorem in the current theory segment.

Description

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.

Uses

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`.

string_of_int (hol88Lib)

```
Compat.string_of_int : int -> string
```

Synopsis

Maps an integer to the corresponding decimal string.

Description

Found in the hol88 library. When given an integer, `string_of_int` returns a string representing the number in standard decimal notation, with a leading minus sign if the number is negative, and no leading zeros.

Failure

Never fails. The function is not available unless the hol88 library has been loaded.

Comments

Not found in hol90, since the author always got it backwards; use `int_to_string` instead. Likewise, `int_of_string` is not found in hol90; use `string_to_int`.

See also

`ascii`, `ascii_code`, `hol88Lib.int_of_string`, `int_to_string`, `string_to_int`.

string_to_int (Lib)

```
string_to_int : string -> int
```

Synopsis

Translates from a string to an integer.

Description

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";
! Uncaught exception:
! HOL_ERR
```

Comments

Similar functionality can be obtained from the Standard Basis Library function `Int.fromString`.

See also

`Lib.int_to_string`.

strip_abs (boolSyntax)

```
strip_abs : term -> term list * term
```

Synopsis

Iteratively breaks apart abstractions.

Description

If `M` has the form `\x1 ... xn.t` then `strip_abs M` returns `([x1,...,xn],t)`. Note that

```
strip_abs(list_mk_abs([x1,...,xn],t))
```

will not return `([x1,...,xn],t)` if `t` is an abstraction.

Failure

Never fails.

See also

`boolSyntax.list_mk_abs`, `Term.dest_abs`.

strip_abs (Term)

```
strip_abs : term -> term list * term
```

Synopsis
Break apart consecutive lambda abstractions.

Description
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_ASSUME_TAC (Tactic)

```
STRIP_ASSUME_TAC : thm_tactic
```

Synopsis
Splits a theorem into a list of theorems and then adds them to the assumptions.

Description

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
```

Uses

`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)

```
strip_binder : term option -> term -> term list * term
```

Synopsis

Break apart consecutive binders.

Description

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)

```
STRIP_BINDER_CONV : term option -> conv -> conv
```

Synopsis

Applies a conversion underneath a binder prefix.

Description

If the application of conv to M yields |- M = N, then STRIP_BINDER_CONV (SOME c) conv (c(\v1. ... returns |- c(\v1. ... (c(\vn.M))...) = c(\v1. ... (c(\vn.N))...) and STRIP_BINDER_CONV NON 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 = |- (\u v w. (\a. a + v * w) u) = (\u 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)

```
strip_comb : term -> term * term list
```

Synopsis

Iteratively breaks apart combinations (function applications).

Description

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)

```
strip_conj : term -> term list
```

Synopsis

Recursively breaks apart conjunctions.

Description

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)

```
strip_disj : term -> term list
```

Synopsis
Recursively breaks apart disjunctions.

Description
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)

```
strip_exists : term -> term list * term
```

Synopsis
Iteratively breaks apart existential quantifications.

Description
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)

```
strip_forall : term -> term list * term
```

Synopsis
Iteratively breaks apart universal quantifications.

Description
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)

```
strip_fun : hol_type -> hol_type list * hol_type
```

Synopsis
Iteratively breaks apart function types.

Description
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)`);
> val it = ([`:a -> 'bool`, `:'b`], `:'c`) : hol_type list * hol_type
```

See also

`boolSyntax.list_mk_fun`, `Type.dom_rng`, `Type.dest_type`.

STRIP_GOAL_THEN (Tactic)

```
STRIP_GOAL_THEN : thm_tactic -> tactic
```

Synopsis

Splits a goal by eliminating one outermost connective, applying the given theorem-tactic to the antecedents of implications.

Description

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
```

Uses

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, Thm_cont.FILTER_STRIP_THEN, Tactic.GEN_TAC, Tactic.STRIP_ASSUME_TAC, Tactic.STRIP_TAC.

strip_imp (boolSyntax)

```
strip_imp : term -> term list * term
```

Synopsis
Iteratively breaks apart implications.

Description
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)";;
> val it = (["T ==> F"; "T"], "F") : term list * term

- 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)

```
strip_imp_only : term -> term list * term
```

Synopsis
Iteratively breaks apart implications.

Description

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)

```
strip_neg : term -> term * int
```

Synopsis

Breaks iterated negations down to an unnegated core.

Description

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`);
<<HOL message: inventing new type variable names: 'a>>
> 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)

```
strip_pabs : term -> term list * term
```

Synopsis

Iteratively breaks apart paired abstractions.

Description

`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)

```
strip_pair : term -> term list
```

Synopsis
Recursively breaks a paired structure into its constituent pieces.

Example

```
- 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)

```
strip_pexists : term -> term list * term
```

Synopsis
Iteratively breaks apart paired existential quantifications.

Description
`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.list_mk_pexists, pairSyntax.dest_pexists.

strip_pforall (pairSyntax)

```
strip_pforall : term -> term list * term
```

Synopsis
Iteratively breaks apart paired universal quantifications.

Description
`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.list_mk_pforall, pairSyntax.dest_pforall.

STRIP_QUANT_CONV (Conv)

```
STRIP_QUANT_CONV : conv -> conv
```

Synopsis
Applies a conversion underneath a quantifier prefix.

Description
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)

```
strip_res_exists : (term -> ((term # term) list # term))
```

Synopsis
Iteratively breaks apart a restricted existentially quantified term.

Description
`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)

```
strip_res_forall : term -> ((term # term) list # term)
```

Synopsis
Iteratively breaks apart a restricted universally quantified term.

Description
`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)

```
STRIP_TAC : tactic
```

Synopsis
Splits a goal by eliminating one outermost connective.

Description

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 ?- v1 \/ ... \/ vn ==> v
============================       ================================
    A u {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
```

Uses

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)

```
STRIP_THM_THEN : thm_tactical
```

Synopsis

STRIP_THM_THEN applies the given theorem-tactic using the result of stripping off one outer connective from the given theorem.

Description

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`.

Uses

`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)

```
STRUCT_CASES_TAC : thm_tactic
```

Synopsis

Performs very general structural case analysis.

Description

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.

Uses

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_CONV (Conv)

```
SUB_CONV : (conv -> conv)
```

Synopsis
Applies a conversion to the top-level subterms of a term.

Description
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. 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"` returns `|- t = t`.

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.RAND_CONV`, `Conv.RATOR_CONV`.

SUBGOAL_THEN (Tactical)

```
SUBGOAL_THEN : term -> thm_tactic -> tactic
```

Synopsis
Allows the user to introduce a lemma.

Description
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
```

Failure
`SUBGOAL_THEN` will fail if an attempt is made to use a nonboolean term as a lemma.

Uses
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)

```
SUBS : (thm list -> thm -> thm)
```

Synopsis

Makes simple term substitutions in a theorem using a given list of theorems.

Description

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
```

Uses

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)

```
SUBS_OCCS : (int list * thm) list -> thm -> thm
```

Synopsis

Makes substitutions in a theorem at specific occurrences of a term, using a list of equational theorems.

Description

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
```

Uses

`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)

```
type ('a,'b) subst
```

Synopsis

Type abbreviation for substitutions.

Description

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)

```
subst : (term,term) subst -> term -> term
```

Synopsis

Substitutes terms in a term.

Description

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";

- 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 = ‘\a’. a‘ : term

- subst [Term‘flip:’a‘ |-> Term‘foo:’a‘]
        (Term‘waddle:’a‘);
> val it = ‘waddle‘ : term
```

See also

Term.inst, Thm.SUBST, Drule.SUBS, Lib.|->.

SUBST (Thm)

```
SUBST : (term,thm) subst -> term -> thm -> thm
```

Synopsis

Makes a set of parallel substitutions in a theorem.

Description

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 `A1 u ... An |- t[u1,...,un]`. 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.

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;

(*    x = ‘x‘
      y = ‘y‘
    th0 = |- SUC 0 = 0 + 1
    th1 = |- SUC 1 = 1 + 1     *)

- 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.

Uses

For substituting at selected occurrences. Often useful for writing special purpose derived inference rules.

See also

Drule.SUBS, Drule.SUBST_CONV, Lib.|->.

SUBST1_TAC (Tactic)

```
SUBST1_TAC : thm_tactic
```

Synopsis
Makes a simple term substitution in a goal using a single equational theorem.

Description
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
```

Uses
`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)

SUBST_ALL_TAC : thm_tactic

Synopsis

Substitutes using a single equation in both the assumptions and conclusion of a goal.

Description

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)

```
subst_assoc : ('a -> bool) -> ('a,'b)subst -> 'b option
```

Synopsis

Treats a substitution as an association list.

Description

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 `redex`es 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)

```
SUBST_CONV : {redex :term, residue :thm} list -> term -> conv
```

Synopsis

Makes substitutions in a term at selected occurrences of subterms, using a list of theorems.

Description

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, re` 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 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}]
             (Term‘(x + y) > SUC 1‘)
             (Term‘(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]
             (Term‘(x + y) > SUC 1‘)
             (Term‘(SUC 0 + SUC 1) > SUC 1‘);
```

Uses

`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)

```
SUBST_MATCH : (thm -> thm -> thm)
```

Synopsis

Substitutes in one theorem using another, equational, theorem.

Description

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
```

Uses

`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)

```
subst_occs : int list list -> term subst -> term -> term
```

Synopsis
Substitutes for particular occurrences of subterms of a given term.

Description
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)

```
SUBST_OCCS_TAC : (int list * thm) list -> tactic
```

Synopsis

Makes substitutions in a goal at specific occurrences of a term, using a list of theorems.

Description

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)
```

Uses

`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)

```
SUBST_TAC : (thm list -> tactic)
```

Synopsis

Makes term substitutions in a goal using a list of theorems.

Description

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
```

Uses

`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)

```
subtract : ''a list -> ''a list -> ''a list
```

Synopsis

Computes the set-theoretic difference of two 'sets'.

Description

Behaves exactly like `set_diff`.

See also

`Lib.set_diff`.

SWAP_EXISTS_CONV (Conv)

```
SWAP_EXISTS_CONV : conv
```

Synopsis
Interchanges the order of two existentially quantified variables.

Description
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)

```
SWAP_PEXISTS_CONV : conv
```

Synopsis
Interchanges the order of two existentially quantified pairs.

Description
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)

```
SWAP_PFORALL_CONV : conv
```

Synopsis
Interchanges the order of two universally quantified pairs.

Description
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)

```
SYM : thm -> thm
```

Synopsis
Swaps left-hand and right-hand sides of an equation.

Description
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`.

SYM_CONV (Conv)

```
SYM_CONV : conv
```

Synopsis
Interchanges the left and right-hand sides of an equation.

Description
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`.

T (boolSyntax)

```
T : term
```

Synopsis
Constant denoting truth.

Description
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)

```
TAC_PROOF : goal * tactic -> thm
```

Synopsis
Attempts to prove a goal using a given tactic.

Description
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`, `hol88Lib.prove_thm`, `VALID`.

tag (Tag)

```
type tag
```

Synopsis
Abstract type of oracle tags.

Description
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)

```
tag : thm -> tag
```

Synopsis
Extract the tag from a theorem.

Description
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 = Kerneltypes.TAG(["MK_THM"], []) : tag

- Thm.tag NOT_FORALL_THM;
> val it = Kerneltypes.TAG([], []) : tag
```

See also
`Thm.mk_oracle_thm`, `Tag.read`, `Tag.merge`, `Tag.pp_tag`.

Term (Parse)

```
Parse.Term : term quotation -> term
```

Synopsis
Parses a quotation into a term value

Description
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 `TruePrefix` 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.

Uses
Turns strings into terms.

See also
Parse.Absyn, Preterm, Type, Parse.overload_on, guessing_overloads, guessing_tyvars, Parse.term_grammar.

term (Term)

```
eqtype term
```

Synopsis
ML datatype of HOL terms.

Description
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
```

Synopsis
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.Term`.

term_to_string (Parse)

```
Parse.term_to_string : term -> string
```

Synopsis
Converts a term to a string.

Description
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`.

tgoal (Defn)

```
tgoal : defn -> proofs
```

Synopsis
Set up a termination proof

Description
`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)))`;

- 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)
```

See also
`TotalDefn.WF_REL_TAC`, `Defn.tprove`, `Defn.Hol_defn`.

THEN (Tactical)

```
op THEN : tactic * tactic -> tactic
```

Synopsis
Applies two tactics in sequence.

Description
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.

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.

THEN1 (Tactical)

```
op THEN1 : tactic * tactic -> tactic
```

Synopsis
A tactical like THEN that applies the second tactic only to the first subgoal.

Description
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_TCL (Thm_cont)

```
$THEN_TCL : (thm_tactical -> thm_tactical -> thm_tactical)
```

Synopsis

Composes two theorem-tacticals.

Description

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)

```
$THENC : (conv -> conv -> conv)
```

Synopsis
Applies two conversions in sequence.

Description
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.

Failure
(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.EVERY_CONV.

THENL (Tactical)

```
$THENL : (tactic -> tactic list -> tactic)
```

Synopsis
Applies a list of tactics to the corresponding subgoals generated by a tactic.

Description
If `T,T1,...,Tn` are tactics, `T THENL [T1;...;Tn]` is a tactic which applies `T` to a goal, and if it does not fail, applies the tactics `T1,...,Tn` to the corresponding subgoals, unless `T` completely solves the goal.

Failure
The application of `THENL` to a tactic and tactic list never fails. The resulting tactic fails if `T` fails when applied to the goal, 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`.

Uses
Applying different tactics to different subgoals.

See also
`Tactical.EVERY`, `Tactical.ORELSE`, `Tactical.THEN`.

theorems (DB)

```
theorems : string -> (string * thm) list
```

Synopsis
All the theorems stored in the named theory.

Description
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 =
    [("I_o_ID", |- !f. (I o f = f) /\ (f o I = f)), ("I_THM", |- !x. I x = x),
     ("W_THM", |- !f x. W f x = f x x),
     ("C_THM", |- !f x y. combin$C f x y = f y x),
     ("S_THM", |- !f g x. S f g x = f x (g x)), ("K_THM", |- !x y. K x y = x),
     ("o_ASSOC", |- !f g h. f o g o h = (f o g) o h),
     ("o_THM", |- !f g x. (f o g) x = f (g x))] : (string * thm) list
```

See also
`DB.thy`, `DB.fetch`, `DB.thms`, `DB.definitions`, `DB.axioms`, `DB.listDB`.

thm (Thm)

```
type thm
```

Synopsis
Type of theorems of the HOL logic.

Description
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)

```
thm_count : unit -> int
```

Synopsis

Returns the current value of the theorem counter.

Description

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

Failure

Never fails.

See also

`set_thm_count`, `timer`.

thms (DB)

```
thms : string -> (string * thm) list
```

Synopsis

All the theorems, definitions, and axioms stored in the named theory.

Description

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 =
    [("C_DEF", |- combin$C = (\f x y. f y x)),
     ("C_THM", |- !f x y. combin$C f x y = f y x), ("I_DEF", |- I = S K K),
     ("I_o_ID", |- !f. (I o f = f) /\ (f o I = f)), ("I_THM", |- !x. I x = x),
     ("K_DEF", |- K = (\x y. x)), ("K_THM", |- !x y. K x y = x),
     ("o_ASSOC", |- !f g h. f o g o h = (f o g) o h),
     ("o_DEF", |- !f g. f o g = (\x. f (g x))),
     ("o_THM", |- !f g x. (f o g) x = f (g x)),
     ("S_DEF", |- S = (\f g x. f x (g x))),
     ("S_THM", |- !f g x. S f g x = f x (g x)),
     ("W_DEF", |- W = (\f x. f x x)), ("W_THM", |- !f x. W f x = f x x)] :
  (string * thm) list
```

See also

DB.thy, DB.theorems, DB.axioms, DB.definitions, DB.fetch, DB.listDB.

thy (DB)

```
thy : string -> data list
```

Synopsis

Return the contents of a theory.

Description

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

```
- DB.thy "pair";
> val it =
    [(("pair", "ABS_PAIR_THM"), (|- !x. ?q r. x = (q,r), Db.Thm)),
     (("pair", "ABS_REP_prod"),
      (|- (!a. ABS_prod (REP_prod a) = a) /\
          !r. IS_PAIR r = (REP_prod (ABS_prod r) = r), Db.Def)),
     (("pair", "CLOSED_PAIR_EQ"),
      (|- !x y a b. ((x,y) = (a,b)) = (x = a) /\ (y = b), Db.Thm)),
        .
        .
        .
```

See also

DB.class, DB.data, DB.listDB, DB.theorems, DB.match, Theory.new_theory.

thy_addon (Theory)

```
type thy_addon
```

Synopsis

Type of theory additions.

Description

The type abbreviation thy_addon, declared as

```
type thy_addon = {sig_ps : (ppstream -> unit) option,
                  struct_ps : (ppstream -> unit) 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.

See also

Theory.adjoin_to_theory.

time (Lib)

```
time : ('a -> 'b) -> 'a -> 'b
```

Synopsis
Measure how long a function application takes.

Description
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`.

Example

```
- time (int_sort) (for 0 999 I);
runtime: 0.771s,    gctime: 0.121s,     systime: 0.771s.
> 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, 100, 101, 102, 103, 104, 105, 106, 107, 108,
   109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123,
   124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138,
   139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153,
   154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168,
   169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183,
   184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198,
   199, ...] : int list
```

See also
`Lib.start_time`, `Lib.end_time`.

TOP_DEPTH_CONV (Conv)

```
TOP_DEPTH_CONV : (conv -> conv)
```

Synopsis
Applies a conversion top-down to all subterms, retraversing changed ones.

Description
`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 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_goal (goalstackLib)

```
top_goal : unit -> term list * term
```

Synopsis
Returns the current goal of the subgoal package.

Description
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.

Uses
Examining the proof state after a proof fails.

See also
goalstackLib.b, goalstackLib.backup, backup_limit, goalstackLib.e, goalstackLib.expand, goalstackLib.expandf, goalstackLib.g, get_state, goalstackLib.p, print_state, goalstackLib.r, rotate, save_top_thm, goalstackLib.set_goal, set_state, goalstackLib.top_thm.

top_thm (goalstackLib)

```
top_thm : unit -> thm
```

Synopsis
Returns the theorem just proved using the subgoal package.

Description
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.

Uses
Accessing the result of an interactive proof session with the subgoal package.

See also
goalstackLib.b, goalstackLib.backup, backup_limit, goalstackLib.e, goalstackLib.expand, goalstackLib.expandf, goalstackLib.g, get_state, goalstackLib.p, print_state, goalstackLib.r, rotate, save_top_thm, goalstackLib.set_goal, set_state, goalstackLib.top_goal.

total (Lib)

```
total : ('a -> 'b) -> 'a -> 'b option
```

Synopsis
Converts a partial function to a total function.

Description
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 second argument to the third argument raises `Interrupt`.

Example
```
- 3 div 0;
! Uncaught exception:
! Div

- total (op div) (3,0);
> val it = NONE : int option

- (partial Div o total) (op div) (3,0);
! Uncaught exception:
! Div
```

See also
`Lib.partial`.

tprove (Defn)

```
tprove : defn * tactic -> thm * thm
```

Synopsis
Prove termination of a `defn`.

Description

`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)

```
trace : string * int -> ('a -> 'b) -> 'a -> 'b
```

Synopsis

Invoke a function with a specified level of tracing.

Description

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";

- 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.
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.
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.
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.
  solved with 2 MESON inferences.

> val it = |- !P. (!x. ?y. P x y) = ?f. !x. P x (f x) : thm

- traces();

> val it =
    [{default = 1, name = "meson", trace_level = 1},
     {default = 10, name = "Subgoal number", trace_level = 10},
     {default = 0, name = "Rewrite", trace_level = 0},
     {default = 0, name = "Ho_Rewrite", trace_level = 0}]
```

See also

Feedback, Feedback.register_trace, Feedback.reset_trace, Feedback.reset_traces, Feedback.set_trace, Feedback.traces, Lib.with_flag.

traces (Feedback)

```
traces : unit -> {name : string, current_value : int,
                  default_value : int, maximum : int} list
```

Synopsis
Returns a list of registered tracing variables.

Description
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.

Example

```
- traces();
> val it =
    [{default = 10, name = "Subgoal number", trace_level = 10},
     {default = 0, name = "Rewrite", trace_level = 0},
     {default = 0, name = "Ho_Rewrite", trace_level = 0}]
```

See also
`Feedback.register_trace`, `Feedback.set_trace`, `Feedback.reset_trace`, `Feedback.reset_traces`, `Feedback.trace`.

TRANS (Thm)

```
TRANS : (thm -> thm -> thm)
```

Synopsis
Uses transitivity of equality on two equational theorems.

Description

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`.

try (Lib)

```
try : ('a -> 'b) -> 'a -> 'b
```

Synopsis

Apply a function and print any exceptions

Description

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);
! Uncaught exception:
! HOL_ERR

- try mk_comb (T,F);

Exception raised at Term.mk_comb:
incompatible types
! Uncaught exception:
! HOL_ERR
```

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)

```
TRY : (tactic -> tactic)
```

Synopsis
Makes a tactic have no effect rather than fail.

Description
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, VALID.`

TRY_CONV (Conv)

```
TRY_CONV : (conv -> conv)
```

Synopsis
Attempts to apply a conversion; applies identity conversion in case of failure.

Description
`TRY_CONV c "t"` attempts to apply the conversion `c` to the term `"t"`; if this fails, then the identity conversion 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"` returns `|- t = t`.

Failure
Never fails.

See also
`Conv.ALL_CONV.`

trye (Lib)

```
trye : ('a -> 'b) -> 'a -> 'b
```

Synopsis
Maps exceptions into HOL_ERR

Description
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 [];
! Uncaught exception:
! Empty

- trye hd [];
! Uncaught exception:
! HOL_ERR

- trye (fn _ => raise Interrupt) 1;
> Interrupted
```

See also
Lib, Feedback.Raise, Lib.try.

tryfind (Lib)

```
tryfind : ('a -> 'b) -> 'a list -> 'b
```

Synopsis
Returns the result of the first successful application of a function to the elements of a list.

Description

`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`.

type_abbrev (Parse)

```
Parse.type_abbrev : string * hol_type -> unit
```

Synopsis

Establishes a type abbreviation.

Description

A call to `type_abbrev(s,ty)` sets up a type abbreviation that will cause the parser to treat the string `s` to be treated 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

Never fails.

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

- ``:num <- bool``;
> val it = ``:bool -> num`` : hol_type
```

Comments

Pretty-printing abbreviations well is complicated, and as yet unimplemented. 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.

See also

`add_infix_type`, `add_type`.

type_of (Term)

```
type_of : term -> hol_type
```

Synopsis

Returns the type of a term.

Failure

Never fails.

Example

```
- type_of boolSyntax.universal;
> val it = ‘:('a -> bool) -> bool‘ : hol_type
```

type_rws (bossLib)

```
type_rws : string -> thm list
```

Synopsis

List rewrites for a concrete type.

Description

An application `type_rws s`, where `s` is the name of a declared datatype, returns a list of rewrite rules corresponding to the types. 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 `s` is not the name of a declared datatype.

Example

```
- type_rws "list";

> val it =
    [|- (!v f. case v f [] = v) /\ !v f a0 a1. case v f (a0::a1) = f a0 a1,
     |- !a1 a0. ~([] = a0::a1),
     |- !a1 a0. ~(a0::a1 = []),
     |- !a0 a1 a0' a1'. (a0::a1 = a0'::a1') = (a0 = a0') /\ (a1 = a1')]

- Hol_datatype `point = <| x:num ; y:num |>`;
<<HOL message: Defined type: "point">>

- type_rws "point";
> val it =
    [|- !f a0 a1. case f (point a0 a1) = f a0 a1,
     |- !a0 a1 a0' a1'.
          (point a0 a1 = point a0' a1') = (a0 = a0') /\ (a1 = a1'),
     |- !z x p. p with <|y := x; x := z|> = p with <|x := z; y := x|>,
     |- (!x p. (p with y := x).x = p.x) /\ (!x p. (p with x := x).y = p.y) /\
        (!x p. (p with x := x).x = x) /\ !x p. (p with y := x).y = x,
     |- (!n n0. (point n n0).x = n) /\ !n n0. (point n n0).y = n0,
     |- (!n1 n n0. point n n0 with x := n1 = point n1 n0) /\
        !n1 n n0. point n n0 with y := n1 = point n n1,
     |- (!p. p with x := p.x = p) /\ !p. p with y := p.y = p,
     |- (!x2 x1 p. p with <|x := x1; x := x2|> = p with x := x1) /\
        !x2 x1 p. p with <|y := x1; y := x2|> = p with y := x1,
     |- (!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|>]
```

Comments

`RW_TAC` and `SRW_TAC` automatically include these rewrites.

See also

`bossLib.rewrites`, `bossLib.RW_TAC`.

type_subst (Type)

```
type_subst : hol_type subst -> hol_type -> hol_type
```

Synopsis
Instantiates types in a type.

Description
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 # ’b‘ : hol_type

- type_subst [Type‘:’a # ’b‘ |-> Type ‘:num‘, alpha |-> bool]
             (Type‘:’a # ’b‘);
> val it = ‘:bool # ’b‘ : hol_type
```

See also
`Term.inst`, `Thm.INST_TYPE`, `Lib.|->`, `Term.subst`.

type_var_in (Type)

```
type_var_in : hol_type -> hol_type -> bool
```

Synopsis
Checks if a type variable occurs in a type.

Description
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_vars : hol_type -> hol_type list
```

Synopsis
Returns the set of type variables in a type.

Description
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 = [`:'a`, `:'b`] : hol_type list
```

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

See also

Type.type_varsl, Type.type_var_in, Type.exists_tyvars, Type.polymorphic, Term.free_vars.

type_vars_in_term (Term)

```
type_vars_in_term : term -> hol_type list
```

Synopsis

Return the type variables occurring in a term.

Description

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 = [`:'b`, `:'a`] : hol_type list
```

See also

Term.free_vars, Type.type_vars.

type_varsl (Type)

```
type_varsl : hol_type list -> hol_type list
```

Synopsis

Returns the set of type variables in a list of types.

Description

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 = [':'a', ':'b'] : 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_var_in`, `Type.exists_tyvars`, `Type.polymorphic`, `Term.free_vars`.

TypeBase

```
structure TypeBase
```

Synopsis

A database of facts stemming from datatype declarations

Description

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

```
- Hol_datatype `tree = Leaf
                     | Node of 'a => tree => tree`;
<<HOL message: Defined type: "tree">>
> val it = () : unit

- TypeBase.read "tree";
> val it =
 SOME----------------------
     ----------------------
     HOL datatype: "tree"
     Primitive recursion:
      |- !f0 f1.
           ?fn.
             (!a. fn (Leaf a) = f0 a) /\
             !a0 a1. fn (Node a0 a1) = f1 a0 a1 (fn a0) (fn a1)
     Case analysis:
      |- (!f f1 a. case f f1 (Leaf a) = f a) /\
         !f f1 a0 a1. case f f1 (Node a0 a1) = f1 a0 a1
     Size:
      |- (!a. tree_size (Leaf a) = 1 + a) /\
         !a0 a1. tree_size (Node a0 a1) = 1 + (tree_size a0 + tree_size a1)
     Induction:
      |- !P.
           (!n. P (Leaf n)) /\ (!t t0. P t /\ P t0 ==> P (Node t t0)) ==>
           !t. P t
     Case completeness: |- !t. (?n. t = Leaf n) \/ ?t' t0. t = Node t' t0
     One-to-one:
      |- (!a a'. (Leaf a = Leaf a') = (a = a')) /\
         !a0 a1 a0' a1'.
           (Node a0 a1 = Node a0' a1') = (a0 = a0') /\ (a1 = a1')
     Distinctness: |- !a1 a0 a. ~(Leaf a = Node a0 a1) : tyinfo option
```

See also

bossLib.Hol_datatype.

types (Theory)

```
types : string -> (string * int) list
```

Synopsis

Lists the types in the named theory.

Description

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 =
    [("one", 0), ("option", 1), ("prod", 2), ("sum", 2),
     ("fun", 2), ("ind", 0), ("bool", 0), ("num", 0),
     ("recspace", 1), ("list", 1)] : (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`.

tyvars (hol88Lib)

```
Compat.tyvars : term -> type list
```

Synopsis

Returns a list of the type variables free in a term.

Description

Found in the hol88 library. When applied to a term, `tyvars` returns a list (possibly empty) of the type variables which are free in the term.

Failure

Never fails. The function is not accessible unless the hol88 library has been loaded.

Example

```
- theorem "pair" "PAIR";
|- !x. (FST x,SND x) = x

- Compat.tyvars (concl PAIR);
val it = [(==‘:’b‘==),(==‘:’a‘==)] : hol_type list

- Compat.tyvars (--‘x + 1 = SUC x‘--);
[] : hol_type list
```

Comments

`tyvars` does not appear in hol90; use `type_vars_in_term` instead. WARNING: the order of the list returned from `tyvars` need not be the same as that returned from `type_vars_in_term`.

In the current HOL logic, there is no binding operation for types, so ‘is free in’ is synonymous with ‘appears in’.

See also

`hol88Lib.tyvarsl`.

tyvarsl (hol88Lib)

```
Compat.tyvarsl : (term list -> type list)
```

Synopsis

Returns a list of the type variables free in a list of terms.

Description

Found in the hol88 library. When applied to a list of terms, `tyvarsl` returns a list (possibly empty) of the type variables which are free in any of those terms.

Failure

Never fails. The function is not accessible unless the hol88 library has been loaded.

Example

```
- tyvarsl [--‘!x. x = 1‘--, --‘!x:’a. x = x‘--];
[(==‘:’a‘==)] : hol_type list
```

Uses

Finding all the free type variables in the assumptions of a theorem, as a check on the validity of certain inferences.

Comments

`tyvarsl` does not appear in hol90. In the current HOL logic, there is no binding operation for types, so 'is free in' is synonymous with 'appears in'.

See also

`hol88Lib.tyvars`.

U (Lib)

```
U : ''a list list -> ''a list
```

Synopsis

Takes the union of a list of sets.

Description

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.

High performance set operations may be found in the ML Standard Basis Library.

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`.

uncurry (Lib)

```
uncurry : ('a -> 'b -> 'c) -> ('a * 'b) -> 'c
```

Synopsis
Converts a function taking two arguments into a function taking a single paired argument.

Description
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)

```
UNCURRY_CONV : conv
```

Synopsis
Uncurrys an application of an abstraction.

Example

```
- UNCURRY_CONV (Term `(\x y. x + y) 1 2`);
> val it = |- (\x y. x + y) 1 2 = (\(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)

```
UNCURRY_EXISTS_CONV : conv
```

Synopsis

Uncurrys consecutive existential quantifications into a paired existential quantification.

Example

```
- 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

- 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)

```
UNCURRY_FORALL_CONV : conv
```

Synopsis
Uncurrys consecutive universal quantifications into a paired universal quantification.

Example

```
- 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

- 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`.

UNDISCH (Drule)

```
UNDISCH : (thm -> thm)
```

Synopsis
Undischarges the antecedent of an implicative theorem.

Description

```
    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 the hypotheses, it will not be duplicated. However, unlike `DISCH`, if the antecedent is alpha-equivalent to one of the hypotheses, it will still be added to the hypotheses.

See also

Thm.DISCH, Drule.DISCH_ALL, Tactic.DISCH_TAC, Thm_cont.DISCH_THEN, Tactic.FILTER_DISCH_TAC, Thm_cont.FILTER_DISCH_THEN, Drule.NEG_DISCH, Tactic.STRIP_TAC, Drule.UNDISCH_ALL, Tactic.UNDISCH_TAC.

UNDISCH_ALL (Drule)

```
UNDISCH_ALL : (thm -> thm)
```

Synopsis

Iteratively undischarges antecedents in a chain of implications.

Description

```
 A |- t1 ==> ... ==> tn ==> t
------------------------------  UNDISCH_ALL
     A, t1, ..., tn |- t
```

Note that UNDISCH_ALL treats "~u" as "u ==> F".

Failure

Unlike UNDISCH, UNDISCH_ALL will, when called on something other than an implication or negation, return its argument unchanged rather than failing.

Comments

Identical terms which are repeated in A, "t1", ..., "tn" will not be duplicated in the hypotheses of the resulting theorem. However, if two or more alpha-equivalent terms appear in A, "t1", ..., "tn", then each distinct term will appear in the result.

See also

Thm.DISCH, Drule.DISCH_ALL, Tactic.DISCH_TAC, Thm_cont.DISCH_THEN, Drule.NEG_DISCH, Tactic.FILTER_DISCH_TAC, Thm_cont.FILTER_DISCH_THEN, Tactic.STRIP_TAC, Drule.UNDISCH, Tactic.UNDISCH_TAC.

UNDISCH_TAC (Tactic)

```
UNDISCH_TAC : term -> tactic
```

Synopsis

Undischarges an assumption.

Description

```
        A ?- t
====================  UNDISCH_TAC v
 A - {v} ?- v ==> t
```

Failure

`UNDISCH_TAC` will fail if `"v"` is not an assumption.

Comments

`UNDISCH`arging `v` will remove all assumptions which are identical to `v`, but those which are alpha-equivalent will remain.

See also

`Thm.DISCH`, `Drule.DISCH_ALL`, `Tactic.DISCH_TAC`, `Thm_cont.DISCH_THEN`, `Drule.NEG_DISCH`, `Tactic.FILTER_DISCH_TAC`, `Thm_cont.FILTER_DISCH_THEN`, `Tactic.STRIP_TAC`, `Drule.UNDISCH`, `Drule.UNDISCH_ALL`.

UNDISCH_THEN (Thm_cont)

```
Thm_cont.UNDISCH_THEN : term -> thm_tactic -> tactic
```

Synopsis

Discharges the assumption given and passes it to a theorem-tactic.

Description

`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 u {t1} ?- t
===============  f (ASSUME t1)
 B u {t1} ?- 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.PAT_ASSUM`, `Thm.DISCH`, `Drule.DISCH_ALL`, `Tactic.DISCH_TAC`, `Thm_cont.DISCH_THEN`, `Drule.NEG_DISCH`, `Tactic.FILTER_DISCH_TAC`, `Thm_cont.FILTER_DISCH_THEN`, `Tactic.STRIP_TAC`, `Drule.UNDISCH`, `Drule.UNDISCH_ALL`, `Tactic.UNDISCH_TAC`.

union (Lib)

```
union : ''a list -> ''a list -> ''a list
```

Synopsis

Computes the union of two 'sets'.

Description

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.

High performance set operations may be found in the ML Standard Basis Library.

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`.

universal (boolSyntax)

```
universal : term
```

Synopsis

Constant denoting universal quantification.

Description

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.conditional`, `boolSyntax.bool_case`, `boolSyntax.let_tm`, `boolSyntax.arb`.

UNPBETA_CONV (PairRules)

```
UNPBETA_CONV : (term -> conv)
```

Synopsis
Creates an application of a paired abstraction from a term.

Description
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`.

unzip (Lib)

```
unzip : ('a * 'b) list -> ('a list * 'b list)
```

Synopsis
Converts a list of pairs into a pair of lists.

Description
`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)

```
update_overload_maps :
  string -> ({Name : string, Thy : string} list *
             {Name : string, Thy : string} list) -> unit
```

Synopsis

Adds to the parser's overloading maps.

Description

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)

```
upto : int -> int -> int list
```

Synopsis
Builds a list of integers

Description
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]
```

uptodate_term (Theory)

```
uptodate_term : term -> bool
```

Synopsis
Tells if a term is out of date.

Description
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 ‘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

- val M = Term ‘!x. 0 < fact x‘;
> val M = ‘!x. 0 < fact x‘ : term

- uptodate_term M;
> val it = true : bool

- delete_const "fact";
> val it = () : unit

- uptodate_term M;
> val it = false : bool
```

See also

`Theory.uptodate_type`, `Theory.uptodate_thm`.

uptodate_thm (Theory)

```
uptodate_thm : thm -> bool
```

Synopsis

Tells if a theorem is out of date.

Description

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.

All items from ancestor theories are fixed, and unable to be overwritten, thus are always up-to-date.

Failure
Never fails.

Example

```
- 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

- val th = EVAL (Term ‘fact 3‘);
> val th = |- fact 3 = 6 : thm

- uptodate_thm th;
> val it = true : bool

- delete_const "fact";
> val it = () : unit

- uptodate_thm th;
> val it = false : 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)

```
uptodate_type : hol_type -> bool
```

Synopsis
Tells if a type is out of date.

Description

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‘;
<<HOL message: Defined type: "foo">>
> val it = () : unit

- val ty = Type ‘:’a foo list‘;
> val ty = ‘:’a 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`.

var_compare (Term)

```
var_compare : term * term -> order
```

Synopsis
Total ordering on variables.

Description
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_occurs (Term)

```
var_occurs : term -> term -> bool
```

Synopsis
Check if a variable occurs in free in a term.

Description
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)

```
variant : term list -> term -> term
```

Synopsis

Modifies a variable name to avoid clashes.

Description

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 normally 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.

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 [Term‘y:bool‘, Term‘z:bool‘] (Term‘x:bool‘);
> val it = ‘x‘ : term

- variant [Term‘x:bool‘, Term‘x’:num‘, Term‘x’’:num‘] (Term ‘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 style of renaming can be altered by modifying the reference variable `Globals.priming`:

```
- with_flag (priming,SOME "_")
  (uncurry variant)
  ([Term‘x:bool‘, Term‘x’:num‘, Term‘x’’:num‘], Term ‘x:bool‘);

> val it = ‘x_1‘ : term
```

Uses

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.

See also

`Term.genvar`, `Term.prim_variant`, `Globals.priming`.

version (Globals)

```
Globals.version : string
```

Synopsis

The version of the `HOL` system being run.

Example

```
- Globals.version;

val it = "Kananaskis" : string
```

W (Lib)

```
W : ('a -> 'a -> 'b) -> 'a -> 'b
```

Synopsis
Duplicates function argument : `W f x` equals `f x x`.

Description
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";
- tautLib.TAUT_PROVE (Term `(a = 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.A`, `Lib.B`, `Lib.C`, `Lib.I`, `Lib.K`, `Lib.S`.

WARNING_outstream (Feedback)

```
WARNING_outstream : TextIO.outstream ref
```

Synopsis
Controlling output stream used when printing HOL_WARNING

Description
The value of reference cell WARNING_outstream controls where HOL_WARNING prints its argument.

The default value of WARNING_outstream is TextIO.stdOut.

Example
```
- val ostrm = TextIO.openOut "foo";
> val ostrm = <outstream> : outstream

- WARNING_outstream := ostrm;
> val it = () : unit

- HOL_WARNING "Module" "Function" "Sufferin' Succotash!";
> val it = () : unit

- TextIO.closeOut ostrm;
> val it = () : unit

- val istrm = TextIO.openIn "foo";
> val istrm = <instream> : instream

- print (TextIO.inputAll istrm);
<<HOL warning: Module.Function: Sufferin' Succotash!>>
```

See also
Feedback, Feedback.HOL_WARNING, Feedback.ERR_outstream, Feedback.MESG_outstream, Feedback.emit_WARNING.

WARNING_to_string (Feedback)

```
WARNING_to_string : (string -> string -> string -> string) ref
```

Synopsis
Alterable function for formatting HOL_WARNING

Description
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.

WF_REL_TAC (bossLib)

```
WF_REL_TAC : term quotation -> tactic
```

Synopsis

Start termination proof.

Description

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 wellfounded 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 wellfounded 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 wellfoundedness 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 wellfounded relations. The most common starting point for dealing with termination problems for recursive functions is to find some function, known as a 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`, `TotalDefn.guessR`.

WF_REL_TAC (TotalDefn)

```
WF_REL_TAC : term quotation -> tactic
```

Synopsis

Initiate a termination proof.

Description

`bossLib.WF_REL_TAC` is identical to `TotalDefn.WF_REL_TAC`.

See also

`bossLib.WF_REL_TAC`.

with_exn (Lib)

```
with_exn : ('a -> 'b) -> 'a -> exn -> 'b
```

Synopsis

Apply a function to an argument, raising supplied exception on failure.

Description

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");
! Uncaught exception:
! Fail  "My kingdom for a horse"

- with_exn (fn _ => raise Interrupt) 1 (Fail "My kingdom for a horse");
> Interrupted.
```

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)

```
with_flag : 'a ref * 'a -> ('b -> 'c) -> 'b -> 'c
```

Synopsis
Apply a function under a particular flag setting.

Description
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; print "\n");
> val print_term_nl = fn : term -> unit

- with_flag (show_types, true) print_term_nl (concl T_DEF);
T = ((\(x :bool). x) = (\(x :bool). x))
> val it = () : unit

- print_term_nl (concl T_DEF);
T = ((\(x. x) = (\x. x))
> val it = () : unit
```

See also
`Feedback.traces`, `Feedback.register_btrace`, `Feedback.trace`, `Lib.time`.

words2 (Lib)

```
words2 : string -> string -> string list
```

Synopsis
Splits a string into a list of substrings, breaking at occurrences of a specified character.

Description
`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)

```
wrap_exn : string -> string -> exn -> exn
```

Synopsis
Adds supplementary information to an application of `HOL_ERR`.

Description
`wrap_exn s1 s2 (HOL_ERR{origin_structure,origin_function,message})` where `s1` typically denotes a structure and `s2` typically denotes a function, returns

`HOL_ERR{origin_structure=s1,origin_function=s2,message}`

where `origin_structure` and `origin_function` have been added to the `message` field. This can be used to achieve a kind of backtrace when an error occurs.

In MoscowML, the `interrupt` signal in Unix is mapped into the `Interrupt` exception. If `wrap_exn` were to translate an interrupt into a `HOL_ERR` exception, crucial information might be lost. For this reason, `wrap_exn s1 s2 Interrupt` raises the `Interrupt` exception.

Every other exception is mapped into an application of `HOL_ERR` by `wrap_exn`.

Failure
Never fails.

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 test_exn = mk_HOL_ERR "Foo" "bar" "incomprehensible input";
> val test_exn = HOL_ERR : exn

- wrap_exn "Fred" "barney" test_exn;
> val it = HOL_ERR : exn

- print(exn_to_string it);

Exception raised at Fred.barney:
Foo.bar - incomprehensible input
```

The following example shows how `wrap_exn` treats the `Interrupt` exception.

```
- wrap_exn "Fred" "barney" Interrupt;
> Interrupted.
```

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 : exn

- print(exn_to_string it);

Exception raised at Fred.barney:
Div
```

See also

`Feedback`, `Feedback.HOL_ERR`, `Feedback.with_exn`.

X_CASES_THEN (Thm_cont)

```
X_CASES_THEN : term list list -> thm_tactical
```

Synopsis

Applies a theorem-tactic to all disjuncts of a theorem, choosing witnesses.

Description

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)

```
X_CASES_THENL : term list list -> thm_tactic list -> thm_tactic
```

Synopsis

Applies theorem-tactics to corresponding disjuncts of a theorem, choosing witnesses.

Description

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 ith 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)

```
X_CHOOSE_TAC : term -> thm_tactic
```

Synopsis
Assumes a theorem, with existentially quantified variable replaced by a given witness.

Description
`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` or `t`, 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)

```
X_CHOOSE_THEN : (term -> thm_tactical)
```

Synopsis

Replaces existentially quantified variable with given witness, and passes it to a theorem-tactic.

Description

`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` or `t`, 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 (Term‘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)

```
X_FUN_EQ_CONV : (term -> conv)
```

Synopsis

Performs extensionality conversion for functions (function equality).

Description

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)

```
X_GEN_TAC : (term -> tactic)
```

Synopsis
Specializes a goal with the given variable.

Description
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, Drule.GENL, Drule.GEN_ALL, Thm.SPEC, Drule.SPECL, Drule.SPEC_ALL, Tactic.SPEC_TAC, Tactic.STRIP_TAC.

X_SKOLEM_CONV (Conv)

```
X_SKOLEM_CONV : (term -> conv)
```

Synopsis
Introduces a user-supplied Skolem function.

Description

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 (bossLib)

```
xDefine : string -> term quotation -> thm
```

Synopsis

General-purpose function definition facility.

Description

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`.

xDefine (TotalDefn)

```
xDefine : string -> term quotation -> thm
```

Synopsis
General purpose function definition facility.

Description
`bossLib.xDefine` is identical to `TotalDefn.xDefine`.

See also
`bossLib.xDefine`.

zip (Lib)

```
zip : 'a list -> 'b list -> ('a * 'b) list
```

Synopsis
Transforms a pair of lists into a list of pairs.

Description
`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`.

|-> (Lib)

```
op |-> : 'a * 'b -> {redex : 'a, residue : 'b}
```

Synopsis
Infix operator for building a component of a substitution.

Description
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 = ‘:’b -> ’c‘ : hol_type
```

See also
`Lib.subst`, `Type.type_subst`, `Term.subst`, `Term.inst`, `Thm.SUBST`.

Index