[For hol98 Taupo-6] **February 14, 2001**

The HOL System
REFERENCE

University of Cambridge DSTO SRI International

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 hol98 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 all the pre-defined ML identifiers in the HOL system, except the identifiers that are bound to pre-proved theorems (for these, see chapter two). 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.

##

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

Synopsis
Maps a pair of functions through a pair.

Description
`##` is an infix operator such that the call `(f ## g) (x, y)` returns the value `(f x, g y)`.

Failure
Never fails.

Example
```
- ((fn x => x + 1) ## not) (3, false);
> val it = (4, true) : int * bool
```

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

++

```
simpLib.++ : simpset * ssdata -> simpset
```

Synopsis
Augments 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 (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
```

Comments
The ++ identifier is not an infix by default, and so needs to be declared as such at the ML top-level loop, e.g.:

```
- infix ++;
> infix 0 ++
```

See also
`mk_simpset, rewrites, SIMP_CONV, bool_ss, UNWIND_ss`

ABS

```
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 = Parse.Term ‘m:num‘
  in
      ABS m (REFL m)
  end;

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

See also

ETA_CONV, EXT, MK_ABS.

ABS_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

```
- let val M = Parse.Term ‘\x. 1 = x‘
  in
  ABS_CONV SYM_CONV M
  end;

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

See also

RAND_CONV, RATOR_CONV, SUB_CONV.

ACCEPT_TAC

```
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
`MATCH_ACCEPT_TAC`.

aconv

```
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.

Failure
Never fails.

Example
A simple case of alpha-convertibility is the renaming of a single quantified variable:

```
- let val M = Parse.Term `?x. x = T`
      val N = Parse.Term `?y. y = T`
  in
  aconv M N
  end;
true : bool
```

See also
`ALPHA`, `ALPHA_CONV`.

AC_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

```
   - let val M = Parse.Term
                 ‘x + (SUC t) + ((3 + y) + z) = 3 + (SUC t) + x + y + z‘
     in
     AC_CONV(ADD_ASSOC,ADD_SYM) M
     end;

   |- (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

`SYM_CONV`.

ADD_ASSUM

```
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
ASSUME, 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.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.term
- dest_comb it;
> val it = {Rand = ‘1011‘, Rator = ‘binary_of‘} :
     {Rand : Term.term, Rator : 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

`add_numeral_form`

add_implicit_rewrites

```
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`, `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 subsequent attempts to call the term parser failing due to precedence conflicts.

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.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 = {Rand = ‘y‘, Rator = ‘$+ x‘} :
      {Rand : Term.term, Rator : 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.term
- dest_abs it;
> val it = {Body = ‘x + y‘, Bvar = ‘$+‘}
           : {Body : Term.term, Bvar : 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 variableaddition 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.term
- dest_comb it;
> val it =
    {Rand = ‘z‘, Rator = ‘$+ (x + y)‘}
    : {Rand : Term.term, Rator : 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.term
- dest_comb it;
> val it =
    {Rand = ‘q + r‘, Rator = ‘$/\ p‘}
    : {Rand : Term.term, Rator : 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
! Uncaught exception:
! HOL_ERR <poly>
```

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
! Uncaught exception:
! HOL_ERR <poly>
```

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‘;
! Uncaught exception:
! HOL_ERR <poly>
- Lib.try Term‘p + q‘;

Exception raised at Parse.Term:
Grammar introduces precedence conflict between tokens + and +
! Uncaught exception:
! HOL_ERR <poly>
```

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

`add_binder, add_rule, add_listform, 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
`add_rule`.

add_numeral_form

```
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 the overloaded operator `&`. When a numeral doesn't have a character suffix, this means that it has been made an argument to the function `fromNum`, and so might take on different types, depending on the context.

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.term
   - type_of it;
   > val it = ‘:int‘ : Type.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.term
   - type_of it;
   > val it = ‘:num # int‘ : Type.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
   - Term‘f 3n /\ p‘;
   > val it = ‘f 3 /\ p‘ : Term.term
```

Comments

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

See also

add_bare_numeral_form, show_numeral_types

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 to 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
```

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

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

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

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

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

The possible ParenStyle values describe when parentheses should be added to terms. The OnlyIfNecessary value will cause parentheses to be added only when required to disambiguate syntax. The ParoundName will cause parentheses to be added if necessary, or where the head symbol has the given term_name and where this term is not the argument of a function with the same head name. This style of parenthesisation is used with tuples, for example. The ParoundPrec value is similar, but causes parentheses to be added when the term is the argument to a function with a different precedence level. 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 /\.

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

allowed_term_constant

```
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
`constants`, `is_constant`, `new_alphanum`, `new_special_symbol`, `special_symbols`, `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_term_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.

See also
`allowed_term_constant`

ALL_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
`NO_CONV`, `REFL`.

ALL_TAC

```
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_TAC THENL [ALL_TAC;tac]`, applied to a goal `g`, applies `INDUCT_TAC` 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
`NO_TAC`, `REPEAT`, `THENL`.

ALL_THEN

```
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
ALL_TAC, FAIL_TAC, NO_TAC, NO_THEN, THEN_TCL, ORELSE_TCL.

ALPHA

```
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.

Example

```
- let val M = Term`!x:num. x = x`
      val N = Term`!y:num. y = y`
  in
      ALPHA M N
  end;

> val it = |- (!x. x = x) = (!y. y = y) : Thm.thm
```

See also
aconv, ALPHA_CONV, GEN_ALPHA_CONV.

ALPHA_CONV

```
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
`ALPHA`, `GEN_ALPHA_CONV`.

ancestry

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

Synopsis
Gets a list of the (proper) ancestry of a theory.

Description
A call to `ancestry "th"` returns a list of all the proper ancestors (i.e. parents, parents of parents, etc.) of the theory `th`.

Failure
Fails if `"th"` is not an ancestor of the current theory.

See also
`parents`.

AND_EXISTS_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`.

See also
`EXISTS_AND_CONV`, `LEFT_AND_EXISTS_CONV`, `RIGHT_AND_EXISTS_CONV`.

AND_FORALL_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)`.

See also
`FORALL_AND_CONV`, `LEFT_AND_FORALL_CONV`, `RIGHT_AND_FORALL_CONV`.

ANTE_CONJ_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`.

ANTE_RES_THEN

```
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`.

See also
`IMP_RES_TAC`, `IMP_RES_THEN`, `MATCH_MP`, `RES_TAC`, `RES_THEN`.

AP_TERM

```
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
`AP_THM`, `MK_COMB`.

AP_TERM_TAC

```
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
`AP_TERM`, `AP_THM`.

AP_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
`AP_TERM, ETA_CONV, EXT, MK_COMB.`

AP_THM_TAC

```
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
`AP_TERM, AP_TERM_TAC, AP_THM, EXT.`

arity

```
arity : (string -> int)
```

Synopsis
Returns the arity of a type operator.

Description
`arity "op"` returns `n` if `op` is the name of an `n`-ary type operator (`n` can be 0), and otherwise fails.

Failure
`arity st` fails if `st` is not the name of a type constant or type operator.

See also
`is_type`.

ASM_CASES_TAC

```
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 = Parse.Term ‘u:bool‘
        val g = Parse.Term ‘(P:bool -> bool) u‘
    in
    ASM_CASES_TAC u ([],g)
    end;

    ([([--‘u‘--],  --‘P u‘--),
      ([--‘~u‘--], --‘P u‘--)], -) : tactic_result
```

Uses

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

See also

`BOOL_CASES_TAC`, `COND_CASES_TAC`, `DISJ_CASES_TAC`, `SPEC`, `STRUCT_CASES_TAC`.

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

`GEN_MESON_TAC`, `MESON_TAC`

ASM_REWRITE_RULE

```
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 with 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
`GEN_REWRITE_RULE`, `ONCE_ASM_REWRITE_RULE`, `PURE_ASM_REWRITE_RULE`, `PURE_ONCE_ASM_REWRITE_RULE`, `REWRITE_RULE`.

ASM_REWRITE_TAC

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

Synopsis
Rewrites a goal including 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 = [Parse.Term `(P:'a->bool) x`]
      val goal = Parse.Term `(P:'a->bool) x = (Q:'a -> bool) x`
  in
  ASM_REWRITE_TAC[](asm, goal)
  end;

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

- let val asm = [Parse.Term `~(P:'a->bool) x`]
      val goal = Parse.Term `(P:'a->bool) x = (Q:'a -> bool) x`
  in
  ASM_REWRITE_TAC[](asm, goal)
  end;

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

See also

basic_rewrites, FILTER_ASM_REWRITE_TAC, FILTER_ONCE_ASM_REWRITE_TAC, GEN_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_REWRITE_TAC, PURE_ASM_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_REWRITE_TAC, REWRITE_TAC, 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
Not clear to this author.

See also
SIMP_CONV, SIMP_RULE.

ASM_SIMP_TAC

```
simpLib.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
Here, hol_ss and the one assumption are used to demonstrate the proof of a simple arithmetic fact:

```
   - ASM_SIMP_TAC hol_ss [] ([Term‘x < y‘], Term‘x + y < y + y‘);
   > val it = ([], fn) : tactic_result
```

See also
++, bool_ss, FULL_SIMP_TAC, hol_ss, mk_simpset, SIMP_CONV, SIMP_TAC.

assert

```
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`.

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
`can`.

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

`assoc1, assoc2, rev_assoc, find, mem, tryfind, exists, forall.`

associate_restriction

```
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`

ASSUME

```
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`.

Comments
The type of `ASSUME` is shown by the system as `conv`.

See also
`ADD_ASSUM, REFL.`

ASSUME_TAC

```
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 = Parse.Term ‘(x:’a) = y‘
    val eq2 = Parse.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 = Parse.Term ‘(x:’a) = y‘
    val eq2 = Parse.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 = Parse.Term ‘(x:’a) = y‘
    val eq2 = Parse.Term ‘(y:’a) = z‘
    val goal = ([eq1,eq2],Parse.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
ACCEPT_TAC, IMP_RES_TAC, RES_TAC, STRIP_ASSUME_TAC.

ASSUM_LIST

```
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 ASSUMEd 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
`ASM_REWRITE_TAC`, `EVERY_ASSUM`, `IMP_RES_TAC`, `POP_ASSUM`, `POP_ASSUM_LIST`, `REWRITE_TAC`.

axiom

```
axiom : (string -> string -> thm)
```

Synopsis
Loads an axiom from a given theory segment of the current theory.

Description
A call of `axiom "thy" "ax"` returns axiom `ax` from the theory segment `thy`. The theory segment `thy` must be part of the current theory. The name `ax` is the name given to the axiom by the user when it was originally added to the theory segment (by a call to `new_axiom`). The name of the current theory segment can be abbreviated by `"-"`.

Failure
The call `axiom "thy" "ax"` will fail if the theory segment `thy` is not part of the current theory. It will also fail if there does not exist an axiom of name `ax` in theory segment `thy`.

Example
```
 - axiom "bool" "BOOL_CASES_AX";
val it = |- !t. (t = T) \/ (t = F) : thm
```

See also
`axioms`, `definition`, `new_axiom`, `print_theory`, `theorem`.

axioms

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

Synopsis
Returns the axioms of a given theory segment of the current theory.

Description
A call `axioms "thy"` returns the axioms of the theory segment `thy` together with their names. The theory segment `thy` must be part of the current theory. 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`). The name of the current theory segment can be abbreviated by `"-"`.

Failure
The call `axioms "thy"` will fail if the theory segment `thy` is not part of the current theory.

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),
   ("IMP_ANTISYM_AX",|- !t1 t2. (t1 ==> t2) ==> (t2 ==> t1) ==> (t1 = t2)),
   ("BOOL_CASES_AX",|- !t. (t = T) \/ (t = F))] : (string * thm) list
```

See also
`axiom, definitions, load_axiom, load_axioms, new_axiom, print_theory, theorems.`

b

```
b : (void -> void)
```

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

B

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

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

Comments
Not yet in hol90

Failure
Never fails.

See also
`##, C, I, K, o, S, W.`

backup

```
backup : (void -> void)
```

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

() : void

#e CONJ_TAC;;
OK..
2 subgoals
"TL[1;2;3] = [2;3]"

"HD[1;2;3] = 1"

() : void

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

() : void

#e (REWRITE_TAC[HD;TL]);;
OK..
goal proved
|- (HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])

Previous subproof:
goal proved
() : void
```

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

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

BETA_CONV

```
BETA_CONV : conv
```

Synopsis
Performs a simple 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

```
- let val tm = Parse.Term `(\x.x+1)y`
  in
  BETA_CONV tm
  end;
 val it = |- (\x. x + 1)y = y + 1 :thm

- let val tm = Parse.Term `(\x y. x+y)y`
  in
  BETA_CONV tm
  end;
val it = |- (\x y. x + y)y = (\y'. y + y') : thm
```

Comments
This primitive inference rule is actually not very primitive, since it does automatic bound variable renaming. It would be logically cleaner for this renaming to be derived rather than built-in, but since beta-reduction is so common this would slow the system down a lot. It is hoped to document the exact renaming algorithm used by `BETA_CONV` in the future.

See also
`BETA_RULE`, `BETA_TAC`, `LIST_BETA_CONV`, `PAIRED_BETA_CONV`, `RIGHT_BETA`, `RIGHT_LIST_BETA`.

BETA_RULE

```
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
`BETA_CONV`, `BETA_TAC`, `PAIRED_BETA_CONV`, `RIGHT_BETA`.

BETA_TAC

```
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
`BETA_CONV, BETA_TAC, PAIRED_BETA_CONV.`

binders

```
binders : (string -> term list)
```

Synopsis
Lists the binders in the named theory.

Description
The function `binders` 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 binders declared in the named theory.

Failure
Fails unless the given theory is an ancestor of the current theory.

Example

```
- binders "bool";
val it = [`$?!`, `$!`, `$@`] : term list

- binders "prod";
val it = [] : term list
```

See also
`ancestors, axioms, constants, definitions, infixes, new_binder, parents, types.`

binder_restrictions

```
binder_restrictions : unit -> (string * string) list
```

Synopsis
Shows the list of binder restrictions currently in force.

Description
`associate_restriction` is used to control the parsing and prettyprinting of restricted binders, which give the illusion of dependent types. The list of current restrictions is found by calling `binder_restrictions`. There are always at least the following restricted binders: [”!”,”?”,”@”,”
”].

Failure
Never fails.

Example

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

binder_restrictions();
[("DURING","RES_DURING"),("!","RES_FORALL"),("?","RES_EXISTS"),
 ("@","RES_SELECT"),("\\","RES_ABSTRACT")] : (string * string) list
```

See also
`associate_restrictions, delete_restriction`

BINOP_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.thm
```

See also

`FORK_CONV, LAND_CONV, RAND_CONV, RATOR_CONV`

body

```
body : (term -> term)
```

Synopsis

Returns the body of an abstraction.

Description

`body ‘\v. t‘` returns `‘t‘`.

Failure

Fails unless the term is an abstraction.

See also

`bvar, dest_abs.`

BODY_CONJUNCTS

```
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
`CONJ, CONJUNCT1, CONJUNCT2, CONJUNCTS, CONJ_TAC.`

bool

```
Type.bool : hol_type
```

Synopsis
Holds the logical type constant `bool`.

BOOL_CASES_TAC

```
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
ASM_CASES_TAC, COND_CASES_TAC, DISJ_CASES_TAC, STRUCT_CASES_TAC.

bool_EQ_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

```
bool_rewrites: unit -> rewrites
```

Synopsis

Contains a number of built-in tautologies used, by default, in rewriting.

Description

The variable `bool_rewrites` represents a kind of database 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
```

Uses

The `bool_rewrites` are automatically included in the simplifications performed by some of the rewriting tools.

The `bool_rewrites` used to include rules for reasoning about pairs in HOL:

```
   |- !x. FST x,SND x = x
   |- !x y. FST(x,y) = x
   |- !x y. SND(x,y) = y
```

However, because of recent changes in the system, the theory of pairs need not be loaded at the same time as the ”bool” theory, so the above rewrites can be accessed through `pairTheory.pair_rws`.

See also

`ABS_SIMP`, `AND_CLAUSES`, `COND_CLAUSES`, `EQ_CLAUSES`, `EXISTS_SIMP`, `FORALL_SIMP`, `FST`, `GEN_REWRITE_RULE`, `GEN_REWRITE_TAC`, `IMP_CLAUSES`, `NOT_CLAUSES`, `OR_CLAUSES`, `PAIR`, `REFL_CLAUSE`, `REWRITE_RULE`, `REWRITE_TAC`, `SND`, `set_bool_rewrites`, `add_bool_rewrites`.

bool_ss

```
boolSimps.bool_ss : simpset
```

Synopsis

Basic simpset containing standard propositional calculus rewrites, beta conversion, and eta conversion.

Description

The `bool_ss` simpset is almost at the base of the system-provided simpset hierarchy. Though not very powerful, it does include rewrite rules such as `|- T /\ P = P`, conversions to perform eta and beta reduction, and congruence rules to let simplification get additional contextual information as it descends through implications and congruences.

Failure

Can’t fail, as it is not a functional value.

Uses
The `bool_ss` simpset is an appropriate simpset to use at the base of new user-defined simpsets, and is also useful in its own right where 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
`hol_ss, pure_ss, SIMP_CONV, SIMP_TAC.`

butlast

```
butlast : (* list -> * 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
`last, hd, tl, el, null.`

bvar

```
bvar : (term -> term)
```

Synopsis
Returns the bound variable of an abstraction.

Description
`bvar ‘\v. t‘` returns `‘v‘.`

Failure
Fails unless the term is an abstraction.

See also
`body`, `dest_abs`.

C

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

Synopsis
Permutes first two arguments to curried function: `C f x y` = `f y x`.

Failure
Never fails.

See also
`##`, `B`, `I`, `K`, `o`, `S`, `W`.

can

```
can : ((* -> **) -> * -> 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
Never fails.

Example

```
#hd [];;
evaluation failed     hd

#can hd [];;
false : bool
```

See also
`assert`.

Cases

```
bossLib.Cases : tactic
```

Synopsis

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

Description

When applied to a universally quantified goal, `Cases` performs a case-split, based on the cases theorem for the type of the universally quantified variable 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.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

`Cases_on, Induct, STRUCT_CASES_TAC`

CASES_THENL

```
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
DISJ_CASES_THENL, X_CASES_THENL.

CBV_CONV

```
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.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.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.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`, `initial_rws`

CCONTR

```
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
`CONTR`, `CONTRAPOS`, `CONTR_TAC`, `NOT_ELIM`.

CCONTR_TAC

```
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
`CHECK_ASSUME_TAC`, `CCONTR`, `CCCONTR`, `CONTRAPOS`, `NOT_ELIM`.

CHANGED_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

```
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
`TRY`, `VALID`.

CHECK_ASSUME_TAC

```
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

`ACCEPT_TAC, ASSUME_TAC, CONTR_TAC, DISCARD_TAC, MATCH_ACCEPT_TAC.`

CHOOSE

```
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

`CHOOSE_TAC`, `EXISTS`, `EXISTS_TAC`, `SELECT_ELIM`.

CHOOSE_TAC

```
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 t1 THEN EXISTS_TAC "t+1" THEN
  UNDISCH_TAC "input (t:num) :bool" THEN MATCH_ACCEPT_TAC t2
```

See also

CHOOSE_THEN, X_CHOOSE_TAC.

CHOOSE_THEN

```
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
CHOOSE_TAC, X_CHOOSE_THEN.

clear_overloads_on

```
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.

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.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.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.thm
```

Uses
If overloading gets too confusing, this function should help to clear away one layer of supposedly helpful obfuscation.

See also
`overload_on`.

clear_prefs_for_term

```
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.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.term
```

See also
`prefer_form_with_tok`

combine

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

Synopsis
Converts 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.

See also
`split`.

concat

```
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, replacing `concatl` in the HOL distribution.

concl

```
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
`dest_thm`, `hyp`.

COND_CASES_TAC

```
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
`ASM_CASES_TAC`, `DISJ_CASES_TAC`, `STRUCT_CASES_TAC`.

COND_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.

CONJ

```
CONJ : (thm -> thm -> thm)
```

Synopsis
Introduces a conjunction.

Description

```
    A1 |- t1      A2 |- t2
   ------------------------  CONJ
     A1 u A2 |- t1 /\ t2
```

Failure
Never fails.

See also
BODY_CONJUNCTS, CONJUNCT1, CONJUNCT2, CONJ_PAIR, LIST_CONJ, CONJ_LIST, CONJUNCTS.

CONJUNCT1

```
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.

See also
BODY_CONJUNCTS, CONJUNCT2, CONJ_PAIR, CONJ, LIST_CONJ, CONJ_LIST, CONJUNCTS.

CONJUNCT2

```
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.

See also

BODY_CONJUNCTS, CONJUNCT1, CONJ_PAIR, CONJ, LIST_CONJ, CONJ_LIST, CONJUNCTS.

conjuncts

```
hol88Lib.conjuncts : term -> term list
```

Synopsis

Iteratively splits conjunctions into a list of conjuncts.

Description

Found in the hol88 library. `conjuncts (--‘t1 /\ ... /\ tn‘--)` returns `[t1,...,tn]`. The argument term may be any tree of conjunctions. It need not have the form

```
  --‘t1 /\ (t2 /\ ( ... /\ tn)...)‘--
```

A term that is not a conjunction is simply returned as the sole element of a list. Note that

```
   conjuncts(list_mk_conj([t1,...,tn]))
```

will not return `[t1,...,tn]` if any of `t1,...,tn` are conjunctions.

Failure

Never fails.

Example

```
  - list_mk_conj [(--‘a /\ b‘--),(--‘c /\ d‘--),(--‘e /\ f‘--)];
  > val it = (--‘(a /\ b) /\ (c /\ d) /\ e /\ f‘--) : term

  - conjuncts it,
  val it = [(--‘a‘--),(--‘b‘--),(--‘c‘--),(--‘d‘--),
            (--‘e‘--),(--‘f‘--)] : term list

  - list_mk_conj it,
  val it = (--‘a /\ b /\ c /\ d /\ e /\ f‘--) : term

  - conjuncts (--‘1‘--);
  val it = [--‘1‘--] : term list
```

Comments

The function `conjuncts` is equivalent to the standard function `strip_conj`, so called in order to be consistent with all the other `strip_` routines. Because `conjuncts` splits both the left and right sides of a conjunction, this operation is not the inverse of `list_mk_conj`. It may be useful to introduce `list_dest_conj` for splitting only the right tails of a conjunction.

See also
`list_mk_conj, dest_conj.`

CONJUNCTS

```
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
`BODY_CONJUNCTS, CONJ_LIST, LIST_CONJ, CONJ, CONJUNCT1, CONJUNCT2, CONJ_PAIR.`

CONJUNCTS_CONV

```
CONJUNCTS_CONV : ((term # term) -> thm)
```

Synopsis
Prove equivalence under idempotence, symmetry and associativity of conjunction.

Description
`CONJUNCTS_CONV` takes a pair of terms `"t1"` and `"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 ("(P /\ Q) /\ R", "R /\ (Q /\ R) /\ P");;
|- (P /\ Q) /\ R = R /\ (Q /\ R) /\ P
```

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
`CONJ_SET_CONV`.

CONJUNCTS_THEN

```
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

CONJUNCT1, CONJUNCT2, CONJUNCTS, CONJ_TAC, CONJUNCTS_THEN2, STRIP_THM_THEN.

CONJUNCTS_THEN2

```
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

`CONJUNCT1, CONJUNCT2, CONJUNCTS, CONJ_TAC, CONJUNCTS_THEN2, STRIP_THM_THEN.`

CONJ_DISCH

```
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
`CONJ_DISCHL`.

CONJ_DISCHL

```
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
`CONJ_DISCH`.

CONJ_LIST

```
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;;
   evaluation failed     CONJ_LIST

   #CONJ_LIST 1 th;;
   [A |- (x /\ y) /\ z /\ w] : thm list

   #CONJ_LIST 2 th;;
   [A |- x /\ y; A |- z /\ w] : thm list

   #CONJ_LIST 3 th;;
   [A |- x /\ y; A |- z; A |- w] : thm list

   #CONJ_LIST 4 th;;
   evaluation failed     CONJ_LIST
```

See also
`BODY_CONJUNCTS`, `LIST_CONJ`, `CONJUNCTS`, `CONJ`, `CONJUNCT1`, `CONJUNCT2`, `CONJ_PAIR`.

CONJ_PAIR

```
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
BODY_CONJUNCTS, CONJUNCT1, CONJUNCT2, CONJ, LIST_CONJ, CONJ_LIST, CONJUNCTS.

CONJ_SET_CONV

```
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 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
CONJUNCTS_CONV.

CONJ_TAC

```
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
`STRIP_TAC`.

constants

```
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

```
#constants `combin`;;
["I"; "S"; "K"; "$o"] : term list
```

See also
`axioms, binders, definitions, infixes, theorems`

CONTR

```
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

`CCONTR`, `CONTRAPOS`, `CONTR_TAC`, `NOT_ELIM`.

CONTRAPOS

```
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
CCONTR, CONTR, CONTRAPOS_CONV, NOT_ELIM.

CONTRAPOS_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
CONTRAPOS.

CONTR_TAC

```
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
CHECK_ASSUME_TAC, CONTR, CCONTR, CONTRAPOS, NOT_ELIM.

CONV_RULE

```
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
CONV_TAC, RIGHT_CONV_RULE.

CONV_TAC

```
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_RULE`.

current_theory

```
current_theory : (void -> string)
```

Synopsis
Returns the name of the current theory.

Description
Within a HOL session there is always a current theory. It is the theory represented by the current theory segment together with its ancestry. A call of `current_theory()` returns the name of the current theory. Initially HOL has current theory `scratch`.

Failure
Never fails.

See also
`export_theory, new_theory, print_theory.`

current_trace

```
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
`register_trace, reset_trace, reset_traces, trace, traces.`

curry

```
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 `\x y. f(x,y)`, so that

```
   curry f x y = f(x,y)
```

Failure

Never fails.

Example

```
- val increment = curry op+ 1;
> val it = increment = fn : int -> int

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

See also

`uncurry`.

define_new_type_bijections

```
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 to `define_new_type_bijections{name=s1,ABS=s2,REP=s3,tyax=th}` fails if `th` is not a theorem of the form returned by `new_type_definition`, or if either `s2` or `s3` is already the name of a constant in the current theory, or there already exists a constant definition, constant specification, type definition or axiom named `s1` in the current theory, or HOL is not in draft mode.

See also

`new_type_definition`, `prove_abs_fn_one_one`, `prove_abs_fn_onto`,
`prove_rep_fn_one_one`, `prove_rep_fn_onto`.

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

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 HOL is not in draft mode; if `op` is already the name of a type constant or type operator in the current theory; 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; if there is already a theorem stored under the name `` `name` `` 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.

See also

`INDUCT_THEN`, `new_recursive_definition`, `prove_cases_thm`, `prove_constructors_distinct`, `prove_constructors_one_one`, `prove_induction_thm`, `prove_rec_fn_exists`.

DEF_EXISTS_RULE

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

Synopsis

Proves that a function defined by a definitional equation exists.

Description

This rule accepts a term of the form `"c = ..."` or `"f x1 ... xn = ..."`, the variables of which may be universally quantified, and returns an existential theorem. The resulting theorem is typically used for generating HOL specifications.

Failure

`DEF_EXISTS_RULE` fails if the definition is not an equation, if there is any variable in the right-hand side which does not occur in the left-hand side, if the definition is recursive, if there is a free type variable, or if the name being defined by the function is not allowed.

Example

The effect of this rule can be understood more clearly through an example:

```
#DEF_EXISTS_RULE "max a b = ((a < b) => b | a)" ;;
|- ?max. !a b. max a b = (a < b => b | a)
```

Comments

In later versions of HOL this function may be made internal.

See also

`new_definition`, `new_gen_definition`, `new_specification`.

delete_restriction

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

Synopsis

Removes a restriction semantics from a binder.

Description

Recall that 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 ----
```

This behaviour may be disabled by calling `delete_restriction` with the binder name ("B" in this example).

Failure

Fails if you attempt to remove one of the builtin restrictions. These are associated with the binders

```
["!","?","@","\\"]
```

Also fails if the named binder is not restricted, i.e., found as the first member of a pair on the list returned by `binder_restrictions`.

Example

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

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

    - delete_restriction "DURING";
    () : unit

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

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

See also

associate_restrictions, binder_restrictions

DEPTH_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 "(\x. (\y. y + x) 1) 2";;
|- (\x. (\y. y + x)1)2 = 1 + 2
```

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 "(\f x. (f x) + 1) (\y.y) 2";;
|- (\f x. (f x) + 1)(\y. y)2 = ((\y. y)2) + 1
```

the right-hand side of the result still contains a beta-redex, because the redex `"(\y.y)2"` is introduced by virtue an application of `BETA_CONV` higher-up in the structure of the input term. By contrast, in the example:

```
#DEPTH_CONV BETA_CONV "(\f x. (f x)) (\y.y) 2";;
|- (\f x. f x)(\y. y)2 = 2
```

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 ADD_CONV "(1 + 2) + (3 + 4 + 5)";;
|- (1 + 2) + (3 + (4 + 5)) = 15
```

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 failure string `‘QCONV‘` 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 a failure with string `‘QCONV‘`, the operation of `DEPTH_CONV` will be unpredictable.

See also

`ONCE_DEPTH_CONV`, `REDEPTH_CONV`, `TOP_DEPTH_CONV`.

dest_abs

```
dest_abs : term -> {Bvar :term, Body :term}
```

Synopsis
Breaks apart an abstraction into abstracted variable and body.

Description
`dest_abs` is a term destructor for abstractions: `dest_abs (--‘\var. t‘--)` returns Bvar = var, Body = t.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Term", origin_function = "dest_abs",
            message = "not a lambda abstraction"}
```

See also
`mk_abs, is_abs, dest_var, dest_const, dest_comb, strip_abs.`

dest_comb

```
dest_comb : term -> {Rator :term, Rand :term}
```

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

Description
`dest_comb` is a term destructor for combinations:

```
   dest_comb (--‘t1 t2‘--)
```

returns Rator = t1, Rand = t2.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Term", origin_function = "dest_comb",
            message = "not a comb"}
```

See also
`mk_comb, is_comb, dest_var, dest_const, dest_abs, strip_comb.`

dest_cond

```
dest_cond : term -> {cond :term, larm :term, rarm :term}
```

Synopsis
Breaks apart a conditional into the three terms involved.

Description
`dest_cond` is a term destructor for conditionals:

```
   dest_cond (--‘t => t1 | t2‘--)
```

returns cond = t, larm = t1, rarm = t2.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_cond",
            message = "not a cond"}
```

if term is not a conditional.

See also
`mk_cond, is_cond.`

dest_conj

```
dest_conj : term -> {conj1 :term, conj2 :term}
```

Synopsis
Term destructor for conjunctions.

Description
`dest_conj(--‘t1 /\ t2‘--)` returns conj1 = t1, conj2 = t2.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_conj",
            message = "not a conj"}
```

if term is not a conjunction.

See also
`mk_conj, is_conj.`

dest_cons

```
dest_cons : term -> {hd :term, tl :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 hd = t, tl = –‘[t1;...;tn]‘–.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_cons",
            message = "not a cons"}
```

if the term is not a non-empty list.

See also
`mk_cons, is_cons, mk_list, dest_list, is_list.`

dest_const

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

Synopsis
Breaks apart a constant into name and type.

Description
`dest_const` is a term destructor for constants:

```
   dest_const (--‘const:ty‘--)
```

returns Name = ”const”, Ty = (==‘:ty‘==).

Failure
Fails with

```
    HOL_ERR{origin_structure = "Term", origin_function = "dest_const",
            message = "not a const"}
```

See also
`mk_const, is_const, dest_var, dest_comb, dest_abs.`

dest_disj

```
dest_disj : term -> {disj1 :term, disj2 :term}
```

Synopsis
Term destructor for disjunctions.

Description
`dest_disj(--‘t1 /\ t2‘--)` returns disj1 = t1, disj2 = t2.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_disj",
            message = "not a disj"}
```

if term is not a disjunction.

See also
`mk_disj, is_disj.`

dest_eq

```
dest_eq : term -> {lhs :term, rhs :term}
```

Synopsis
Term destructor for equality.

Description
`dest_eq(--‘t1 = t2‘--)` returns lhs = t1, rhs = t2.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_eq",
            message = "not an ="}
```

See also
`mk_eq, is_eq.`

dest_exists

```
dest_exists : term -> {Bvar :term, Body :term}
```

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

Description
`dest_exists` is a term destructor for existential quantification: `dest_exists (--‘!var. t‘--)` returns Bvar = var, Body = t.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_exists",
            message = "not an exists"}
```

if term is not a existential quantification.

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

dest_forall

```
dest_forall : term -> {Bvar :term, Body :term}
```

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

Description
`dest_forall` is a term destructor for universal quantification: `dest_forall (--‘!var. t‘--)` returns Bvar = var, Body = t.

Failure
Fails with

```
   HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_forall",
           message = "not a forall"}
```

if term is not a universal quantification.

See also
`mk_forall`, `is_forall`, `strip_forall`.

dest_imp

```
dest_imp : term -> {ant :term, conseq :term}
```

Synopsis
Breaks apart an implication (or negation) into antecedent and consequent.

Description

`dest_imp` is a term destructor for implications, which treats negations as implications with consequent `F`. Thus

```
   dest_imp (--`t1 ==> t2`--)
```

returns

```
   {ant = t1, conseq = t2}
```

and also

```
   dest_imp (--`~t`--)
```

returns

```
   {ant = t, conseq = (--`F`--)}
```

Failure

Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_imp",
            message = "not an ==>"}
```

if term is neither an implication nor a negation.

Comments

Destructs negations for increased functionality of HOL-style resolution.

See also

`mk_imp, is_imp, strip_imp.`

dest_let

```
dest_let : term -> {func :term, arg :term}
```

Synopsis

Breaks apart a let-expression.

Description

`dest_let` is a term destructor for general let-expressions: `dest_let (--`LET f x`--)` returns func = f, arg = x.

Example

```
- dest_let (--`LET ($= 1) 2`--);
{func=(--`$= 1`--), arg=(--`2`--)}

- dest_let (--`let x = 2 in (x = 1)`--);
{func=(--`\x. x = 1`--), arg=(--`2`--)}
```

Failure

Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_let",
            message = "not a let term"}
```

if term is not a `let`-expression or of the more general `--`LET f x`--` form.

See also

`mk_let, is_let.`

dest_list

```
dest_list : term -> {els :term list, ty :type}
```

Synopsis

Iteratively breaks apart a list term.

Description

`dest_list` is a term destructor for lists: `dest_list (--`[t1;...;tn]:ty list`--)` returns els = [t1;...;tn], ty = ty.

Failure

Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_list",
            message = "not a list"}
```

if the term is not a list.

See also

`mk_list, is_list, mk_cons, dest_cons, is_cons.`

dest_neg

```
dest_neg : (term -> term)
```

Synopsis
Breaks apart a negation, returning its body.

Description
`dest_neg` is a term destructor for negations: `dest_neg "~t"` returns `"t"`.

Failure
Fails with `dest_neg` if term is not a negation.

See also
`mk_neg, is_neg.`

dest_pabs

```
dest_pabs : term -> {varstruct : term, body :term}
```

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

Description
`dest_pabs` is a term destructor for paired abstractions: `dest_pabs (--‘\(v1..(..)..vn). t‘--)` returns varstruct = –‘(v1..(..)..vn)‘–, body = t.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_pabs",
            message = "not a paired abstraction"}
```

unless the term is a paired abstraction.

See also
`mk_pabs, is_pabs, dest_abs, dest_var, dest_const, dest_comb.`

`dest_pair`

```
dest_pair : term -> {fst :term, snd :term}
```

Synopsis
Breaks apart a pair into two separate terms.

Description
`dest_pair` is a term destructor for pairs: `dest_pair (--‘(t1,t2)‘--)` returns fst = t1, snd = t2.

Failure
Fails with

```
   HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_pair",
           message = "not a pair"}
```

if term is not a pair.

See also
`mk_pair`, `is_pair`, `strip_pair`.

`dest_select`

```
dest_select : term -> {Bvar :term, Body :term}
```

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

Description
`dest_select` is a term destructor for choice terms:

```
   dest_select (--‘@var. t‘--)
```

returns Bvar = var, Body = t.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "dest_select",
            message = "not a @"}
```

if term is not an epsilon-term.

See also
`mk_select`, `is_select`.

dest_thm

```
dest_thm : (thm -> goal)
```

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 "p=T");;
(["p = T"], "p = T") : goal
```

See also
`concl`, `hyp`.

dest_type

```
dest_type : type -> {Tyop :string, Args :hol_type list}
```

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

Description

`dest_type(==‘:(ty1,...,tyn)op‘==)` returns
  Tyop = ”op”, Args = [ty1,...,tyn].

Example

```
- dest_type (==‘:bool‘==);
{Tyop = "bool", Args = []}

- dest_type (==‘:bool list‘==);
{Tyop = "list", Args = [==‘:bool‘==]}

- dest_type (==‘:num -> bool‘==);
{Tyop = "fun", Args = [==‘:num‘==;  ==‘:bool‘==]}
```

Failure

Fails with

```
    HOL_ERR{origin_structure = "Type", origin_function = "dest_type",
            message = ""}
```

if the type is a type variable.

See also

`mk_type, dest_vartype.`

dest_var

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

Synopsis

Breaks apart a variable into name and type.

Description

`dest_var (--‘var:ty‘--)` returns Name = ”var”, Ty = (==‘:ty‘==).

Failure

Fails with

```
    HOL_ERR{origin_structure = "Term", origin_function = "dest_var",
            message = "not a var"}
```

See also

`mk_var, is_var, dest_const, dest_comb, dest_abs.`

dest_vartype

```
dest_vartype : (type -> string)
```

Synopsis
Breaks a type variable down to its name.

Description
`dest_vartype ":*..."` returns `‘*...‘`.

Failure
Fails with `dest_vartype` if the type is not a type variable.

Example

```
#dest_vartype ":*test";;
‘*test‘ : string

#dest_vartype ":bool";;
evaluation failed     dest_vartype

#dest_vartype ":* -> bool";;
evaluation failed     dest_vartype
```

See also
`mk_vartype, is_vartype, dest_type.`

DISCARD_TAC

```
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

`POP_ASSUM, POP_ASSUM_LIST.`

disch

```
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

`filter.`

DISCH

```
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
`DISCH_ALL`, `DISCH_TAC`, `DISCH_THEN`, `FILTER_DISCH_TAC`, `FILTER_DISCH_THEN`, `NEG_DISCH`, `STRIP_TAC`, `UNDISCH`, `UNDISCH_ALL`, `UNDISCH_TAC`.

DISCH_ALL

```
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
DISCH, DISCH_TAC, DISCH_THEN, NEG_DISCH, FILTER_DISCH_TAC, FILTER_DISCH_THEN, STRIP_TAC, UNDISCH, UNDISCH_ALL, UNDISCH_TAC.

DISCH_TAC

```
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
DISCH, DISCH_ALL, DISCH_THEN, FILTER_DISCH_TAC, FILTER_DISCH_THEN, NEG_DISCH, STRIP_TAC, UNDISCH, UNDISCH_ALL, UNDISCH_TAC.

DISCH_THEN

```
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 `ASSUME`ing 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
`DISCH`, `DISCH_ALL`, `DISCH_TAC`, `NEG_DISCH`, `FILTER_DISCH_TAC`, `FILTER_DISCH_THEN`, `STRIP_TAC`, `UNDISCH`, `UNDISCH_ALL`, `UNDISCH_TAC`.

DISJ1

```
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";;
|- T \/ F
```

Comments
The system shows the type of `DISJ1` as `(thm -> conv)`.

See also
`DISJ1_TAC`, `DISJ2`, `DISJ2_TAC`, `DISJ_CASES`.

DISJ1_TAC

```
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
`DISJ1, DISJ2, DISJ2_TAC.`

DISJ2

`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;;
|- F \/ T
```

See also
`DISJ1, DISJ1_TAC, DISJ2_TAC, DISJ_CASES.`

DISJ2_TAC

`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
`DISJ1, DISJ1_TAC, DISJ2.`

disjuncts

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

Synopsis
Iteratively breaks apart a disjunction.

Description
Found in the hol88 library. `disjuncts (--‘t1 \/ ... \/ tn‘--)` returns `[(--‘t1‘--),...,(--‘tn‘-`
The argument term may be any tree of disjunctions, it need not have the form `(--‘t1 \/ (t2 \/ (`
A term that is not a disjunction is simply returned as the sole element of a list. Note that

```
   disjuncts(list_mk_disj([(--‘t1‘--),...,(--‘tn‘--)]))
```

will not return `[(--‘t1‘--),...,(--‘tn‘--)]` if any of `t1`,...,`tn` are disjunctions.

Failure
Never fails. Unless, of course, you have not loaded the hol88 library.

Example

```
- list_mk_disj [(--‘a \/ b‘--),(--‘c \/ d‘--),(--‘e \/ f‘--)];
(--‘(a \/ b) \/ (c \/ d) \/ e \/ f‘--) : term

- disjuncts it;
[(--‘a‘--),(--‘b‘--),(--‘c‘--),(--‘d‘--),(--‘e‘--),(--‘f‘--)] : term list

- list_mk_disj it;
(--‘a \/ b \/ c \/ d \/ e \/ f‘--) : term

- disjuncts (--‘1‘--);
[(--‘1‘--)] : term list
```

Comments

`disjuncts` is not in hol90. There, somewhat misleadingly, it is called `strip_disj`, in order to be consistent with all the other `strip_` routines. Because `disjuncts` splits both the left and right sides of a disjunction, this operation is not the inverse of `list_mk_disj`. It may be useful to introduce `list_dest_disj` for splitting only the right tails of a disjunction.

See also

`list_mk_disj, dest_disj.`

DISJ_CASES

```
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;;
   |- (PRE m = m) = (m = 0)
```

Comments

Neither of the ‘case’ theorems is required to have either disjunct as a hypothesis, but otherwise `DISJ_CASES` is pointless.

See also

`DISJ_CASES_TAC, DISJ_CASES_THEN, DISJ_CASES_THEN2, DISJ_CASES_UNION, DISJ1, DISJ2.`

DISJ_CASES_TAC

```
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 ([],"(P:num -> bool) m");;
  ([(["m = 0"], "P m");
    (["?n. m = SUC n"], "P m")], -) : subgoals
```

Uses
Performing a case analysis according to a disjunctive theorem.

See also
`ASSUME_TAC, ASM_CASES_TAC, COND_CASES_TAC, DISJ_CASES_THEN, STRUCT_CASES_TAC.`

DISJ_CASES_THEN

```
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

`STRIP_THM_THEN, CHOOSE_THEN, CONJUNCTS_THEN, CONJUNCTS_THEN2, DISJ_CASES_TAC, DISJ_CASES_THEN2, DISJ_CASES_THENL.`

DISJ_CASES_THEN2

```
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

`STRIP_THM_THEN`, `CHOOSE_THEN`, `CONJUNCTS_THEN`, `CONJUNCTS_THEN2`, `DISJ_CASES_THEN`, `DISJ_CASES_THENL`.

DISJ_CASES_THENL

```
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
`CHOOSE_THEN`, `CONJUNCTS_THEN`, `CONJUNCTS_THEN2`, `DISJ_CASES_THEN`, `DISJ_CASES_THEN2`, `STRIP_THM_THEN`.

DISJ_CASES_UNION

```
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;;
["0 < n"] |- (m MOD n = m) \/ (m MOD n = (m - n) MOD n)
```

See also

DISJ_CASES, DISJ_CASES_TAC, DISJ1, DISJ2.

DISJ_IMP

```
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;;
   |- ~m < n ==> n <= m
```

See also

DISJ_CASES.

e

```
e : (tactic -> void)
```

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

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

el

```
el : (int -> * list -> *)
```

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];;
7 : int
```

See also
`hd, tl.`

empty_rewrites

```
empty_rewrites: rewrites
```

Synopsis
The empty database of rewrite rules.

Description
Uses
Used

to build other rewrite sets.

See also
`base_rewrites, add_base_rewrites, add_rewrites.`

end_itlist

```
end_itlist : ((* -> * -> *) -> * list -> *)
```

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 with `end_itlist` if list is empty.

Example

```
#end_itlist (\x y. x + y) [1;2;3;4];;
10 : int
```

See also
`itlist, rev_itlist.`

EQF_ELIM

```
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
`EQF_INTRO, EQT_ELIM, EQT_INTRO.`

EQF_INTRO

```
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
`EQF_ELIM`, `EQT_ELIM`, `EQT_INTRO`.

EQT_ELIM

```
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
`EQT_INTRO`, `EQF_ELIM`, `EQF_INTRO`.

EQT_INTRO

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

Synopsis
Introduces equality with `T`.

Description

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

Failure
Never fails.

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

EQ_IMP_RULE

```
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
EQ_MP, EQ_TAC, IMP_ANTISYM_RULE.

EQ_MP

```
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
EQ_IMP_RULE, IMP_ANTISYM_RULE, MP.

EQ_TAC

```
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

`EQ_IMP_RULE, IMP_ANTISYM_RULE.`

ETA_CONV

```
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.

EVERY

```
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
`FIRST, MAP_EVERY, THEN.`

EVERY_ASSUM

```
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 `ASSUME`d assumptions of the goal, or if any of the resulting tactics fail when applied sequentially.

See also
ASSUM_LIST, MAP_EVERY, MAP_FIRST, THEN.

EVERY_CONJ_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.

Failure
Fails if the conversion argument fails when applied to any 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.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.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 conjunctive skeleton.

See also
EVERY_DISJ_CONV, RATOR_CONV, RAND_CONV, LAND_CONV

EVERY_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
`THENC`.

EVERY_DISJ_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.

Failure
Fails if the conversion argument fails when applied to any 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.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

EVERY_CONJ_CONV, RATOR_CONV, RAND_CONV, LAND_CONV.

EVERY_TCL

```
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
FIRST_TCL, ORELSE_TCL, REPEAT_TCL, THEN_TCL.

EXISTENCE

```
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
EXISTS_UNIQUE_CONV.

exists

```
exists : ((* -> bool) -> * list -> bool)
```

Synopsis
Tests a list to see if it has at least one element satisfying a predicate.

Description
`exists p l` applies `p` to the elements of `l` in order until one is found which satisfies `p`, or until the list is exhausted, returning `true` or `false` accordingly.

Failure
Never fails.

See also

`forall`, `find`, `tryfind`, `mem`, `assoc`, `rev_assoc`.

EXISTS

```
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 ("?x. x=T","T") (REFL "T");;
   |- ?x. x = T

   #EXISTS ("?x:bool. x=x","T") (REFL "T");;
   |- ?x. x = x
```

See also
`CHOOSE`, `EXISTS_TAC`.

EXISTS_AND_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
`AND_EXISTS_CONV, LEFT_AND_EXISTS_CONV, RIGHT_AND_EXISTS_CONV.`

EXISTS_EQ

```
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
`AP_TERM`, `EXISTS_IMP`, `FORALL_EQ`, `MK_EXISTS`, `SELECT_EQ`.

EXISTS_IMP

```
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
`EXISTS_EQ`.

EXISTS_IMP_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
`LEFT_IMP_FORALL_CONV, RIGHT_IMP_EXISTS_CONV.`

EXISTS_NOT_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
FORALL_NOT_CONV, NOT_EXISTS_CONV, NOT_FORALL_CONV.

EXISTS_OR_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
OR_EXISTS_CONV, LEFT_OR_EXISTS_CONV, RIGHT_OR_EXISTS_CONV.

EXISTS_TAC

```
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
EXISTS.

EXISTS_UNIQUE_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
`EXISTENCE`.

expand

```
expand : (tactic -> void)
```

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;;
OK..
evaluation failed     no goals to expand

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

() : void

#expand CONJ_TAC;;
OK..
2 subgoals
"TL[1;2;3] = [2;3]"

"HD[1;2;3] = 1"

() : void

#expand (REWRITE_TAC[HD]);;
OK..
goal proved
|- HD[1;2;3] = 1

Previous subproof:
"TL[1;2;3] = [2;3]"

() : void

#expand (REWRITE_TAC[TL]);;
OK..
goal proved
|- TL[1;2;3] = [2;3]
|- (HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])

Previous subproof:
goal proved
() : void
```

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.

```
#set_goal([],"1=2");;
"1 = 2"

() : void

#expand (REWRITE_TAC[ASSUME "1=2"]);;
OK..
evaluation failed     Invalid tactic
```

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

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

expandf

```
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

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

EXT

```
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 either of the functions `t1` or `t2`.

See also

`AP_THM, ETA_CONV, FUN_EQ_CONV.`

FAIL_TAC

```
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
`ALL_TAC`, `NO_TAC`.

filter

```
filter : ((* -> bool) -> * list -> * 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
Never fails.

See also
`mapfilter`, `partition`, `remove`.

FILTER_ASM_REWRITE_RULE

```
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
`ASM_REWRITE_RULE`, `FILTER_ONCE_ASM_REWRITE_RULE`, `FILTER_PURE_ASM_REWRITE_RULE`, `FILTER_PURE_ONCE_ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWRITE_RULE`.

FILTER_ASM_REWRITE_TAC

```
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
`ASM_REWRITE_TAC`, `FILTER_ONCE_ASM_REWRITE_TAC`, `FILTER_PURE_ASM_REWRITE_TAC`, `FILTER_PURE_ONCE_ASM_REWRITE_TAC`, `GEN_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`.

FILTER_DISCH_TAC

```
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

DISCH, DISCH_ALL, DISCH_TAC, DISCH_THEN, FILTER_DISCH_THEN, NEG_DISCH, STRIP_TAC, UNDISCH, UNDISCH_ALL, UNDISCH_TAC.

FILTER_DISCH_THEN

```
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
DISCH, DISCH_ALL, DISCH_TAC, DISCH_THEN, FILTER_DISCH_TAC, NEG_DISCH, STRIP_TAC, UNDISCH, UNDISCH_ALL, UNDISCH_TAC.

FILTER_GEN_TAC

```
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
`GEN, GEN_TAC, GENL, GEN_ALL, SPEC, SPECL, SPEC_ALL, SPEC_TAC, STRIP_TAC.`

FILTER_ONCE_ASM_REWRITE_RULE

```
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
ASM_REWRITE_RULE, FILTER_ASM_REWRITE_RULE, FILTER_PURE_ASM_REWRITE_RULE, FILTER_PURE_ONCE_ASM_REWRITE_RULE, GEN_REWRITE_RULE, ONCE_ASM_REWRITE_RULE, ONCE_DEPTH_CONV, PURE_ASM_REWRITE_RULE, PURE_ONCE_ASM_REWRITE_RULE, PURE_REWRITE_RULE, REWRITE_RULE.

FILTER_ONCE_ASM_REWRITE_TAC

```
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
ASM_REWRITE_TAC, FILTER_ASM_REWRITE_TAC, FILTER_PURE_ASM_REWRITE_TAC, FILTER_PURE_ONCE_ASM_REWRITE_TAC, GEN_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_DEPTH_CONV, PURE_ASM_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_REWRITE_TAC, REWRITE_TAC.

FILTER_PURE_ASM_REWRITE_RULE

```
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
`ASM_REWRITE_RULE`, `FILTER_ASM_REWRITE_RULE`, `FILTER_ONCE_ASM_REWRITE_RULE`, `FILTER_PURE_ONCE_ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWRITE_RULE`.

FILTER_PURE_ASM_REWRITE_TAC

```
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
`ASM_REWRITE_TAC`, `FILTER_ASM_REWRITE_TAC`, `FILTER_ONCE_ASM_REWRITE_TAC`, `FILTER_PURE_ONCE_ASM_REWRITE_TAC`, `GEN_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`.

FILTER_PURE_ONCE_ASM_REWRITE_RULE

```
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
`ASM_REWRITE_RULE`, `FILTER_ASM_REWRITE_RULE`, `FILTER_ONCE_ASM_REWRITE_RULE`, `FILTER_PURE_ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_ASM_REWRITE_RULE`, `ONCE_DEPTH_CONV`, `PURE_ASM_REWRITE_RULE`, `PURE_ONCE_ASM_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWRITE_RULE`.

FILTER_PURE_ONCE_ASM_REWRITE_TAC

```
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

ASM_REWRITE_TAC, FILTER_ASM_REWRITE_TAC, FILTER_ONCE_ASM_REWRITE_TAC, FILTER_PURE_ASM_REWRITE_TAC, GEN_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_DEPTH_CONV, PURE_ASM_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_REWRITE_TAC, REWRITE_TAC.

FILTER_STRIP_TAC

```
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

CONJ_TAC, FILTER_DISCH_TAC, FILTER_DISCH_THEN, FILTER_GEN_TAC, STRIP_ASSUME_TAC, STRIP_TAC.

FILTER_STRIP_THEN

```
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
`CONJ_TAC, FILTER_DISCH_TAC, FILTER_DISCH_THEN, FILTER_GEN_TAC, FILTER_STRIP_TAC, STRIP_ASSUME_TAC, STRIP_GOAL_THEN.`

find

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

Synopsis
Returns the first element of a list which satisfies a predicate.

Description
Found in the hol88 library. `find p [x1;...;xn]` returns the first `xi` in the list such that `(p xi)` is `true`.

Failure
Fails with `find` if no element satisfies the predicate. This will always be the case if the list is empty.

Comments
`find` is in Compat, because is is not found in hol90 (`Lib.first` is equivalent and is used instead).

See also
`tryfind, mem, exists, forall, assoc, rev_assoc.`

FIRST

```
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
`EVERY, ORELSE.`

FIRST_ASSUM

```
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
`ASSUM_LIST, EVERY, EVERY_ASSUM, FIRST, MAP_EVERY, MAP_FIRST.`

FIRST_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
`ORELSEC`.

FIRST_TCL

```
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
`EVERY_TCL`, `ORELSE_TCL`, `REPEAT_TCL`, `THEN_TCL`.

FIRST_X_ASSUM

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

FORALL_AND_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
`AND_FORALL_CONV`, `LEFT_AND_FORALL_CONV`, `RIGHT_AND_FORALL_CONV`.

FORALL_EQ

```
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
`AP_TERM`, `EXISTS_EQ`, `SELECT_EQ`.

FORALL_IMP_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
`LEFT_IMP_EXISTS_CONV`, `RIGHT_IMP_FORALL_CONV`.

FORALL_NOT_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
`EXISTS_NOT_CONV, NOT_EXISTS_CONV, NOT_FORALL_CONV.`

FORALL_OR_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
`OR_FORALL_CONV, LEFT_OR_FORALL_CONV, RIGHT_OR_FORALL_CONV.`

FORK_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.thm
```

See also
BINOP_CONV, LAND_CONV, RAND_CONV, RATOR_CONV.

frees

```
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
`freesl, free_in, thm_frees.`

freesl

```
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
`frees, free_in, thm_frees.`

FREEZE_THEN

```
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
`ASSUME, IMP_RES_TAC, PROVE_HYP, RES_TAC, REWR_CONV.`

free_in

```
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.

Failure
Never fails.

Example
In the following example `free_in` returns `false` because the `x` in `SUC x` in the second term is bound:

```
   #free_in "SUC x" "!x. SUC x = x + 1";;
   false : bool
```

whereas the following call returns `true` because the first instance of `x` in the second term is free, even though there is also a bound instance:

```
   #free_in "x:bool" "x /\ (?x. x=T)";;
   true : bool
```

See also
`frees, freesl, thm_frees.`

FRONT_CONJ_CONV

```
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. The system shows its type as `(term list -> conv)`.

front_last

```
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
```

fst

```
fst : ((* # **) -> *)
```

Synopsis
Extracts the first component of a pair.

Description
`fst (x,y)` returns `x`.

Failure
Never fails.

See also
`snd`, `pair`.

FULL_SIMP_TAC

```
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 it 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 hol_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
`ASM_SIMP_TAC, hol_ss, SIMP_CONV, SIMP_RULE, SIMP_TAC.`

funpow

```
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`. `funpow 0 f x` returns `x`. If `n` is negative, `funpow n f x` returns `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
```

FUN_EQ_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

`EXT, X_FUN_EQ_CONV.`

g

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

GEN

```
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.

```
#top_print print_all_thm;;
- : (thm -> void)

#let t = ASSUME "x=T";;
t = x = T |- x = T

#GEN "x:bool" t;;
evaluation failed     GEN
```

See also
`GENL`, `GEN_ALL`, `GEN_TAC`, `SPEC`, `SPECL`, `SPEC_ALL`, `SPEC_TAC`.

GENL

```
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

`GEN, GEN_ALL, GEN_TAC, SPEC, SPECL, SPEC_ALL, SPEC_TAC.`

genvar

```
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";;
"GEN%VAR%357" : term

#genvar ":num";;
"GEN%VAR%358" : term
```

Trying to anticipate `genvar` doesn't work:

```
#let v = mk_var(`GEN%VAR%359`,":bool");;
v = "GEN%VAR%359" : term

#genvar ":bool";;
"GEN%VAR%360" : term
```

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
`GSPEC, variant.`

GEN_ALL

```
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

GEN, GENL, GEN_ALL, SPEC, SPECL, SPEC_ALL, SPEC_TAC.

GEN_ALPHA_CONV

```
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". The function is_binder determines what is regarded as a binder in this context.

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)  = (!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 (that is, is_binder `B` returns true). 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
ALPHA, ALPHA_CONV, is_binder.

GEN_BETA_CONV

```
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:

```
#GEN_BETA_CONV "(\(x,y). x + y) (1,2)";;
|- (\(x,y). x + y)(1,2) = 1 + 2
```

whereas in the following, the operand of the beta-redex is not a pair, so FST and SND are introduced:

```
#GEN_BETA_CONV "(\(x,y). x + y) numpair";;
|- (\(x,y). x + y)numpair = (FST numpair) + (SND numpair)
```

The introduction of FST and SND will be done more than once as necessary:

```
#GEN_BETA_CONV "(\(w,x,y,z). w + x + y + z) (1,triple)";;
|- (\(w,x,y,z). w + (x + (y + z)))(1,triple) =
   1 + ((FST triple) + ((FST(SND triple)) + (SND(SND triple))))
```

See also
BETA_CONV, 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
`ASM_MESON_TAC`, `MESON_TAC`

GEN_REWRITE_CONV

```
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

ONCE_REWRITE_CONV, PURE_REWRITE_CONV, REWR_CONV, REWRITE_CONV.

GEN_REWRITE_RULE

```
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
`ASM_REWRITE_RULE`, `FILTER_ASM_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWR_CONV`, `REWRITE_RULE`.

GEN_REWRITE_TAC

```
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

`ASM_REWRITE_TAC`, `GEN_REWRITE_RULE`, `ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWR_CONV`, `REWRITE_TAC`,

GEN_TAC

```
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
`FILTER_GEN_TAC`, `GEN`, `GENL`, `GEN_ALL`, `SPEC`, `SPECL`, `SPEC_ALL`, `SPEC_TAC`, `STRIP_TAC`, `X_GEN_TAC`.

GSPEC

```
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
GEN, GENL, genvar, GEN_ALL, GEN_TAC, SPEC, SPECL, SPEC_ALL, SPEC_TAC.

GSUBST_TAC

```
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

`SUBST1_TAC, SUBST_OCCS_TAC, SUBST_TAC.`

GSYM

```
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

```
#ADD;;
|- (!n. 0 + n = n) /\ (!m n. (SUC m) + n = SUC(m + n))
Run time: 0.0s

#GSYM ADD;;
|- (!n. n = 0 + n) /\ (!m n. SUC(m + n) = (SUC m) + n)
```

See also

`NOT_EQ_SYM, REFL, SYM.`

HALF_MK_ABS

```
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
`ETA_CONV`, `MK_ABS`, `MK_COMB`, `MK_EXISTS`.

hidden

```
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 has been hidden, via 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
`hide`, `reveal`.

hide

```
hide : string -> unit
```

Synopsis
Stops the quotation parser from recognizing a constant.

Description
A call `hide "c"` where `c` is the name of a constant, will prevent the quotation parser from parsing it as such; it will just be parsed as a variable. The effect can be reversed by `Parse.reveal "c"`. However, this call does not affect the treatment of overloaded constants. If `"+"` is overloaded (to addition over the naturals and integers, for example), then `hide "+"` will not affect its parsing. This must be achieved by first removing the overloading, using `clear_overloads_on`.

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
`hidden`, `known_constants`, `reveal`, `set_known_constants`.

hol_ss

```
HOLSimps.hol_ss : simpset
```

Synopsis
The most powerful simpset provided by the HOL system.

Description
The `hol_ss` simpset includes simplifications appropriate for use with the theories of pairs, sums, options, lists, and numbers. It includes an arithmetic decision procedure for linear arithmetic over the natural numbers (`ARITH_CONV`) and a variety of other powerful techniques. The way in which these components are applied to terms is described in the entry for `SIMP_CONV`.

Failure
Can't fail as it is not a functional value.

Example

```
- SIMP_CONV hol_ss []
    (Term‘P (2 * 2) /\ (P 4 ==> (x = y + 3)) ==> P x /\ y < x‘);
> val it =
    |- P (2 * 2) /\ (P 4 ==> (x = y + 3)) ==> P x /\ y < x =
       P 4 /\ (P 4 ==> (x = y + 3)) ==> P (y + 3)
    : thm
```

Comments
It can be very difficult to predict what simplification will manage to do to one's terms.

See also
`++, ASM_SIMP_TAC, bool_ss, FULL_SIMP_TAC, pure_ss, SIMP_CONV, SIMP_TAC.`

hyp

```
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
`dest_thm`, `concl`.

hyp_union

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

Synopsis
Returns union of assumption lists of the given theorems.

Description
When applied to a list of theorems, `hyp_union` returns the union (see `union`) of their assumption lists. Straight repetitions only arise if there were multiple instances of an assumption in a single assumption list. There is no elimination of alpha-equivalent pairs of assumptions, only ones which are actually equal.

```
   hyp_union [A1 |- t1; ... ; An |- tn] = A1 u...u An
```

Failure
Never fails.

Uses
Designed for internal use, in writing primitive inference rules.

See also
`union`.

I

```
I : (* -> *)
```

Synopsis
Performs identity operation: `I x` = `x`.

Failure
Never fails.

See also
`#, B, C, CB, Co, K, KI, o, oo, S, W.`

IMP_ANTISYM_RULE

```
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
`EQ_IMP_RULE, EQ_MP, EQ_TAC.`

IMP_CANON

```
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
`CONJ1, CONJ2, CONJUNCTS, DISJ1, DISJ2, EXISTS, SPEC.`

IMP_CONJ

```
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
CONJ.

IMP_ELIM

```
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
NOT_INTRO, NOT_ELIM.

IMP_RES_TAC

```
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
`IMP_RES_THEN, RES_CANON, RES_TAC, RES_THEN.`

IMP_RES_THEN

```
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 with '`no implication`' 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 with '`no resolvents`' if no assumption of the goal `A ?- g` can be resolved with the implication or implications derived from `th`. Evaluation also fails, with '`no tactics`', 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
`IMP_RES_TAC`, `MATCH_MP`, `RES_CANON`, `RES_TAC`, `RES_THEN`.

IMP_TRANS

```
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

IMP_ANTISYM_RULE, SYM, TRANS.

Induct

```
bossLib.Induct : tactic
```

Synopsis

Performs tactical proof by induction over the type of the goal's outermost universally quantified variable.

Description

Given a universally quantified goal, `Induct` attempts to perform an induction on the variable that is universally quantified. The induction theorem to be used is looked up in the `TypeBase` database of theorems about the system's defined 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 (as necessarily happens, for example, with all variable types).

Example

If attempting to prove that

```
   ``!list. LENGTH (REVERSE list) = LENGTH list``
```

one can begin the proof by doing an induction on the list, thus:

```
   - Induct ([], ``!list. LENGTH (REVERSE list) = LENGTH list``);
   > val it =
        ([([], `LENGTH (REVERSE []) = LENGTH []`),
          ([`LENGTH (REVERSE list) = LENGTH list`],
           `!h. LENGTH (REVERSE (CONS h list)) =
                  LENGTH (CONS h list)`)],
         fn)
     : goal list * validation
```

where the two subgoals in the list above are the base case and step case respectively of the induction theorem for lists.

The same tactic can be used for induction over numbers, thus:

```
- Induct ([], ``!n. n > 2 ==>
                  !x y z. ~(x EXP n + y EXP n = z EXP n)``);
> val it =
     ([([], `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)`)],
      fn)
  : goal list * validation
```

See also

Induct_on, completeInduct_on, measureInduct_on

INDUCT

```
INDUCT : ((thm # thm) -> thm)
```

Synopsis

Performs a proof by mathematical induction on the natural numbers.

Description

The derived inference rule `INDUCT` implements the rule of mathematical induction:

```
   A1 |- P[0]      A2 |- !n. P[n] ==> P[SUC n]
-----------------------------------------------  INDUCT
           A1 u A2 |- !n. P[n]
```

When supplied with a theorem `A1 |- P[0]`, which asserts the base case of a proof of the proposition `P[n]` by induction on `n`, and the theorem `A2 |- !n. P[n] ==> P[SUC n]`, which asserts the step case in the induction on `n`, the inference rule `INDUCT` returns `A1 u A2 |- !n. P[n]`.

Failure

`INDUCT th1 th2` fails if the theorems `th1` and `th2` do not have the forms `A1 |- P[0]` and `A2 |- !n. P[n] ==> P[SUC n]` respectively.

See also

INDUCT_TAC.

INDUCT_TAC

```
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

```
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 `i`th constructor to a sequence of variables, then if `ttac` is a function that maps the induction hypotheses `hypi` of the `i`th subgoal to the tactic:

```
      A  ?- P[Ci(vs)/x]
   ======================  MAP_EVERY ttac hypi
         A1 ?- Gi
```

then `INDUCT_THEN th ttac` is an induction tactic that decomposes the goal `A ?- !x.P` into a set of `n` subgoals, one for each constructor, as follows:

```
            A ?- !x.P
  ================================  INDUCT_THEN th ttac
     A1 ?- G1  ...   An ?- Gn
```

The resulting subgoals correspond to the cases in a structural induction on the variable `x` of type `ty`, with induction hypotheses treated as determined by `ttac`.

Failure

`INDUCT_THEN th ttac g` fails if `th` is not a structural induction theorem of the form returned by `prove_induction_thm`, or if the goal does not have the form `A ?- !x:ty.P` where `ty` is the type for which `th` is the induction theorem, or if `ttac` fails for any subgoal in the induction.

Example

The built-in structural induction theorem for lists is:

```
   |- !P. P[] /\ (!t. P t ==> (!h. P(CONS h t))) ==> (!l. P l)
```

When `INDUCT_THEN` is applied to this theorem, it constructs and returns a specialized induction tactic (parameterized by a theorem-tactic) for doing induction on lists:

```
   #let LIST_INDUCT_THEN = INDUCT_THEN list_INDUCT;;
   LIST_INDUCT_THEN = - : (thm_tactic -> tactic)
```

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

`define_type`, `new_recursive_definition`, `prove_cases_thm`, `prove_constructors_distinct`, `prove_constructors_one_one`, `prove_induction_thm`, `prove_rec_fn_exists`.

initial_rws

```
initial_rws : unit -> computeLib.comp_rws
```

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 (initial_rws()) (--‘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.thm
```

See also

CBV_CONV, REDUCE_CONV

inst

```
inst : hol_type subst -> term -> term
```

Synopsis

Performs type instantiations in a term. NOT the same as the hol88 inst; the first argument (the ”away-from” list) used in hol88 inst is unnecessary and hence dispensed with, PLUS hol90 insists that all redexes be type variables.

Description

The function inst should be used as follows:

```
inst [{redex_1, residue_1},...,{redex_n, residue_n}] tm
```

where the redexes are all hol_type variables, and the residues all hol_types 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

Fails if there exists a redex in the substition that is not a type variable.

Example

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

- let val tm = --‘(x:’a) = (x:’a)‘--
  in inst [{redex = ==‘:’a‘==, residue = ==‘:num‘==}] tm
  end;
> val it = (--‘(x :num) = (x :num)‘--) : term

- inst [{redex = ==‘:bool‘==, residue = ==‘:num‘==}] (--‘x:bool‘--)
  handle e => Raise e;
Exception raised at Term.inst:
redex in type substitution not a variable

- let val x = --‘x:bool‘--
  in inst [{redex = ==‘:’a‘==, residue = ==‘:bool‘==}]
          (--‘\x:’a. ^x‘--)
  end;
(--‘\(x’ :bool). (x :bool)‘--) : term
```

Uses

Performing internal functions connected with type instantiation.

See also

type_subst, Compat.inst_type, INST_TYPE.

INST

```
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 |- t[t1,...,tn/x1,...,xn]
```

where the variables x1, ..., xn are not free in the assumptions A.

Failure

INST fails if a variable being instantiated is free in the assumptions.

Example

In the following example a theorem is instantiated for a specific term:

```
- load"arithmeticTheory";

- CONJUNCT1 arithmeticTheory.ADD_CLAUSES;
|- 0 + m = m

- INST [``m:num`` |-> ``2*x``]
       (CONJUNCT1 arithmeticTheory.ADD_CLAUSES);

val it = |- 0 + (2 * x) = 2 * x : thm
```

See also

INST_TY_TERM, INST_TYPE, ISPEC, ISPECL, SPEC, SPECL, SUBS, subst, SUBST.

INST_TYPE

```
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 |- t[ty1,...,tyn/vty1,...,vtyn]
```

where none of the types vtyi are free in the assumption list. Variables will be renamed if necessary to prevent distinct variables becoming identical after the instantiation.

Failure

INST_TYPE fails if any of the type variables occurs free in the hypotheses of the theorem, or if upon instantiation two distinct variables (with the same name) become equal.

Uses

INST_TYPE is employed to make use of polymorphic theorems.

Example

Suppose 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";
   > val it = () : unit

   - SPECL [(--‘a:num‘--), (--‘b:num‘--)]
           (INST_TYPE [‘‘:’a‘‘ |-> ‘‘:num‘‘] EQ_SYM_EQ);

   > val it = |- (a = b) = (b = a) : Thm.thm
```

See also

INST, INST_TY_TERM.

INST_TY_TERM

```
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
`INST, INST_TYPE, ISPEC, SPEC, SUBS, SUBST.`

intersect

```
intersect : (* list -> * list -> * 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];;
[1; 3] : int list

#intersect [1;2;4;1] [1;2;3;2];;
[1; 2; 1] : int list
```

See also
`setify, set_equal, union, subtract.`

int_of_string

```
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`, `string_of_int`, `int_to_string`, `string_to_int`.

ISPEC

```
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
`INST_TY_TERM`, `INST_TYPE`, `ISPECL`, `SPEC`, `match_term`.

ISPECL

```
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
`INST_TYPE`, `INST_TY_TERM`, `ISPEC`, `MATCH`, `SPEC`, `SPECL`.

is_abs

```
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
`mk_abs, dest_abs, is_var, is_const, is_comb.`

is_axiom

```
is_axiom : ((string # string) -> bool)
```

Synopsis
Tests if there is an axiom with the given name in the given theory.

Description
The call `is_axiom(‘th‘,‘ax‘)`, where `th` is the name of a theory (as usual `‘-‘` means the current theory), tests if there is an axiom called `ax` in that theory.

Failure
Fails unless the given theory is an ancestor.

Example
```
#is_axiom(‘bool‘,‘BOOL_CASES_AX‘);;
true : bool

#is_axiom(‘bool‘,‘INFINITY_AX‘);;
false : bool

#is_axiom(‘ind‘,‘INFINITY_AX‘);;
true : bool
```

See also
`axioms, new_axiom.`

is_binder

```
is_binder : (string -> bool)
```

Synopsis
Determines whether a given string represents a binder.

Description
This predicate returns true if the given string argument is the name of a binder: it returns false otherwise.

Example
```
#binders ‘bool‘;;
["$?!"; "$!"; "$@"] : term list

#is_binder ‘$?!‘;;
false : bool

#is_binder ‘?!‘;;
true : bool
```

See also
`binders, is_binder_type, is_infix, is_constant`

is_comb

```
is_comb : (term -> bool)
```

Synopsis
Tests a term to see if it is a combination (function application).

Description
`is_comb "t1 t2"` returns `true`. If the term is not a combination the result is `false`.

Failure
Never fails

See also
`mk_comb, dest_comb, is_var, is_const, is_abs.`

is_cond

```
is_cond : (term -> bool)
```

Synopsis
Tests a term to see if it is a conditional.

Description
`is_cond "t => t1 | t2"` returns `true`. If the term is not a conditional the result is `false`.

Failure
Never fails.

See also
`mk_cond, dest_cond.`

is_conj

```
is_conj : (term -> bool)
```

Synopsis
Tests a term to see if it is a conjunction.

Description
`is_conj "t1 /\ t2"` returns `true`. If the term is not a conjunction the result is `false`.

Failure
Never fails.

See also
`mk_conj, dest_conj.`

is_cons

```
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
`mk_cons`, `dest_cons`, `mk_list`, `dest_list`, `is_list`.

is_const

```
is_const : (term -> bool)
```

Synopsis
Tests a term to see if it is a constant.

Description
`is_const "const:ty"` returns `true`. If the term is not a constant the result is `false`.

Failure
Never fails.

See also
`mk_const`, `dest_const`, `is_var`, `is_comb`, `is_abs`.

is_constant

```
is_constant : (string -> bool)
```

Synopsis
Determines whether a string is the name of a constant.

Description
This predicate returns `true` if the given string argument is the name of a constant defined in the current theory or its ancestors: it returns `false` otherwise.

Example

```
#is_constant `SUC`;;
true : bool

#is_constant `3`;;
true : bool

#is_constant `$!`;;
false : bool

#is_constant `!`;;
true : bool

#is_constant `xx`;;
false : bool
```

See also
`is_infix, is_binder`

is_disj

```
is_disj : (term -> bool)
```

Synopsis
Tests a term to see if it is a disjunction.

Description
`is_disj "t1 \/ t2"` returns `true`. If the term is not a disjunction the result is `false`.

Failure
Never fails.

See also
`mk_disj, dest_disj.`

is_eq

```
is_eq : (term -> bool)
```

Synopsis
Tests a term to see if it is an equation.

Description
`is_eq "t1 = t2"` returns `true`. If the term is not an equation the result is `false`.

Failure
Never fails.

See also
`mk_eq`, `dest_eq`.

is_exists

```
is_exists : (term -> bool)
```

Synopsis
Tests a term to see if it as an existential quantification.

Description
`is_exists "?var. t"` returns `true`. If the term is not an existential quantification the result is `false`.

Failure
Never fails.

See also
`mk_exists`, `dest_exists`.

is_forall

```
is_forall : (term -> bool)
```

Synopsis
Tests a term to see if it is a universal quantification.

Description

`is_forall "!var. t"` returns `true`. If the term is not a universal quantification the result is `false`.

Failure

Never fails.

See also

`mk_forall, dest_forall.`

is_hidden

```
is_hidden : (string -> bool)
```

Synopsis

Determines whether a constant is hidden.

Description

This predicate returns `true` if the named `ML` constant has been hidden by the function `hide_constant`; it returns `false` if the constant is not hidden. Hiding a constant forces the quotation parser to treat the constant as a variable (lexical rules permitting).

Example

```
#is_hidden `0`;;
false : bool

#hide_constant `0`;;
() : void

#is_hidden `0`;;
true : bool

#unhide_constant `0`;;
() : void

#is_hidden `0`;;
false : bool
```

See also

`hide_constant, unhide_constant`

is_imp

```
is_imp : (term -> bool)
```

Synopsis
Tests a term to see if it is an implication (or a negation).

Description
`is_imp "t1 ==> t2"` returns `true`. `is_imp "~t"` 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 compatibility with PPLAMBDA code).

See also
`mk_imp`, `dest_imp`.

is_infix

```
is_infix : (string -> bool)
```

Synopsis
Determines whether an operator is infix.

Description
This predicate returns `true` if the given string argument is the name of an infix operator (a constant); it returns `false` otherwise.

Example

```
#is_infix `$+`;;
false : bool

#is_infix `+`;;
true : bool

#is_infix `SUC`;;
false : bool
```

See also

`infixes, is_binder, is_constant.`

is_let

```
is_let : (term -> bool)
```

Synopsis

Tests a term to see if it is a `let`-expression.

Description

`is_let "LET f x"` returns `true`. If the term is not a `let`-expression (or of the more general `"LET f x"` form) the result is `false`.

Failure

Never fails.

Example

```
#is_let "LET ($= 1) 2";;
true : bool

#is_let "let x = 2 in (x = 1)";;
true : bool
```

See also

`mk_let, dest_let.`

is_list

```
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
`mk_list`, `dest_list`, `mk_cons`, `dest_cons`, `is_cons`.

is_neg

```
is_neg : (term -> bool)
```

Synopsis
Tests a term to see if it is a negation.

Description
`is_neg "~t"` returns `true`. If the term is not a negation the result is `false`.

Failure
Never fails.

See also
`mk_neg`, `dest_neg`.

is_pabs

```
is_pabs : (term -> bool)
```

Synopsis
Tests a term to see if it is a paired abstraction.

Description
`is_pabs "\(v1..(..)..vn). t"` returns `true`. If the term is not a paired abstraction the result is `false`.

Failure
Never fails.

See also
`mk_pabs, dest_pabs, is_abs, is_var, is_const, is_comb.`

is_pair

```
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
`mk_pair, dest_pair.`

is_select

```
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
`mk_select`, `dest_select`.

is_type

```
is_type : (string -> bool)
```

Synopsis
Tests whether a string is the name of a type.

Description
`is_type ‘op‘` returns `true` if `‘op’` is the name of a type or type operator and `false` otherwise.

Failure
Never fails.

See also
`arity`.

is_var

```
is_var : (term -> bool)
```

Synopsis
Tests a term to see if it is a variable.

Description
`is_var "var:ty"` returns `true`. If the term is not a variable the result is `false`.

Failure
Never fails.

See also
`mk_var, dest_var, is_const, is_comb, is_abs.`

is_vartype

```
is_vartype : (type -> bool)
```

Synopsis
Tests a type to see if it is a type variable.

Description
`is_vartype(":*...")` returns `true`. For types which are not type variables it returns `false`.

Failure
Never fails.

Example

```
#is_vartype ":*test";;
true : bool

#is_vartype ":bool";;
false : bool

#is_vartype ":* -> bool";;
false : bool
```

See also
`mk_vartype, dest_vartype.`

itlist

```
itlist : ((* -> ** -> **) -> * list -> ** -> **)
```

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

Description
`itlist f [x1;...;xn] y` returns

```
   f x1 (f x2 ... (f xn y)...)
```

It returns `y` if list is empty.

Failure
Never fails.

Example

```
#itlist (\x y. x + y) [1;2;3;4] 0;;
10 : int
```

See also
`rev_itlist`, `end_itlist`.

itlist2

```
itlist2 : (((* # **) -> *** -> ***) -> (* list # ** list) -> *** -> ***)
```

Synopsis
Applies a paired function between adjacent elements of 2 lists.

Description
`itlist2 f ([x1;...;xn],[y1;...;yn]) z` returns

```
   f (x1,y1) (f (x2,y2) ... (f (xn,yn) z)...)
```

It returns `z` if both lists are empty.

Failure
Fails with `itlist2` if the two lists are of different lengths.

Example

```
#itlist2 (\(x,y) z. (x * y) + z) ([1;2],[3;4]) 0;;
11 : int
```

See also
`itlist`, `rev_itlist`, `end_itlist`, `uncurry`.

K

```
K : (* -> ** -> *)
```

Synopsis
Forms a constant function: `(K x) y` = `x`.

Failure
Never fails.

See also
`#`, `B`, `C`, `CB`, `Co`, `I`, `KI`, `o`, `oo`, `S`, `W`.

known_constants

```
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
hide, reveal, set_known_constants.

LAND_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.thm
```

See also
ABS_CONV, BINOP_CONV, RAND_CONV, RATOR_CONV.

last

```
Compat.last : 'a list -> 'a
```

Synopsis
Computes the last element of a list.

Description
`last [x1,...,xn]` returns `xn`.

Failure
Found in the hol88 library. Fails with `last` if the list is empty. It will not be found unless the hol88 library has been loaded.

Comments
Not in hol90, since it was never used in the implementation.

See also
`butlast, hd, tl, el, null.`

LEFT_AND_EXISTS_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
`AND_EXISTS_CONV, EXISTS_AND_CONV, RIGHT_AND_EXISTS_CONV.`

LEFT_AND_FORALL_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
`AND_FORALL_CONV`, `FORALL_AND_CONV`, `RIGHT_AND_FORALL_CONV`.

LEFT_IMP_EXISTS_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
`FORALL_IMP_CONV`, `RIGHT_IMP_FORALL_CONV`.

LEFT_IMP_FORALL_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
`EXISTS_IMP_CONV`, `RIGHT_IMP_FORALL_CONV`.

LEFT_OR_EXISTS_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
`EXISTS_OR_CONV`, `OR_EXISTS_CONV`, `RIGHT_OR_EXISTS_CONV`.

LEFT_OR_FORALL_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
`OR_FORALL_CONV`, `FORALL_OR_CONV`, `RIGHT_OR_FORALL_CONV`.

lhs

```
lhs : (term -> term)
```

Synopsis
Returns the left-hand side of an equation.

Description
`lhs "t1 = t2"` returns `"t1"`.

Failure
Fails with `lhs` if the term is not an equation.

See also
`rhs`, `dest_eq`.

libraries

```
libraries : (void -> string list)
```

Synopsis
Evaluating `libraries()` returns a list of the libraries that have been successfully loaded during the current session.

Failure
Never fails.

See also
`library_pathname`, `load_library`.

LIST_BETA_CONV

```
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 "(\x y. x+y) 1 2";;
|- (\x y. x + y)1 2 = 1 + 2
```

See also
`BETA_CONV`, `BETA_RULE`, `BETA_TAC`, `RIGHT_BETA`, `RIGHT_LIST_BETA`.

LIST_CONJ

```
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` will fail with `‘end_itlist‘` if applied to an empty list of theorems.

Comments
The system shows the type as `proof`.

`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
`BODY_CONJUNCTS`, `CONJ`, `CONJUNCT1`, `CONJUNCT2`, `CONJUNCTS`, `CONJ_PAIR`, `CONJ_TAC`.

LIST_INDUCT

```
LIST_INDUCT : ((thm # thm) -> thm)
```

Synopsis
Performs proof by structural induction on lists.

Description

The derived inference rule `LIST_INDUCT` implements the rule of mathematical induction:

```
    A1 |- P[NIL/l]      A2 |- !t. P[t/l] ==> !h. P[CONS h t/l]
   ------------------------------------------------------------  LIST_INDUCT
                     A1 u A2 |- !l. P
```

When supplied with a theorem `A1 |- P[NIL]`, which asserts the base case of a proof of the proposition `P[l]` by structural induction on the list `l`, and the theorem

```
   A2 |- !t. P[t] ==> !h. P[CONS h t]
```

which asserts the step case in the induction on `l`, the inference rule `LIST_INDUCT` returns `A1 u A2 |- !l. P[l]`.

Failure

`LIST_INDUCT th1 th2` fails if the theorems `th1` and `th2` do not have the forms `A1 |- P[NIL]` and `A2 |- !t. P[t] ==> !h. P[CONS h t]` respectively (where the empty list `NIL` in `th1` and the list `CONS h t` in `th2` have the same type).

See also

`LIST_INDUCT_TAC`.

LIST_INDUCT_TAC

```
LIST_INDUCT_TAC : tactic
```

Synopsis

Performs tactical proof by structural induction on lists.

Description

`LIST_INDUCT_TAC` reduces a goal `!l.P[l]`, where `l` ranges over lists, to two subgoals corresponding to the base and step cases in a proof by structural induction on `l`. The induction hypothesis appears among the assumptions of the subgoal for the step case. The specification of `LIST_INDUCT_TAC` is:

```
                    A ?- !l. P
   ====================================================  LIST_INDUCT_TAC
    A |- P[NIL/l]   A u {P[l'/l]} ?- !h. P[CONS h l'/l]
```

where `l'` is a primed variant of `l` that does not appear free in the assumptions `A` (usually, `l'` is just `l`). When `LIST_INDUCT_TAC` is applied to a goal of the form `!l.P`, where `l` does not appear free in `P`, the subgoals are just `A ?- P` and `A u {P} ?- !h.P`.

Failure
`LIST_INDUCT_TAC g` fails unless the conclusion of the goal `g` has the form `!l.t`, where the variable `l` has type `(ty)list` for some type `ty`.

See also
`LIST_INDUCT`.

list_mk_abs

```
list_mk_abs : ((term list # term) -> term)
```

Synopsis
Iteratively constructs abstractions.

Description
`list_mk_abs(["x1";...;"xn"],"t")` returns `"\x1 ... xn. t"`.

Failure
Fails with `list_mk_abs` if the terms in the list are not variables.

Comments
The system shows the type as `goal -> term`.

See also
`strip_abs`, `mk_abs`.

list_mk_comb

```
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 with `list_mk_comb` 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("1",[]);;
"1" : term

#list_mk_comb("$/\",["T"]);;
"$/\ T" : term

#list_mk_comb("$/\",["1"]);;
evaluation failed     list_mk_comb
```

See also

`strip_comb, mk_comb.`

list_mk_conj

```
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 with `list_mk_conj` 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"];;
"T /\ F /\ T" : term

#list_mk_conj ["T";"1";"F"];;
evaluation failed     list_mk_conj

#list_mk_conj ["1"];;
"1" : term
```

See also
`conjuncts`, `mk_conj`.

list_mk_disj

```
list_mk_disj : (term list -> term)
```

Synopsis
Constructs the disjunction of a list of terms.

Description
`list_mk_disj(["t1";...;"tn"])` returns `"t1 \/ ... \/ tn"`.

Failure
Fails with `list_mk_disj` 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"];;
"T \/ F \/ T" : term

#list_mk_disj ["T";"1";"F"];;
evaluation failed     list_mk_disj

#list_mk_disj ["1"];;
"1" : term
```

See also
`disjuncts`, `mk_disj`.

list_mk_exists

```
list_mk_exists : ((term list # term) -> term)
```

Synopsis
Iteratively constructs existential quantifications.

Description
`list_mk_exists(["x1";...;"xn"],"t")` returns `"?x1 ... xn. t"`.

Failure
Fails with `list_mk_exists` 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.

Comments
The system shows the type as `(goal -> term)`.

See also
`strip_exists, mk_exists.`

LIST_MK_EXISTS

```
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
`EXISTS_EQ`, `MK_EXISTS`.

list_mk_forall

```
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 with `list_mk_forall` 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.

Comments
The system shows the type as `(goal -> term)`.

See also
`strip_forall`, `mk_forall`.

list_mk_imp

```
list_mk_imp : (goal -> term)
```

Synopsis
Iteratively constructs implications.

Description
`list_mk_imp(["t1";...;"tn"],"t")` returns `"t1 ==> ( ... (tn ==> t)...)"`.

Failure
Fails with `list_mk_imp` if any of `t1`,...,`tn` are not of type `":bool"` or 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");;
"T ==> F ==> T" : term

#list_mk_imp (["T";"1"],"T");;
evaluation failed     list_mk_imp

#list_mk_imp (["T";"F"],"1");;
evaluation failed     list_mk_imp

#list_mk_imp ([],"1");;
"1" : term
```

See also
`strip_imp, mk_imp.`

list_mk_pair

```
list_mk_pair : (term list -> term)
```

Synopsis
Constructs a tuple from a list of terms.

Description
`list_mk_pair(["t1";...;"tn"])` returns `"(t1,...,tn)"`.

Failure
Fails with `list_mk_pair` if the list is empty.

Example

```
#list_mk_pair ["1";"T";"2"];;
"1,T,2" : term

#list_mk_pair ["1"];;
"1" : term
```

See also
strip_pair, mk_pair.

LIST_MP

```
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
EQ_MP, MATCH_MP, MATCH_MP_TAC, MP, MP_TAC.

list_of_binders

```
list_of_binders : term list
```

Synopsis
List of binders in the current theory.

Description
For implementation reasons, a list containing the binders in the current theory is maintained in the assignable ML variable `list_of_binders`. This variable is not for general use, and users should never make assignments to it.

Failure
Evaluating the assignable variable `list_of_binders` never fails.

map2

```
map2 : (((* # **) -> ***) -> (* list # ** list) -> *** list)
```

Synopsis
Maps a binary 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 with `map2` if the two lists are of different lengths.

Example

```
#map2 $+ ([1;2;3],[3;2;1]);;
[4; 4; 4] : int list
```

See also
`map, uncurry.`

mapfilter

```
mapfilter : ((* -> **) -> * list -> ** list)
```

Synopsis
Applies a function to every element of a list, returning a list of results for those elements for which application succeeds.

Failure
Never fails.

Example

```
#mapfilter hd [[1;2;3];[4;5];[];[6;7;8];[]];;
[1; 4; 6] : int list
```

See also
`filter`, `map`.

MAP_EVERY

```
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
`EVERY`, `FIRST`, `MAP_FIRST`, `THEN`.

MAP_FIRST

```
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
`EVERY`, `FIRST`, `MAP_EVERY`, `ORELSE`.

MATCH_ACCEPT_TAC

```
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
`ACCEPT_TAC`.

MATCH_MP

```
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.

```
#let ith =
# (GENL ["x:num"; "z:num"] o DISCH_ALL o AP_TERM "$+ (w + z)")
#   (ASSUME "x:num = y");;
ith = |- !x z. (x = y) ==> ((w + z) + x = (w + z) + y)

#let th = ASSUME "w:num = z";;
th = w = z |- w = z

#MATCH_MP5 ith th;;
w = z |- !z'. (w' + z') + w = (w' + z') + z
```

See also

`EQ_MP, MATCH_MP_TAC, MP, MP_TAC.`

MATCH_MP_TAC

```
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
`EQ_MP, MATCH_MP, MP, MP_TAC.`

match_term

```
match_term :
term -> term -> (term,term) subst * (hol_type,hol_type) subst
```

Synopsis
Finds instantiations to match one term to another.

Description

When applied to two terms, `match_term` attempts to find a set of type and term instantiations for the first term (only) to make it alpha-convertible to the second. If it succeeds, it returns the instantiations in the form of a pair containing a term substitution and a type substitution. If the first term represents the conclusion of a theorem, the returned instantiations are of the appropriate form to be passed to `INST_TY_TERM`.

Failure

Fails if the term cannot be matched by one-way instantiation.

Example

The following shows how `match_term` could be used to match the conclusion of a theorem to a term.

```
  - val th = REFL ``x:'a``;
  th = |- x = x

  - match_term (concl th) ``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

Note that there is no guarantee that the returned instantiations will be possible for `INST_TY_TERM` to achieve, because some of the variables (term or type) which need to be instantiated may be free in the assumptions, eg:

```
- (show_types := true; show_assums := true);
() : unit

- val th = ASSUME ``x:'a = x``;
val th = [(x :'a) = (x :'a)] |- (x :'a) = (x :'a) : thm

- match_term (concl th) (--`1 = 1`--);
val it = ([{redex = ``x :num``, residue = ``1``}],
          [{redex = ``:'a``, residue = ``:num``}])
  : term subst * hol_type subst

- INST_TY_TERM it th handle e => Raise e;
Exception raised at Thm.INST_TYPE:
type variable(s) in assumptions would be instantiated in concl
```

In fact, for instantiating a theorem, `PART_MATCH` is usually easier.

See also

`match_type, INST_TY_TERM, PART_MATCH.`

match_type

```
match_type : hol_type -> hol_type -> hol_type subst
```

Synopsis

Finds a substitution theta such that instantiating the first argument with theta equals the second argument.

Description

If `match_type ty1 ty2` succeeds, then

```
Type.type_subst (match_type ty1 ty2) ty1 = ty2
```

`match_type` is not found in hol88.

Failure

It fails if no such substitution can be found.

Example

```
- match_type (==`:'a`==) (==`:num`==);
> val it =
    [{redex = (==`:'a`==), residue = (==`:num`==)}] : hol_type subst

- let val patt = ==`:('a -> bool) -> 'b`==
=     val ty =   ==`:(num -> bool) -> bool`==
= in
= type_subst (match_type patt ty) patt = ty
= end;
> val it = true : bool
```

See also

match_term

max_print_depth

```
max_print_depth : (int -> int)
```

Synopsis

Sets depth of block nesting.

Description

The function `max_print_depth` is used to define the maximum depth of nesting that the pretty printer will allow. If the number of blocks is greater than the the value set by `max_print_depth` then the blocks are truncated and this is indicated by the holophrast `&`. The function always returns the previous maximum depth setting.

Failure

Never fails.

Example

If the maximum depth setting is the default (500) and we want to change this to 20 the command will be:

```
   #max_print_depth 20;;
```

The system will then return the following:

```
   500 : int
```

See also
`print_begin, print_ibegin, print_end, set_margin, print_break`

mem

```
mem : (* -> * list -> bool)
```

Synopsis
Tests whether a list contains a certain member.

Description
`mem x [x1;...;xn]` returns `true` if some `xi` in the list is equal to `x`. Otherwise it returns `false`.

Failure
Never fails.

See also
`find, tryfind, exists, forall, assoc, rev_assoc.`

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

`ASM_MESON_TAC, GEN_MESON_TAC`

mk_abs

```
mk_abs : {Bvar: term, Body : term} -> term
```

Synopsis
Constructs an abstraction.

Description
`mk_abs {Bvar = v, Body = t}` returns the abstraction `--‘\v. t‘--`.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Term", origin_function = "mk_abs",
            message = "Bvar not a variable"}
```

See also
`dest_abs, is_abs, list_mk_abs, mk_var, mk_const, mk_comb.`

MK_ABS

```
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
`ABS, HALF_MK_ABS, MK_COMB, MK_EXISTS.`

mk_comb

```
mk_comb : {Rator : term, Rand : term} -> term
```

Synopsis
Constructs a combination (function application).

Description
`mk_comb {Rator = t1, Rand = t2}` returns the combination `--‘t1 t2‘--`.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Term", origin_function = "mk_comb",
            message = "incompatible types"}
```

where `t1` does not have a function type, orif `t1` has a function type, but its domain does not equal the type of `t2`.

Example

```
   - mk_comb{Rator = --‘$~‘--, Rand = --‘T‘--};
   > val (--‘~T‘--) : term

   - mk_comb{Rator = --‘T‘--, Rand = --‘T‘--} handle e => Raise e;

   Exception raised at Term.mk_comb:
   incompatible types
   ! Uncaught exception:
   ! HOL_ERR <poly>
```

See also
`dest_comb, is_comb, list_mk_comb, mk_var, mk_const, mk_abs.`

MK_COMB

```
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
`AP_TERM, AP_THM.`

mk_cond

```
mk_cond : {cond :term, larm :term, rarm :term} -> term
```

Synopsis
Constructs a conditional term.

Description
`mk_cond{cond = t, larm = t1, rarm = t2}` returns `--‘t => t1 | t2‘--`.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_cond",
            message = ""}
```

if `cond` is not of type `==‘:bool‘==` or if `larm` and `rarm` are of different types.

See also
`dest_cond, is_cond.`

mk_conj

```
mk_conj : {conj1 :term, conj2 : term} -> term
```

Synopsis
Constructs a conjunction.

Description
`mk_conj{conj1 = t1, conj2 = t2}` returns ``--`t1 /\ t2`--``.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_conj",
            message = "Non-boolean argument"}
```

See also
`dest_conj, is_conj, list_mk_conj.`

mk_cons

```
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 with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_cons",
            message = ""}
```

if `tl` is not a list or if `hd` is not of the same type as the elements of the list.

See also
`dest_cons, is_cons, mk_list, dest_list, is_list.`

mk_const

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

Synopsis
Constructs a constant.

Description
`mk_const{Name = "const", Ty = ty}` returns the constant –‘const:ty‘–.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_const",
            message}
```

where `message` is prefixed with `"not in term signature"` if the string supplied is not the name of a known constant, or `"not a type instance"` if the string is known as a constant but the type supplied is not an instance of the declared type of that constant.

Example

```
   - mk_const {Name = "T", Ty = ==`:bool`==};
   > val it = (--`T`--) : term

   - Dsyntax.mk_const {Name = "T", Ty = ==`:num`==} handle e => Raise e;
   Exception raised at Dsyntax.mk_const:
   not a type instance: T

   - mk_const {Name = "test", Ty = ==`:bool`==} handle e => Raise e;
   Exception raised at Dsyntax.mk_const:
   not in term signature: test
```

See also
`dest_const, is_const, mk_var, mk_comb, mk_abs.`

mk_disj

```
mk_disj : {disj1 :term, disj2 : term} -> term
```

Synopsis
Constructs a disjunction.

Description
`mk_disj{disj1 = t1, disj2 = t2}` returns `--`t1 \/ t2`--`.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_disj",
            message = "Non-boolean argument"}
```

See also
dest_disj, is_disj, list_mk_disj.

mk_eq

```
mk_eq : {lhs : term, rhs: term} -> term
```

Synopsis
Constructs an equation.

Description
`mk_eq{lhs = t1, rhs = t2}` returns `--‘t1 = t2‘--`.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_eq",
            message = "lhs and rhs have different types"}
```

See also
dest_eq, is_eq.

mk_exists

```
mk_exists : {Bvar : term, Body : term} -> term
```

Synopsis
Term constructor for existential quantification.

Description
`mk_exists{Bvar = v, Body = t}` returns `--‘?v. t‘--`.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_exists",
            message = ""}
```

if `Bvar` is not a variable or if `Body` is not of type ==`:bool`==.

See also
`dest_exists, is_exists, list_mk_exists.`

MK_EXISTS

```
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
`AP_TERM, EXISTS_EQ, GEN, LIST_MK_EXISTS, MK_ABS.`

mk_forall

```
mk_forall : {Bvar : term, Body : term} -> term
```

Synopsis
Term constructor for universal quantification.

Description
`mk_forall{Bvar = v, Body = t}` returns `--‘!v. t‘--`.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_forall",
            message = ""}
```

if `Bvar` is not a variable or if `Body` is not of type `==‘:bool‘==`.

See also
`dest_forall, is_forall, list_mk_forall.`

mk_imp

```
mk_imp : {ant : term, conseq : term} -> term
```

Synopsis
Constructs an implication.

Description
`mk_imp{ant = t1, conseq = t2}` returns `--‘t1 ==> t2‘--`.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_imp",
            message = "Non-boolean argument"}
```

See also
`dest_imp, is_imp, list_mk_imp.`

mk_let

```
mk_let : {func : term, arg : term} -> term
```

Synopsis

Constructs a `let` term.

Description

`mk_let {func = f, arg = x)` returns `--‘LET f x‘--`. If `func` is of the form `--‘\y. t‘--` then the result will be pretty-printed as `--‘let y = x in t‘--`.

Failure

Fails with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_let",
            message = ""}
```

if the types of `func` and `arg` are such that `--‘LET func arg‘--` is not well-typed. `--‘LET‘--` has most general type:

```
   ==‘:('a -> 'b) -> 'a -> 'b‘==
```

Example

```
- mk_let{func = --‘$= 1‘--, arg = --‘2‘--};
> val it = (--‘LET ($= 1) 2‘--) : term

- mk_let{func= --‘\y. y = 1‘--, arg = --‘2‘--};
> val it = (--‘let y = 2 in (y = 1)‘--) : term
```

See also

`dest_let, is_let.`

mk_list

```
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 with

```
    HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_list",
            message = ""}
```

if any term in the list is not of the type specified as the second argument.

See also
`dest_list, is_list, mk_cons, dest_cons, is_cons.`

`mk_neg`

```
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
`dest_neg, is_neg.`

`mk_pabs`

```
mk_pabs : {varstruct :term, body :term} -> term
```

Synopsis
Constructs a paired abstraction.

Description
`mk_pabs {varstruct = --`(v1,..(..)..,vn)`--, body = t}` returns the abstraction `--`\(v1,..(..`

Failure
Fails unless `varstruct` is an arbitrarily nested pair composed from variables.

See also
`dest_pabs`, `is_pabs`, `mk_abs`.

mk_pair

```
mk_pair : {fst :term, snd :term} -> term
```

Synopsis
Constructs object-level pair from a pair of terms.

Description
`mk_pair{fst = t1, snd = t2}` returns ``--`(t1,t2)`--``.

Failure
Never fails.

See also
`dest_pair`, `is_pair`, `list_mk_pair`.

mk_primed_var

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

Synopsis
Primes a variable name sufficiently to make it distinct from all constants.

Description
When applied to a record made from 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{Name = "x", Ty = ==`:bool`==};
> val it = (--`x`--) : term

- new_constant{Name = "x", Ty = ==`:num`==};
> val it = () : unit

- mk_primed_var{Name = "x",Ty = ==`:bool`==};
> val it = (--`x'`--) : term
```

See also

genvar, variant.

mk_select

```
mk_select : {Bvar : term, Body : term} -> term
```

Synopsis

Constructs a choice-term.

Description

`mk_select{Bvar = v, Body = t}` returns `` --`@var. t`-- ``.

Failure

Fails with

```
HOL_ERR{origin_structure = "Dsyntax", origin_function = "mk_select",
        message = ""}
```

if `Bvar` is not a variable or if `Body` is not of type `` ==`:bool`== ``.

See also

dest_select, is_select.

mk_simpset

```
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
`++, rewrites, SIMP_CONV`

mk_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)
```

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:

```
   #mk_thm([],"F");;
   |- F
```

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 5 axioms and 8 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.

Note that the system shows the type of `mk_thm` as `(goal -> thm)`.

See also
`new_axiom`.

mk_type

```
mk_type : {Tyop :string, Args :hol_type list} -> hol_type
```

Synopsis
Constructs a type (other than a variable type).

Description
`mk_type{Tyop = "op", Args = [ty1,...,tyn]}` returns

```
   ==`:(ty1,...,tyn)op`==
```

where `op` is the name of a known `n`-ary type constructor.

Failure
Fails with

```
    HOL_ERR{origin_structure = "Dsyntax",origin_function="mk_type", message}
```

where `message` is ”type op not defined”, if `Tyop` is not the name of a known type, or ”arities don’t match” if the type is known but the length of the list of argument types is not equal to the arity of the type constructor.

Example

```
- mk_type {Tyop = "bool", Args = []};
> val it = (==‘:bool‘==) : hol_type

- mk_type {Tyop = "list", Args = [==‘:bool‘==]};
> val it = (==‘:bool list‘==) : hol_type

- mk_type {Tyop = "fun", Args = [==‘:num‘==,  ==‘:bool‘==]};
> val it = (==‘:num -> bool‘==) : hol_type
```

See also
`dest_type`, `mk_vartype`.

mk_var

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

Synopsis
Constructs a variable of given name and type.

Description
`mk_var{Name = "var", Ty = ty}` returns the variable `--‘var:ty‘--`.

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
`dest_var`, `is_var`, `mk_const`, `mk_comb`, `mk_abs`.

mk_vartype

```
mk_vartype : (string -> type)
```

Synopsis
Constructs a type variable of the given name.

Description
`mk_vartype(‘*...‘)` returns `":*..."`.

Failure
Fails with `mk_vartype` if the string does not begin with `‘*‘`.

Example
```
#mk_vartype ‘*test‘;;
":*test" : type

#mk_vartype ‘test‘;;
evaluation failed     mk_vartype
```

Comments
`mk_vartype` can be used to create type variables with names which will not parse, i.e. they cannot be entered by quotation.

See also
`dest_vartype, is_vartype, mk_type.`

MP

```
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
EQ_MP, LIST_MP, MATCH_MP, MATCH_MP_TAC, MP_TAC.

MP_TAC

```
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
MATCH_MP_TAC, MP, UNDISCH_TAC.

NEG_DISCH

```
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
`DISCH, NOT_ELIM, NOT_INTRO.`

new_axiom

```
Compat.new_axiom : (string * term) -> thm
```

Synopsis
Sets up a new axiom in the current theory.

Description
Found in the hol88 library. If `tm` is a term of type `bool`, a call `new_axiom("name",tm)` creates a theorem

```
   |- !x1..xn. tm
```

and stores it away in the theory file. Note that all free variables in `tm` are generalized automatically before the axiom is set up.

Failure
Fails if HOL is not in draft mode, or there is already an axiom or definition of that name in the current theory, or it the given term does not have type `bool`. The function will not be available unless the hol88 library is loaded.

Example

```
- new_theory "gurk";
() : unit

- new_axiom("untrue",--`x = 1`--));
|- !x. x = 1
```

Comments

hol90 doesn't have `new_axiom`; use `new_open_axiom` instead, which does not automatically generalize the term being asserted as an axiom. 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

`mk_thm`, `new_definition`.

new_binder

```
new_binder : {Name :string, Ty :hol_type} -> unit
```

Synopsis

Sets up a new binder in the current theory.

Description

A call `new_binder{Name ="bnd",Ty = 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 HOL is not in draft mode, or there is already a constant of some sort of that name in the current theory, or if the type does not correspond to the above pattern.

Example

```
- new_theory "anorak";
() : unit

- new_binder{Name = "!!", Ty = ==`:(bool->bool)->bool`==};
() : unit

- --`!!x. T`--;
(--`!! x. T`--) : term
```

See also

binders, is_binder, constants, infixes, new_constant, new_infix, new_definition, new_infix_definition, new_binder_definition.

new_binder_definition

```
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 is not already a constant, 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 called when HOL is not in draft mode. It also fails if there is already an axiom, definition or specification with the given name in the current theory segment, if `‘b‘` is already a constant in the current theory or is not an allowed name for a constant, 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 teh 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‘,
   "$?! = \P:(*->bool). ($? P) /\ (!x y. ((P x) /\ (P y)) ==> (x=y))");;
|- $?! = (\P. $? P /\ (!x y. P x /\ P y ==> (x = y)))
```

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‘, "$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

`new_definition, new_gen_definition, new_infix_definition, new_infix_list_rec_definition, new_prim_rec_definition, new_list_rec_definition, new_prim_rec_definition.`

new_constant

```
new_constant : {Name :string, Ty :hol_type} -> unit
```

Synopsis
Declares a new constant in the current theory.

Description
A call `new_constant{Name="c", Ty = ty}` makes `c` a constant in the current theory. Note that it does not specify its value. The constant may have a polymorphic type, which can be used in arbitrary instantiations.

Failure
Fails if HOL is not in draft mode, or if the name is not a valid constant name, or there is already a constant of that name in the current theory.

Example
```
   - new_theory "zonk";
   () : unit

   - new_constant{Name = "graham's_number", Ty = ==`:num`==};
   () : unit
```

See also
`constants`, `infixes`, `binders`, `is_constant`, `is_infix`, `is_binder`, `new_infix`, `new_binder`, `new_definition`, `new_infix_definition`, `new_binder_definition`.

new_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 (the name) `name`. Optionally, the definitional term argument may have any of its variables universally quantified.

Failure
`new_definition` fails if called when HOL is not in draft mode. It also fails if there is already an axiom, definition or specification of the given name in the current theory segment; if `‘c‘` is already a constant in the current theory or is not an allowed name for a constant; if `t` contains free variables that are not in any of the variable structures `v_1`, ..., `v_n` (this is equivalent to requiring `\v_1 ... v_n. t` to be a closed term); or 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.thm
```

See also
`new_binder_definition, new_gen_definition, new_infix_definition, new_infix_list_rec_definition, new_prim_rec_definition, new_list_rec_definition, new_prim_rec_definition, new_recursive_definition, new_specification.`

new_gen_definition

```
Parse.new_gen_definition : (string * term * fixity) -> thm
```

Synopsis
Defines a new constant and associates it with a parsing fixity.

Description

The function `new_gen_definition` provides a facility for definitional extensions to the current theory. It takes a tuple of three arguments. The first component of this tuple is the name under which the resulting definition will be saved in the current theory segment. The second component is a term giving the desired definition. The third component is a fixity (typically one of `Binder`, `Infixl n`, `Infixr n`, `Suffix n`, `TruePrefix n` or `Closefix`). The value returned by `new_gen_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_gen_definition flag (‘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, generally of the form `|- !x_1 ... x_m. c v_1 ... v_n = t` , and is saved in the current theory under (the name) `name`. If `flag` is `‘infix‘` or `‘binder‘`, the constant is given infix or binder status accordingly. Optionally, the definitional term argument may have any of its variables universally quantified.

Failure

`new_gen_definition` fails if there is already an axiom, definition or specification of the given name in an ancestral theory segment; if `c` is not an allowed name for a constant; if `t` contains free variables that are not in any of the variable structures `v_1`, ..., `v_n` (this is equivalent to requiring `\v_1 ... v_n. t` to be a closed term); or 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`.

See also

`DEF_EXISTS_RULE, new_binder_definition, new_definition, new_infix_definition, new_specification.`

new_infix

```
new_infix : {Name :string, Ty :hol_type, Prec :int} -> unit
```

Synopsis

Declares a new infix constant in the current theory.

Description

A call `new_infix{Name = "i",Ty = ty, Prec = 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)`. The initial infixes (and their precedences) in the system are:

```
           $,  ---> 50
           $=  ---> 100
         $==>  ---> 200
          $\/  ---> 300
          $/\  ---> 400
       $>, $<  ---> 450
     $>=, $<=  ---> 450
       $+, $-  ---> 500
     $*, $DIV  ---> 600
         $MOD  ---> 650
         $EXP  ---> 700
         $o    ---> 800
```

Note that the arithmetic operators `+`, `-`, `*`, `DIV` and `MOD` are left associative in hol98 releases from Taupo onwards.

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_theory "groke";
   <<HOL message: Created theory "groke".>>
 > val it = () : unit

  - new_infix{Name = "orelse", Ty ==`:bool->bool->bool`==, Prec = 50};
  val it = () : unit

  - --`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 = {Rator = (--`$orelse (T \/ T)`--),  Rand = (--`F`--)} : ...
```

See also

`add_infix, precedence, constants, infixes, binders, is_constant, is_infix, is_binder, new_constant, new_binder, new_definition, new_infix_definition, new_binder_definition.`

new_infixl_definition

```
Parse.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")`, where `ix` is not already a constant, 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 there is already an axiom, definition or specification of the given name in an ancestral theory segment; if `‘ix‘` is not an allowed name for a constant; 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‘, "$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

`new_binder_definition`, `new_definition`, `new_gen_definition`, `new_infixr_definition`, `new_infix_list_rec_definition`, `new_prim_rec_definition`, `new_list_rec_definition`, `new_prim_rec_definition`.

new_infixr_definition

```
Parse.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.

Failure
`new_infixr_definition` fails if there is already an axiom, definition or specification of the given name in an ancestral theory segment; if `‘ix‘` is not an allowed name for a constant; 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`.

See also
`new_definition, new_infix, new_infixl_definition`

new_infix_prim_rec_definition

```
Compat.new_infix_prim_rec_definition : (string * term) -> thm
```

Synopsis
Defines an infix primitive recursive function over the type `num`.

Description
Found in the hol88 library. The function `new_infix_prim_rec_definition` provides the facility for defining primitive recursive functions with infix status on the type `num`. 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_infix_prim_rec_definition` is a theorem which states the definition requested by the user. This theorem is derived by formal proof from an instance of the theorem `num_Axiom`:

```
|- !e f. ?! fn. (fn 0 = e) /\ (!n. fn(SUC n) = f(fn n)n)
```

Evaluating

```
new_infix_prim_rec_definition
 ("fun_DEF",
  (--`(fun 0 x = f_1[x]) /\
      (fun (SUC n) x = f_2[fun n x', n, x])`--));;
```

where all the free variables in the term `x'` are contained in `{n, x}`, automatically proves the theorem:

```
|-  ?fun. !x. fun 0 x = f_1[x] /\
          !x. fun (SUC n) x = f_2[fun n x', n, x]
```

and then declares a new constant `fun` with this property and infix status as its specification. This constant specification is returned as a theorem and is saved with name `fun_DEF` in the current theory segment.

The ML function `new_infix_prim_rec_definition` is, in fact, slightly more general than is indicated above. In particular, a curried primitive recursive function can be defined by primitive recursion on either one of its arguments using this ML function. The ML function `new_infix_prim_rec_definition` also allows the user to partially specify the value of a function defined (possibly recursively) on the natural numbers by giving its value for only one of `0` or `SUC n`.

Failure

Failure occurs if HOL cannot prove there is a function satisfying the specification (ie. if the term supplied to `new_prim_rec_definition` is not a well-formed primitive recursive definition); if the type of `fun` is not of the form `ty_1->ty_2->ty_3`, or if any other condition for making a constant specification is violated (see the failure conditions for `new_specification`). The function will not be accessible unless the hol88 library has been loaded.

Example

Here is the recursive definition of the constant + used by the system:

```
new_infix_prim_rec_definition
 ("ADD",
  (--`($+ 0 n = n) /\
      ($+ (SUC m) n = SUC($+ m n))`--))
```

The $'s are there (as documentation) to indicate that the constant + is being declared to be an infix. Evaluating this ML expression will create the following constant specification in the current theory segment:

```
ADD = |- (!n. 0 + n = n) /\ (!m n. (SUC m) + n = SUC(m + n))
```

Comments

`new_infix_prim_rec_definition` is not in hol90; it has been superceded by `new_recursive_definit`

See also

`new_definition`, `new_infix_definition`, `new_infix_list_rec_definition`, `new_prim_rec_definition`, `new_list_rec_definition`, `new_recursive_definition`, `new_type_definition`, `new_specification`, `num_Axiom`.

new_list_rec_definition

```
new_list_rec_definition : ((string # term) -> thm)
```

Synopsis

Defines a primitive recursive function over the type of lists.

Description

The function `new_list_rec_definition` provides the facility for defining primitive recursive functions on the type of lists. 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_list_rec_definition` is a theorem which states the definition requested by the user. This theorem is derived by formal proof from an instance of the theorem list_Axiom:

```
|- !x f. ?! fn. (fn[] = x) /\ (!h t. fn(CONS h t) = f(fn t)h t)
```

Evaluating

```
new_list_rec_definition
 (`fun_DEF`,
  "(fun x_1 ... [] ... x_i = f_1[x_1, ..., x_i]) /\
   (fun x_1 ... (CONS h t) ... x_i =
          f_2[fun t_1 ... t ... t_i, x_1, ..., h, t, ..., x_i])");;
```

where all the free variables in the terms t_1, ..., t_i are contained in {h,t,x_1,...,x_i}, automatically proves the theorem:

```
|-  ?fun. !x_1 ... x_i. fun x_1 ... [] ... x_i = f_1[x_1, ..., x_i] /\
          !x_1 ... x_i. fun (CONS h t) x_1 ... x_i =
                        f_2[fun t_1 ... t ... t_i, x_1, ..., h, t, ...,x_i]
```

and then declares a new constant fun with this property as its specification. This constant specification is returned as a theorem by new_list_rec_definition and is saved with name fun_DEF in the current theory segment.

The ML function new_list_rec_definition also allows the user to partially specify the value of a function defined (possibly recursively) on lists by giving its value for only one of [] or CONS h t. See the examples below.

Failure

Failure occurs if HOL cannot prove there is a function satisfying the specification (ie. if the term supplied to mlnew_list_rec_definition 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

The HOL system defines a length function, LENGTH, on lists by the primitive recursive definition on lists shown below:

```
new_list_rec_definition
  (`LENGTH`,
  "(LENGTH NIL = 0) /\
   (!h:*. !t. LENGTH (CONS h t) = SUC (LENGTH t))")
```

When this ML expression is evaluated, HOL uses list_Axiom to prove existence of a function that satisfies the given primitive recursive definition, introduces a constant to name this function using a constant specification, and stores the resulting theorem:

```
 LENGTH   |- (LENGTH[] = 0) /\ (!h t. LENGTH(CONS h t) = SUC(LENGTH t))
```

in the current theory segment (in this case, the theory `list`).

Using `new_list_rec_definition`, the predicate `NULL` and the selectors `HD` and `TL` are defined in the theory `list` by the specifications:

```
NULL |- NULL[] /\ (!h t. ~NULL(CONS h t))

HD   |- !(h:*) t. HD(CONS h t) = h

TL   |- !(h:*) t. TL(CONS h t) = t
```

See also

`new_definition`, `new_infix_definition`, `new_infix_list_rec_definition`, `new_infix_prim_rec_definition`, `new_prim_rec_definition`, `new_recursive_definition`, `new_type_definition`, `new_specification`, `list_Axiom`.

new_axiom

```
new_open_axiom : (string * term) -> thm
```

Synopsis

Sets up a new axiom in the current theory.

Description

If `tm` is a term of type `bool`, a call `new_open_axiom("name",tm)` creates a theorem

```
|- tm
```

and stores it away in the current theory.

Failure

Fails if HOL is not in draft mode, or there is already an axiom or definition of that name in the current theory, or it the given term does not have type `bool`.

Example

```
- new_theory "gurk";
() : unit

- new_axiom("untrue",--`x = 1`--));
|- x = 1
```

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

`mk_thm`, `new_definition`.

new_prim_rec_definition

```
Compat.new_prim_rec_definition : (string * term) -> thm
```

Synopsis

Define a primitive recursive function over the type `:num`.

Description

Found in the hol88 library. The function `new_prim_rec_definition` provides the facility for defining primitive recursive functions on the type `num`. 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_prim_rec_definition` is a theorem which states the definition requested by the user.

This theorem is derived by formal proof from an instance of the theorem `num_Axiom`:

```
|- !e f. ?! fn. (fn 0 = e) /\ (!n. fn(SUC n) = f(fn n)n)
```

Evaluating

```
new_prim_rec_definition
 ("fun_DEF",
  --`(fun x_1 ... 0 ... x_i = f_1[x_1, ..., x_i]) /\
     (fun x_1 ... (SUC n) ... x_i =
              f_2[fun t_1 ... n ... t_i, x_1, ..., n, ..., x_i])`--);
```

where all the free variables in the terms `t_1`, ..., `t_i` are contained in `{n, x_1, ..., x_i}`, automatically proves the theorem:

```
|-  ?fun. !x_1 ... x_i. fun x_1 ... 0 ... x_i = f_1[x_1, ..., x_i] /\
          !x_1 ... x_i. fun (SUC n) x_1 ... x_i =
                            f_2[fun t_1 ... n ... t_i, x_1, ..., n, ...,x_i]
```

and then declares a new constant `fun` with this property as its specification. This constant specification is returned as a theorem by `new_prim_rec_definition` and is saved with name `fun_DEF` in the current theory segment.

The ML function `new_prim_rec_definition` also allows the user to partially specify the value of a function defined (possibly recursively) on the natural numbers by giving its value for only one of `0` or `SUC n`. See the example below.

Failure

Failure occurs if HOL cannot prove there is a function satisfying the specification (ie. if the term supplied to `new_prim_rec_definition` 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`). The function will not be available unless the hol88 library has been loaded.

Example

A curried addition function `plus:num->num->num` can be defined by primitive recursion on its first argument:

```
- val PLUS = new_prim_rec_definition
=           (‘PLUS‘,
=             (--‘(plus 0 n = n) /\
=                 (plus (SUC m) n = SUC(plus m n))‘--));
PLUS = |- (!n. plus 0 n = n) /\ (!m n. plus(SUC m)n = SUC(plus m n))
```

or by primitive recursion on its second argument:

```
- val PLUS = new_prim_rec_definition
=             (‘PLUS‘,
=              (--‘(plus m 0 = m) /\
=                  (plus m (SUC n) = SUC(plus m n))‘--));
PLUS = |- (!m. plus m 0 = m) /\ (!m n. plus m(SUC n) = SUC(plus m n))
```

A decrement function `DEC`, whose value is specified for only positive natural numbers, can be defined using `new_prim_rec_definition` as follows

```
- val DEC = new_prim_rec_definition
=            (‘DEC‘, (--‘DEC (SUC n) = n‘--));
DEC = |- !n. DEC(SUC n) = n
```

This definition specifies the value of the function `DEC` only for positive natural numbers. In particular, the value of `DEC 0` is left unspecified, and the only non-trivial property that can be proved to hold of the constant `DEC` is the property stated by the theorem returned by the call to `new_prim_rec_definition` shown in the session above.

Comments

`new_prim_rec_definition` is not in hol90; it has been superceded by `new_recursive_definition`.

See also

`new_definition`, `new_infix_definition`, `new_infix_list_rec_definition`, `new_infix_prim_rec_definition`, `new_list_rec_definition`, `new_recursive_definition`, `new_type_definition`, `new_specification`, `num_Axiom`.

new_recursive_definition

```
new_recursive_definition :
{name:string,def:term,fixity:fixity,rec_axiom:thm} -> thm
```

Synopsis

Defines a primitive recursive function over a concrete recursive type.

Description

new_recursive_definition provides the 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. fixity is a value of type :fixity which indicates whether the defined function will be a prefix, binder, or infix. 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 ith 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 ith 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", fixity=f, 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. The constant is given the parsing status defined by f (one of Prefix, Infix ¡int¿, or Binder).

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",
=        fixity = Prefix,
=        rec_axiom = th,
=        def= --`(Leaves (LEAF (x:'a)) = 1) /\
=                (Leaves (NODE t1 t2) = (Leaves t1) + (Leaves t2))`--};
Leaves =
|- (!x. Leaves(LEAF x) = 1) /\
   (!t1 t2. Leaves(NODE t1 t2) = (Leaves t1) + (Leaves t2))
```

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",
=         fixity = Prefix,
=         rec_axiom = th,
=         def = --`(IsLeaf (NODE t1 t2) = F) /\
=                  (IsLeaf (LEAF (x:'a)) = T)`--};
IsLeaf = |- (!t1 t2. IsLeaf(NODE t1 t2) = F) /\ (!x. IsLeaf(LEAF x) = T)
```

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",
=        fixity = Prefix,
=        rec_axiom = th,
=        def = --`Label (LEAF (x:'a)) = x`--};
Label = |- !x. Label(LEAF x) = x
```

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 Subtree = new_recursive_definition
=        {name = "Subtree",
=         fixity = Infix 120,
=         rec_axiom = th,
=         def = --`(<< (t:'a bintree) (LEAF (x:'a)) = F) /\
=                  (<< t (NODE t1 t2) = (t = t1) \/
=                                       (t = t2) \/
=                                       (<< t t1) \/
=                                       (<< t t2))`--};
Subtree =
|- (!t x. t << (LEAF x) = F) /\
   (!t t1 t2.
     t << (NODE t1 t2) = (t = t1) \/ (t = t2) \/ (t << t1) \/ (t << t2))
```

Note that the constant << is an infix only after the definition has been made. Furthermore, the function << is recursive on its second argument.

See also

`define_type`, `prove_rec_fn_exists`.

new_specification

```
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 specification 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 whether the new constants are infixes or binders or neither. If `fi` is `Prefix` then `ci` is declared an ordinary constant, if it is `Infix i` then `ci` is declared an infix with precedence `i`, and if it is `Binder` then `ci` is declared a binder.

Failure

`new_specification` fails if called when HOL is not in draft mode. It also fails if there is already an axiom, definition or specification of the given name in the current theory segment; if the theorem argument has assumptions or free variables; 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. Failure also occurs if the type of `ci` is not suitable for a constant with the syntactic status specified by the fixity `fi`. 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 = Infix 650, const_name = "MOD"},
                             {fixity = Infix 600, const_name = "DIV"}],
                   sat_thm = th};
```

This introduces the constants MOD and DIV with the defining property shown above.

See also

new_definition, new_binder_definition, new_gen_definition, new_infix_definition.

new_theory

```
new_theory : (string -> void)
```

Synopsis

Creates a new theory by extending the current theory with a new theory segment.

Description

A theory consists of a hierarchy of named parts called theory segments. The theory segment at the top of the hierarchy tree in each theory is said to be current. All theory segments have a theory of the same name associated with them consisting of the theory segment itself and all its ancestors. Each axiom, definition, specification and theorem belongs to a particular theory segment.

Calling new_theory ‘thy‘ creates a new theory segment and associated theory 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 call switches the system into draft mode. This allows new axioms, constants, types, constant specifications, infix constants, binders and parents to be added to the theory segment. Inconsistencies will be introduced into the theory if inconsistent axioms are asserted. New theorems can also be added as when in proof mode. The theory file in which the data of the new theory segment is ultimately stored will have name `thy.th` in the directory from which HOL was called. The theory segment might not be written to this file until the session is finished with a call to `close_theory`. If HOL is quitted without closing the session with `close_theory`, parts of the theory segment created during the session may be lost. If the system is in draft mode when a call to `new_theory` is made, the previous session is closed; all changes made in it will be written to the associated theory file.

Failure

The call `new_theory ‘thy‘` will fail if there already exists a file `thy.th` in the current search path. It will also fail if the name `thy.th` is unsuitable for a filename. Since it could involve writing to the file system, if a write fails for any reason `new_theory` will fail.

Uses

Hierarchically extending the current theory. By splitting a theory into theory segments using `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

`close_theory`, `current_theory`, `extend_theory`, `load_theory`, `new_axiom`, `new_binder`, `new_constant`, `new_definition`, `new_infix`, `new_parent`, `new_specification`, `new_type`, `print_theory`, `save_thm`, `search_path`.

new_type

```
new_type : {Name :string, Arity :int} -> unit
```

Synopsis

Declares a new type or type constructor.

Description

A call `new_type{Name = "t", Arity = 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
Fails if HOL is not in draft mode, or if the name is not a valid type name, or there is already a type operator of that 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`;;
() : unit

- new_type{Name="set", Arity=0};
() : unit

- new_infix{Name="mem",Ty = ==`:set->set->bool`==};
() : unit

- new_open_axiom("ext", --`(!z. z mem x = z mem y) ==> (x = y)`--);
|- (!z. z mem x = z mem y) ==> (x = y)
```

See also
`types, type_abbrevs, new_type_abbrev.`

new_type_definition

```
new_type_definition : {name :string, pred :term, inhab_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{name = "op", pred = "t", inhab_thm = |- ?x. t x}
```

results in `op` 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->ty. 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 `op` 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)op` and the set of values of type `ty` that satisfy `P`.

Failure

Executing `new_type_definition{name="op",pred="t",inhab_thm=th}` fails if `op` is already the name of a type or type operator in the current theory, if `"t"` does not have a type of the form `ty->bool` or `th` is not an assumption-free theorem of the form `|- ?x. t x`, if there already exists a constant definition, constant specification, type definition or axiom named `op_TY_DEF` in the current theory segment, or if HOL is not in draft mode.

See also

`define_new_type_bijections`, `prove_abs_fn_one_one`, `prove_abs_fn_onto`, `prove_rep_fn_one_one`, `prove_rep_fn_onto`.

NOT_ELIM

```
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
`IMP_ELIM`, `NOT_INTRO`.

NOT_EQ_SYM

```
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
`DEPTH_CONV`, `REFL`, `SYM`.

NOT_EXISTS_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
`EXISTS_NOT_CONV`, `FORALL_NOT_CONV`, `NOT_FORALL_CONV`.

NOT_FORALL_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
`EXISTS_NOT_CONV`, `FORALL_NOT_CONV`, `NOT_EXISTS_CONV`.

NOT_INTRO

```
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
`IMP_ELIM`, `NOT_ELIM`.

NO_CONV

```
NO_CONV : conv
```

Synopsis
Conversion that always fails.

Failure
`NO_CONV` always fails.

See also
`ALL_CONV`.

NO_TAC

```
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
`ALL_TAC`, `ALL_THEN`, `FAIL_TAC`, `NO_THEN`.

NO_THEN

```
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
`ALL_TAC`, `ALL_THEN`, `FAIL_TAC`, `NO_TAC`.

ONCE_ASM_REWRITE_RULE

```
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

ASM_REWRITE_RULE, FILTER_ASM_REWRITE_RULE, FILTER_ONCE_ASM_REWRITE_RULE, GEN_REWRITE_RULE, ONCE_DEPTH_CONV, ONCE_REWRITE_RULE, PURE_ASM_REWRITE_RULE, PURE_ONCE_ASM_REWRITE_RULE, PURE_REWRITE_RULE, REWRITE_RULE.

ONCE_ASM_REWRITE_TAC

```
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 []
#            (["(a:*) = b"; "(b:*) = c"], "P (a:*): bool") ;;
([(["a = b"; "b = c"], "P b")], -) : subgoals

#(ONCE_ASM_REWRITE_TAC [] THEN ONCE_ASM_REWRITE_TAC [])
#            (["(a:*) = b"; "(b:*) = c"], "P (a:*): bool") ;;
([(["a = b"; "b = c"], "P c")], -) : subgoals
```

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

basic_rewrites, ASM_REWRITE_TAC, FILTER_ASM_REWRITE_TAC, FILTER_ONCE_ASM_REWRITE_TAC, GEN_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_REWRITE_TAC, PURE_ASM_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_ONCE_REWRITE_TAC, PURE_REWRITE_TAC, REWRITE_TAC, SUBST_TAC.

ONCE_DEPTH_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 "(\x. (\y. y + x) 1) 2";;
   |- (\x. (\y. y + x)1)2 = (\y. y + 2)1
```

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 num_CONV "(\x. (\y. y + x) 1) 2";;
   |- (\x. (\y. y + x)1)2 = (\x. (\y. y + x)(SUC 0))(SUC 1)
```

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 failure string `‘QCONV‘` may be generated and later trapped. The behaviour of the function is dependent on this use of failure. So, if the conversion given as argument happens to generate a failure with string `‘QCONV‘`, the operation of `ONCE_DEPTH_CONV` will be unpredictable.

See also

`DEPTH_CONV, REDEPTH_CONV, TOP_DEPTH_CONV.`

ONCE_REWRITE_CONV

```
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
GEN_REWRITE_CONV, PURE_ONCE_REWRITE_CONV, PURE_REWRITE_CONV, REWRITE_CONV.

ONCE_REWRITE_RULE

```
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
ASM_REWRITE_RULE, GEN_REWRITE_RULE, ONCE_ASM_REWRITE_RULE, PURE_ONCE_REWRITE_RULE, PURE_REWRITE_RULE, REWRITE_RULE.

ONCE_REWRITE_TAC

```
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:

```
th1 = [ |- a = b; |- b = c; |- c = a]
```

the tactic `ONCE_REWRITE_TAC thl` can be iterated as required without diverging:

```
#ONCE_REWRITE_TAC thl ([], "P a");;
([([], "P b")], -) : subgoals

#(ONCE_REWRITE_TAC thl THEN ONCE_REWRITE_TAC thl) ([], "P a");;
([([], "P c")], -) : subgoals

#(ONCE_REWRITE_TAC thl THEN ONCE_REWRITE_TAC thl THEN ONCE_REWRITE_TAC thl)
#([], "P a");;
([([], "P a")], -) : subgoals
```

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
`ASM_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `PURE_ASM_REWRITE_TAC`, `PURE_ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

ORELSE

```
$ORELSE : (tactic -> tactic -> tactic)
```

Synopsis
Applies first tactic, and iff 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 iff 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
`EVERY, FIRST, THEN.`

ORELSEC

```
$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
`FIRST_CONV.`

ORELSE_TCL

```
$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
`EVERY_TCL, FIRST_TCL, THEN_TCL.`

OR_EXISTS_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
`EXISTS_OR_CONV, LEFT_OR_EXISTS_CONV, RIGHT_OR_EXISTS_CONV.`

OR_FORALL_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
`FORALL_OR_CONV`, `LEFT_OR_FORALL_CONV`, `RIGHT_OR_FORALL_CONV`.

overload_on

```
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.

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.thm
```

Then we can set up the overloading we want. One of the possible meanings for the string /\ is usual logical conjunction:

```
- overload_on ("/\\", Term‘$/\‘);
> val it = () : unit
```

We also 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.term
- type_of it;
> val it = ‘:’a -> bool‘ : Type.hol_type
```

In order to make normal conjunction the preferred choice, we can repeat the call to `overload_on`:

```
- overload_on ("/\\", Term‘$/\ :bool -> bool -> bool‘);
> val it = () : unit
- Term‘p /\ q‘;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = ‘p /\ q‘ : Term.term
- type_of it;
> val it = ‘:bool‘ : Type.hol_type
```

Note that in order to override the overloading, we had to specify the type of the logical conjunction explicitly. Otherwise, the parsing would have taken the /\ to mean our previously preferred alternative (`inter`).

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

`clear_overloads_on`.

p

```
p : (int -> void)
```

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

pair

```
pair : (* -> ** -> (* # **))
```

Synopsis
Makes two values into a pair.

Description
`pair x y` returns `(x,y)`.

Failure
Never fails.

See also
`fst, snd, curry, uncurry.`

PAIRED_BETA_CONV

```
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`.

Example

The following is a typical use of the conversion:

```
#PAIRED_BETA_CONV "(\((a,b),(c,d)). a + b + c + d)  ((1,2),(3,4))";;
|- (\((a,b),c,d). a + (b + (c + d)))((1,2),3,4) = 1 + (2 + (3 + 4))
```

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:

```
#PAIRED_BETA_CONV "(\((a,b),p). a + b)  ((1,2),(3+5,4))";;
|- (\((a,b),p). a + b)((1,2),3 + 5,4) = 1 + 2
```

but the following call to `PAIRED_BETA_CONV` fails:

```
#PAIRED_BETA_CONV "(\((a,b),(c,d)). a + b + c + d)  ((1,2),p)";;
evaluation failed     PAIRED_BETA_CONV
```

because `p` is not a pair.

See also

`BETA_CONV, BETA_RULE, BETA_TAC, LIST_BETA_CONV, RIGHT_BETA, RIGHT_LIST_BETA.`

PAIRED_ETA_CONV

```
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. Disabling the relevant prettyprinting reveals the true form of a paired eta redex:

```
   #set_flag(`print_uncurry`,false);;
   true : bool

   #let tm = "\(x:num,y:num). FST (x,y)";;
   tm = "UNCURRY(\x y. FST(x,y))" : term
```

Example

The following is a typical use of the conversion:

```
   let SELECT_PAIR_EQ = PROVE
    ("(@(x:*,y:**). (a,b) = (x,y)) = (a,b)",
     CONV_TAC (ONCE_DEPTH_CONV PAIRED_ETA_CONV) THEN
     ACCEPT_TAC (SYM (MATCH_MP SELECT_AX (REFL "(a:*,b:**)"))));;
```

See also

`ETA_CONV`.

`parents`

```
parents : (string -> string list)
```

Synopsis

Lists the parent theories of a named theory.

Description

The function `parents` returns a list of strings that identify the parent theories of a named theory. The function does not recursively descend the theory hierarchy in search of the 'leaf' theories. The named theory must be the current theory or an ancestor of the current theory.

Failure

Fails if the named theory is not an ancestor of the current theory.

Example

Initially, the only parent is the main HOL theory:

```
#new_theory `my-theory`;;
() : void

#parents `my-theory`;;
[`HOL`] : string list

#parents `HOL`;;
[`tydefs`; `sum`; `one`; `BASIC-HOL`] : string list

#parents `tydefs`;;
[`ltree`; `BASIC-HOL`] : string list

#parents `string`;;
evaluation failed     parents -- string is not an ancestor
```

However, loading the string library creates several additional ancestor theories:

```
#load_library `string`;;
Loading library `string` ...
Updating search path
.Updating help search path
.Declaring theory string a new parent
Theory string loaded
......
Library `string` loaded.
() : void

#parents `string`;;
[`ascii`; `HOL`] : string list

#parents `my-theory`;;
[`string`; `HOL`] : string list
```

See also

ancestors, ancestry.

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.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 : Type.hol_type frag list -> Type.hol_type
  val Term = fn : Term.term frag list -> Term.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.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.term
- Parse.Term‘2 + 3‘;
> val it = ‘2 + 3‘ : Term.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
`add_rule, Term, Type`

parse_in_context

```
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

`Term`

parse_preTerm

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

Synopsis

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

Description

The "let" expression `‘let x = e1 in e2‘` will turn 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 provided includes a syntax error. This phase of parsing is unconcerned with types, and will happily parse meaningless expressions that are syntactically valid.

Example

Uses

Comments

See also

PART_MATCH

```
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
`INST_TYPE`, `INST_TY_TERM`, `match`.

PAT_ASSUM

```
Ho_tactics.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 matching in the sense of `Ho_match.match_ter`
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 ``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

`ASSUM_LIST`, `EVERY`, `PAT_ASSUM`, `EVERY_ASSUM`, `FIRST`, `MAP_EVERY`, `MAP_FIRST`, `UNDISCH_THEN`, `match_term`.

POP_ASSUM

```
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 (\th. REWRITE_TAC[REWRITE_RULE[num_CONV "4"; INV_SUC_EQ] th]))
```

Uses

Making more delicate use of an assumption than rewriting or resolution using it.

See also

`ASSUM_LIST, EVERY_ASSUM, IMP_RES_TAC, POP_ASSUM_LIST, REWRITE_TAC.`

POP_ASSUM_LIST

```
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

ASSUM_LIST, EVERY_ASSUM, IMP_RES_TAC, POP_ASSUM, REWRITE_TAC.

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.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
`clear_prefs_for_term`

print_term

```
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
`term_to_string`

prove

```
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
`PROVE, prove_thm, TAC_PROOF, VALID.`

PROVE

```
Compat.PROVE : (term * tactic) -> thm
```

Synopsis
Attempts to prove a boolean term using the supplied tactic.

Description
Found in the hol88 library. 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.

Comments
In hol90, use `prove` instead; in hol90 `PROVE` has been replaced by `prove` and `prove_thm` has been replaced by `store_thm`.

See also
`TAC_PROOF, prove, prove_thm, VALID.`

prove_abs_fn_one_one

```
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

`new_type_definition`, `define_new_type_bijections`, `prove_abs_fn_onto`, `prove_rep_fn_one_one`, `prove_rep_fn_onto`.

prove_abs_fn_onto

```
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

`new_type_definition`, `define_new_type_bijections`, `prove_abs_fn_one_one`, `prove_rep_fn_one_one`, `prove_rep_fn_onto`.

prove_cases_thm

```
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
`define_type`, `INDUCT_THEN`, `new_recursive_definition`, `prove_constructors_distinct`, `prove_constructors_one_one`, `prove_induction_thm`, `prove_rec_fn_exists`.

prove_constructors_distinct

```
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

`define_type`, `INDUCT_THEN`, `new_recursive_definition`, `prove_cases_thm`, `prove_constructors_one_one`, `prove_induction_thm`, `prove_rec_fn_exists`.

prove_constructors_one_one

```
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

`define_type`, `INDUCT_THEN`, `new_recursive_definition`, `prove_cases_thm`, `prove_constructors_distinct`, `prove_induction_thm`, `prove_rec_fn_exists`.

PROVE_HYP

```
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

`DISCH, MP, UNDISCH.`

prove_induction_thm

```
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

`define_type, INDUCT_THEN, new_recursive_definition, prove_cases_thm, prove_constructors_distinct, prove_constructors_one_one, prove_rec_fn_exists.`

prove_rec_fn_exists

```
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.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

`prove_rec_fn_exists` can be used to prove the existence of primitive recursive functions over binary trees. Suppose the value of `th` is this theorem. Then the existence of a recursive function `Leaves`, which computes the number of leaves in a binary tree, can be proved as shown below:

```
#prove_rec_fn_exists th
#  "(Leaves (LEAF (x:*)) = 1) /\
#   (Leaves (NODE t1 t2) = (Leaves t1) + (Leaves t2))";;
|- ?Leaves. (!x. Leaves(LEAF x) = 1) /\
            (!t1 t2. Leaves(NODE t1 t2) = (Leaves t1) + (Leaves t2))
```

The result should be compared with the example given under `new_recursive_definition`.

See also

`define_type`, `new_recursive_definition`.

prove_rep_fn_one_one

```
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
`new_type_definition`, `define_new_type_bijections`, `prove_abs_fn_one_one`, `prove_abs_fn_onto`, `prove_rep_fn_onto`.

prove_rep_fn_onto

```
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
`new_type_definition`, `define_new_type_bijections`, `prove_abs_fn_one_one`, `prove_abs_fn_onto`, `prove_rep_fn_one_one`.

prove_thm

```
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
`prove, PROVE, TAC_PROOF, VALID.`

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

```
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
`ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `PURE_ONCE_ASM_REWRITE_RULE`.

PURE_ASM_REWRITE_TAC

```
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
`ASM_REWRITE_TAC`, `GEN_REWRITE_TAC`, `FILTER_ASM_REWRITE_TAC`, `FILTER_ONCE_ASM_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_ONCE_ASM_REWRITE_TAC`, `PURE_ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

PURE_ONCE_ASM_REWRITE_RULE

```
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
`ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_ASM_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_ASM_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWRITE_RULE`.

PURE_ONCE_ASM_REWRITE_TAC

```
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
`ASM_REWRITE_TAC`, `GEN_REWRITE_TAC`, `FILTER_ASM_REWRITE_TAC`, `FILTER_ONCE_ASM_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_ASM_REWRITE_TAC`, `PURE_ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

PURE_ONCE_REWRITE_CONV

```
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
`GEN_REWRITE_CONV`, `ONCE_DEPTH_CONV`, `ONCE_REWRITE_CONV`, `PURE_REWRITE_CONV`, `REWRITE_CONV`.

PURE_ONCE_REWRITE_RULE

```
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
`ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_DEPTH_CONV`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWRITE_RULE`.

PURE_ONCE_REWRITE_TAC

```
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
`ASM_REWRITE_TAC`, `GEN_REWRITE_TAC`, `FILTER_ASM_REWRITE_TAC`, `FILTER_ONCE_ASM_REWRITE_TAC`, `ONCE_ASM_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `PURE_ASM_REWRITE_TAC`, `PURE_ONCE_ASM_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_TAC`.

PURE_REWRITE_CONV

```
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
`GEN_REWRITE_CONV`, `ONCE_REWRITE_CONV`, `PURE_ONCE_REWRITE_CONV`, `REWRITE_CONV`.

PURE_REWRITE_RULE

```
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
`ASM_REWRITE_RULE`, `GEN_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_ASM_REWRITE_RULE`, `PURE_ONCE_ASM_REWRITE_RULE`, `PURE_ONCE_REWRITE_RULE`, `REWRITE_RULE`.

PURE_REWRITE_TAC

```
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[]([],"b = T");;
([([], "b = T")], -) : subgoals

#REWRITE_TAC[]([],"b = T");;
([([], "b")], -) : subgoals
```

See also
`ASM_REWRITE_TAC, FILTER_ASM_REWRITE_TAC, FILTER_ONCE_ASM_REWRITE_TAC, GEN_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_REWRITE_TAC, PURE_ASM_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_ONCE_REWRITE_TAC, REWRITE_TAC, SUBST_TAC.`

pure_ss

```
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 `PURE_ss ssdata` value in the same `pureSimps` structure that I can't be bothered giving its own special manual entry. It plausibly doesn't need to be there at all.

See also

`bool_ss, hol_ss, PURE_REWRITE_TAC, SIMP_CONV, SIMP_TAC`

r

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

rand

```
rand : (term -> term)
```

Synopsis
Returns the operand from a combination (function application).

Description
`rand "t1 t2"` returns `"t2"`.

Failure
Fails with `rand` if term is not a combination.

See also
`rator, dest_comb.`

RAND_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 num_CONV "SUC 2";;
|- SUC 2 = SUC(SUC 1)
```

See also
`ABS_CONV, BINOP_CONV, LAND_CONV, RATOR_CONV, SUB_CONV.`

rator

```
rator : (term -> term)
```

Synopsis
Returns the operator from a combination (function application).

Description
`rator("t1 t2")` returns `"t1"`.

Failure
Fails with `rator` if term is not a combination.

See also
`rand, dest_comb.`

RATOR_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 "(\x y. x + y) 1 2";;
|- (\x y. x + y)1 2 = (\y. 1 + y)2
```

See also

`ABS_CONV`, `RAND_CONV`, `SUB_CONV`.

REDEPTH_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 "(\f x. (f x) + 1) (\y.y) 2";;
|- (\f x. (f x) + 1)(\y. y)2 = 2 + 1
```

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 failure string ‘QCONV‘ may be generated and later trapped. The behaviour of the function is dependent on this use of failure. So, if the conversion given as argument happens to generate a failure with string ‘QCONV‘, the operation of REDEPTH_CONV will be unpredictable.

See also

DEPTH_CONV, ONCE_DEPTH_CONV, TOP_DEPTH_CONV.

REFINE_EXISTS_TAC

```
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
EXISTS_TAC.

REFL

```
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
ALL_CONV, REFL_TAC.

REFL_TAC

```
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
`ACCEPT_TAC`, `MATCH_ACCEPT_TAC`, `REWRITE_TAC`.

doc.

register_trace.doc

```
register_trace.doc : string -> int ref -> unit
```

Synopsis
Registers a new tracing variable.

Description
A call to `register_trace n r` registers the integer reference variable `r` as a tracing variable associated with name `n`. 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.

See also
`current_trace`, `reset_trace`, `reset_traces`, `trace`, `traces`.

remove_rules_for_term

```
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.term
- remove_rules_for_term "!";
> val it = () : unit
- t;
> val it = ‘! (\x. P x /\ ~Q x)‘ : Term.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.term
- remove_rules_for_term "COND";
> val it = () : unit
- t;
> val it = ‘COND p q r‘ : Term.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

`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
- Term‘p => q | r‘;
<<HOL message: inventing new type variable names: ’a.>>
> val it = ‘COND p q r‘ : Term.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.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
`remove_rules_for_term`, `prefer_form_with_tok`

REPEAT

```
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
`EVERY`, `FIRST`, `ORELSE`, `THEN`, `THENL`.

REPEATC

```
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+1`th term `"t(n+1)"`, then `REPEATC c "t1"` returns `|- t1 = t(n+1)`. And if `c "t"` fails, them `REPEATC c "t"` returns `|- t = t`.

Failure
Never fails, but can diverge if the supplied conversion never fails.

REPEAT_GTCL

```
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
`REPEAT_TCL, THEN_TCL.`

REPEAT_TCL

```
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
`REPEAT_GTCL`, `THEN_TCL`.

reset_trace

```
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 it had when it was registered.

Failure
Fails if the name given is not associated with a registered tracing variable.

See also
`current_trace`, `register_trace`, `reset_traces`, `trace`, `traces`.

reset_traces

```
.reset_traces : unit -> unit
```

Synopsis
Resets all registered tracing variables to their default values.

Failure
Never fails.

See also
`current_trace`, `register_trace`, `reset_trace`, `trace`, `traces`.

RES_CANON

```
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 implication in both directions:

```
     A |- t1 /\ t2                          A |- t1 = t2
  --------------------           ----------------------------------
   A |- t1    A |- t2             A |- t1 ==> t2    A |- t2 ==> t1
```

Conjunctive antecedents are transformed by:

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

and disjunctive antecedents by:

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

The scope of universal quantifiers is restricted, if possible:

```
   A |- !x. t1 ==> t2
  --------------------         [if x is not free in t1]
   A |- t1 ==> !x. t2
```

and existentially-quantified antecedents are eliminated by:

```
     A |- (?x. t1) ==> t2
  ---------------------------  [x' chosen so as not to be free in t2]
   A |- !x'. t1[x'/x] ==> t2
```

Finally, when no further applications of the above rules are possible, and the theorem is an implication:

```
  A |- !x1...xn. t1 ==> t2
```

then the theorem `A u {t1} |- t2` is transformed by a recursive application of `RES_CANON` to get a list of theorems:

```
  [A u {t1} |- t21 ; ... ; A u {t1} |- t2n]
```

and the result of discharging `t1` from these theorems:

```
  [A |- !x1...xn. t1 ==> t21 ; ... ; A |- !x1...xn. t1 ==> t2n]
```

is returned. That is, the transformation rules are recursively applied to the conclusions of all implications.

Failure
RES_CANON th fails if no implication(s) can be derived from th using the transformation rules shown above.

Example
The uniqueness of the remainder k MOD n is expressed in HOL by the built-in theorem MOD_UNIQUE:

```
|- !n k r. (?q. (k = (q * n) + r) /\ r < n) ==> (k MOD n = r)
```

For this theorem, the canonical list of implications returned by RES_CANON is as follows:

```
#RES_CANON MOD_UNIQUE;;
[|- !k q n r. (k = (q * n) + r) ==> r < n ==> (k MOD n = r);
 |- !r n. r < n ==> (!k q. (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
IMP_RES_TAC, IMP_RES_THEN, RES_TAC, RES_THEN.

RES_TAC

```
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 `REPEAT`ed. However, since the final resolvents added to the original assumptions are never used as 'candidate antecedents' it is sometimes necessary to apply `RES_TAC` more than once to derive the desired result.

See also

`IMP_RES_TAC, IMP_RES_THEN, RES_CANON, RES_THEN.`

RES_THEN

```
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
`IMP_RES_TAC`, `IMP_RES_THEN`, `MATCH_MP`, `RES_CANON`, `RES_TAC`.

reveal

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

Synopsis
Restores recognition of a constant by the quotation parser.

Description
A call `reveal "c"`, where `c` is a (perhaps) hidden constant, will unhide the constant, that is, will make the quotation parser recognize it as such rather than treating it as a variable. It reverses the effect of the call `Parse.hide "c"`.

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.

See also
`Parse.hide`, `Parse.hidden`.

rev_assoc

```
Compat.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

Found in the hol88 library. `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. The function will not be available if the hol88 library has not been loaded.

Example

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

Comments

Not found in hol90, since we use an option type instead of exceptions.

assoc1; val it = fn : ”a -¿ (”a * ’b) list -¿ (”a * ’b) option - assoc2; val it = fn : ”a -¿ (’b * ”a) list -¿ (’b * ”a) option

See also

`assoc, find, mem, tryfind, exists, forall.`

rev_itlist

```
rev_itlist : ((* -> ** -> **) -> * list -> ** -> **)
```

Synopsis

Applies a binary function between adjacent elements of the reverse of a list.

Description

`rev_itlist f [x1;...;xn] y` returns `f xn ( ... (f x2 (f x1 y))...)`. It returns `y` if the list is empty.

Failure

Never fails.

Example

```
#rev_itlist (\x y. x * y) [1;2;3;4] 1;;
24 : int
```

See also

`itlist, end_itlist.`

rewrites

```
simpLib.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 hol_ss thmlist`, one could write

```
   SIMP_CONV (hol_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
`++`, `mk_simpset`, `SIMPSET`, `SIMP_CONV`.

REWRITE_CONV

```
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`, `GEN_REWRITE_CONV`, `ONCE_REWRITE_CONV`, `PURE_REWRITE_CONV`, `REWR_CONV`, `SUBST_CONV`.

REWRITE_RULE

```
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

`ASM_REWRITE_RULE`, `basic_rewrites`, `GEN_REWRITE_RULE`, `ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWR_CONV`, `REWRITE_CONV`, `SUBST`.

REWRITE_TAC

```
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

ASM_REWRITE_TAC, GEN_REWRITE_TAC, FILTER_ASM_REWRITE_TAC, FILTER_ONCE_ASM_REWRITE_TAC, ONCE_ASM_REWRITE_TAC, ONCE_REWRITE_TAC, PURE_ASM_REWRITE_TAC, PURE_ONCE_ASM_REWRITE_TAC, PURE_ONCE_REWRITE_TAC, PURE_REWRITE_TAC, REWR_CONV, REWRITE_CONV, SIMP_TAC, SUBST_TAC.

REWR_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:*. !y. (x = y) = (y = x)
```

and `REWR_CONV EQ_SYM` behaves as follows:

```
#REWR_CONV EQ_SYM_EQ "1 = 2";;
|- (1 = 2) = (2 = 1)

#REWR_CONV EQ_SYM_EQ "1 < 2";;
evaluation failed     REWR_CONV: lhs of theorem doesn't match term
```

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 "!x:*. f x = x") "f 2:num";;
evaluation failed     REWR_CONV: lhs of theorem doesn't match term
```

The application fails, however, because the type variable `*` appears in the assumption of the theorem returned by `ASSUME "!x:*. f x = x"`.

Failure will also occur in situations like:

```
#REWR_CONV (ASSUME "f (n:num) = n") "f 2:num";;
evaluation failed     REWR_CONV: lhs of theorem doesn't match term
```

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_CONV`.

rhs

```
rhs : (term -> term)
```

Synopsis
Returns the right-hand side of an equation.

Description
`rhs "t1 = t2"` returns `"t2"`.

Failure
Fails with `rhs` if term is not an equality.

See also
`lhs`, `dest_eq`.

RIGHT_AND_EXISTS_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
`AND_EXISTS_CONV`, `EXISTS_AND_CONV`, `LEFT_AND_EXISTS_CONV`.

RIGHT_AND_FORALL_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
`AND_FORALL_CONV`, `FORALL_AND_CONV`, `LEFT_AND_FORALL_CONV`.

RIGHT_BETA

```
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

`BETA_CONV`, `BETA_RULE`, `BETA_TAC`, `RIGHT_LIST_BETA`.

RIGHT_CONV_RULE

```
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_RULE`.

RIGHT_IMP_EXISTS_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
`EXISTS_IMP_CONV`, `LEFT_IMP_FORALL_CONV`.

RIGHT_IMP_FORALL_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
FORALL_IMP_CONV, LEFT_IMP_EXISTS_CONV.

RIGHT_LIST_BETA

```
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
BETA_CONV, BETA_RULE, BETA_TAC, LIST_BETA_CONV, RIGHT_BETA.

RIGHT_OR_EXISTS_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
`OR_EXISTS_CONV`, `EXISTS_OR_CONV`, `LEFT_OR_EXISTS_CONV`.

RIGHT_OR_FORALL_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
`OR_FORALL_CONV`, `FORALL_OR_CONV`, `LEFT_OR_FORALL_CONV`.

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

```
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
`ASSUM_LIST, MAP_EVERY, MAP_FIRST, POP_ASSUM_LIST.`

S

```
S : ((* -> ** -> ***) -> (* -> **) -> * -> ***)
```

Synopsis
Performs function composition: `S f g x` = `f x (g x)` (the `S` combinator).

Failure
Never fails.

See also
`#, B, C, CB, Co, I, K, KI, o, oo, W.`

save_thm

```
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 returned as a value. The call can be made in both proof and draft mode. The name `name` must be a distinct name within the theory segment, but may be the same as for items within other theory segments of the theory. If the current theory segment is named `thy`, the theorem will be written to the file `thy.th` in the directory from which HOL was called. If the system is in draft mode, other changes made to the current theory segment during the session will also be written to the theory file. If the theory file does not exist, it will be created.

Failure
A call to `save_thm` will fail if the name given is the same as the name of an existing fact in the current theory segment. Saving the theorem involves writing to the file system. If the write fails for any reason `save_thm` will fail. For example, on start up the initial theory is `HOL`. The associated theory files are read-only so an attempt to save a theorem in that theory segment will fail.

Uses
Adding theorems to the current theory. Saving theorems for retrieval in later sessions. The theorem may be retrieved using the function `theorem`. Binding the result of `save_thm` to an ML variable makes it easy to access the theorem in the current terminal session.

See also
`new_theory`, `prove_thm`, `save_top_thm`, `theorem`.

SELECT_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 "(@n. n < m) < m";;
|- (@n. n < m) < m = (?n. n < m)
```

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

`SELECT_ELIM, SELECT_INTRO, SELECT_RULE.`

SELECT_ELIM

```
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 ("g:num->num", th2);;
|- ?h. ONE_ONE h /\ ~ONTO h
```

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
CHOOSE, SELECT_AX, SELECT_CONV, SELECT_INTRO, SELECT_RULE.

SELECT_EQ

```
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 "x:num" th;;
   |- (@x. x MOD 2 = 0) = (@x. ?y. x = 2 * y)
```

See also
ABS, AP_TERM, EXISTS_EQ, FORALL_EQ, SELECT_AX, SELECT_CONV, SELECT_ELIM, SELECT_INTRO.

SELECT_INTRO

```
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:

```
#let th2 = SELECT_INTRO th1;;
th2 = |- (\n. m = n)(@n. m = n)
```

This theorem could now be used to derive a further result:

```
#EQ_MP(BETA_CONV(concl th2))th2;;
|- m = (@n. m = n)
```

See also
`EXISTS, SELECT_AX, SELECT_CONV, SELECT_ELIM, SELECT_RULE.`

SELECT_RULE

```
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;;
   |- ONE_ONE(@f. ONE_ONE f /\ ~ONTO f) /\ ~ONTO(@f. ONE_ONE f /\ ~ONTO f)
```

Uses

May be used to introduce an epsilon term to permit rewriting with a constant defined using the epsilon operator.

See also

`CHOOSE, SELECT_AX, SELECT_CONV, SELECT_ELIM, SELECT_INTRO.`

setify

```
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`.

set_backup

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

() : void

#e CONJ_TAC;;
OK..
2 subgoals
"TL[1;2;3] = [2;3]"

"HD[1;2;3] = 1"

() : void

#e (REWRITE_TAC[HD]);;
OK..
goal proved
|- HD[1;2;3] = 1

Previous subproof:
"TL[1;2;3] = [2;3]"

() : void

#b();;
2 subgoals
"TL[1;2;3] = [2;3]"

"HD[1;2;3] = 1"

() : void

#b();;
evaluation failed     backup:  backup list is empty
```

See also

b, backup, e, expand, expandf, g, get_state, p, print_state, r, rotate, save_top_thm, set_goal, set_state, top_goal, top_thm.

set_fixity

```
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 though the application of `set_fixity` may succeed, it may also cause the first subsequent application of the `Term` parsing function to fail. This latter will happen if the new rule causes a conflict in the precedence matrix.

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.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.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.term
- set_fixity "+" (Infixr 501);
> val it = () : unit
- t;
> val it = `(2 + 3) + 4 - 6` : Term.term
- dest_comb (Term`3 - 1 + 2`);
> val it = (`$- 3`, `1 + 2`) : Term.term * 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
`add_rule, add_infix, remove_rules_for_term, remove_termtok`

set_goal

```
set_goal : (goal -> void)
```

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

() : void
```

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.

Comments

A more sophisticated subgoal management package will be implemented in the future.

See also

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

set_base_rewrites

```
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, empty_rewrites, add_rewrites.`

set_known_constants

```
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.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.term
- strip_comb it;
> val it = (`$/\`, [`p`, `q`]) : Term.term * Term.term list
- dest_var (#1 it);
> val it = ("/\\", `:'a -> 'b -> 'c`) : string * Type.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
`hidden, hide, known_constants, reveal.`

show_numeral_types

```
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.term

- show_numeral_types := true;
> val it = () : unit

- Term‘~3‘;
> val it = ‘~3i‘ : Term.term
```

Uses
Can help to disambiguate terms involving numerals.

See also
`add_numeral_form`, `show_types`

show_types

```
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.thm

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

- BOOL_CASES_AX;;
> val it = |- !(t :bool). (t = T) \/ (t = F) : Thm.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

`print_term`.

SIMPSET

```
simpLib.SIMPSET : { ac     : (thm * thm) list,
                    congs  : thm list,
                    convs  : {conv  : (term list -> term -> thm) -> 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 starting 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. It is not known what the role of the list of terms is. However, if it is the list is left as `[]`, the second component of the pair, the bare term is used as a pattern. The conversion will only be applied to sub-terms that match the pattern. 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. We still know too little about how this code works.

See also

`++, bool_ss, mk_simpset, rewrites, SIMP_CONV`

SIMP_CONV

```
simpLib.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 bound up in the the provided simpset 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 sub-terms. (If a conversion is applied to an inappropriate sub-term and fails, this failure is caught by the simplifier and ignored.) Instead, conversions in simpsets are accompanied by a term-pattern which specifies the sort of situations in which they should be applied. This facility is used in the definition of `bool_ss` to include `ETA_CONV`, but stop it from transforming `!x. P x` into `$! P`. 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 (Ho_theorems.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 hol_ss simpset includes an arithmetic decision procedure implemented in this way.

Failure
SIMP_CONV never fails, but may diverge.

Example

```
- SIMP_CONV hol_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.

Comments
This documentation is incomplete, due to a lack of understanding on the author's part of another's code.

See also
++, ASM_SIMP_TAC, FULL_SIMP_TAC, hol_ss, mk_simpset, rewrites, SIMP_RULE, SIMP_TAC, SIMPSET

SIMP_PROVE

```
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
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.

```
- ASSUME_TAC (SIMP_PROVE hol_ss [] ``x < y ==> x < y + 6``)
             ([], ``x + y = 10``)
> val it =
    ([([`x < y ==> x < y + 6`], `x + y = 10`)], fn)
    : tactic_result
```

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
SIMP_CONV, SIMP_RULE, SIMP_TAC.

SIMP_RULE

```
simpLib.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 hol_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
`ASM_SIMP_RULE`, `SIMP_CONV`, `SIMP_TAC`.

SIMP_TAC

```
simpLib.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 hol_ss simpset combine to prove quite difficult seeming goals:

```
- val (_, p) =
    SIMP_TAC hol_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

++, ASM_SIMP_TAC, bool_ss, FULL_SIMP_TAC, hol_ss, mk_simpset, REWRITE_TAC, SIMP_CONV, SIMP_PROVE, SIMP_RULE.

SKOLEM_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
`X_SKOLEM_CONV`.

snd

```
snd : ((* # **) -> **)
```

Synopsis
Extracts the second component of a pair.

Description
`snd (x,y)` returns `y`.

Failure
Never fails.

See also
`fst`, `pair`.

sort

```
sort : (((* # *) -> bool) -> * list -> * list)
```

Synopsis
Sorts a list using a given transitive ‘ordering’ relation.

Description
The call

```
   sort op list
```

where `op` is an (uncurried) transitive relation on the elements of `list`, will topologically sort the list, i.e. will permute it such that if `x op y` but not `y op x` then `x` will occur to the left of `y` in the sorted list. In particular if `op` 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 $< [3; 1; 4; 1; 5; 9; 2; 6; 5; 3; 5; 8; 9; 7; 9];;
[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 ($< o (fst # fst)) [(1,3); (7,11); (3,2); (3,4); (7,2); (5,1)];;
[(1, 3); (3, 4); (3, 2); (5, 1); (7, 2); (7, 11)] : (int # int) list
```

SPEC

```
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 xv = "x:bool" and yv="y:bool" in
   #     (GEN xv o DISCH xv o GEN yv o DISCH yv) (ASSUME xv);;
   |- !x. x ==> (!y. y ==> x)

   #SPEC "~y" it;;
   |- ~y ==> (!y'. y' ==> ~y)
```

See also

ISPEC, SPECL, SPEC_ALL, SPEC_VAR, GEN, GENL, GEN_ALL.

SPECL

```
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 as that of the appropriate quantified variable in the original theorem.

Example

The following is a specialization of a theorem from theory `arithmetic`.

```
#let t = theorem `arithmetic` `LESS_EQ_LESS_EQ_MONO`;;
t = |- !m n p q. m <= p /\ n <= q ==> (m + n) <= (p + q)

#SPECL ["1"; "2"; "3"; "4"] t;;
|- 1 <= 3 /\ 2 <= 4 ==> (1 + 2) <= (3 + 4)
```

See also

`GEN, GENL, GEN_ALL, GEN_TAC, SPEC, SPEC_ALL, SPEC_TAC.`

SPEC_ALL

```
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

The following example shows how variables are also renamed to avoid clashing with the names of constants.

```
#let v=mk_var(`T`,":bool") in ASSUME "!^v. ^v \/ ~^v";;
!T. T \/ ~T |- !T. T \/ ~T

#SPEC_ALL it;;
!T. T \/ ~T |- T' \/ ~T'
```

See also
GEN, GENL, GEN_ALL, GEN_TAC, SPEC, SPECL, SPEC_ALL, SPEC_TAC.

SPEC_TAC

```
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
GEN, GENL, GEN_ALL, GEN_TAC, SPEC, SPECL, SPEC_ALL, STRIP_TAC.

SPEC_VAR

```
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
`GEN`, `GENL`, `GEN_ALL`, `GEN_TAC`, `SPEC`, `SPECL`, `SPEC_ALL`.

split

```
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`.

See also
`combine`.

string_of_int

```
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`, `int_of_string`, `int_to_string`, `string_to_int`.

strip_abs

```
strip_abs : (term -> goal)
```

Synopsis
Iteratively breaks apart abstractions.

Description

`strip_abs "\x1 ... xn. t"` 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

`list_mk_abs`, `dest_abs`.

STRIP_ASSUME_TAC

```
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
`ASSUME_TAC`, `CHOOSE_TAC`, `CHOOSE_THEN`, `CONJUNCTS_THEN`, `DISJ_CASES_TAC`, `DISJ_CASES_THEN`.

strip_comb

```
strip_comb : (term -> (term # term list))
```

Synopsis
Iteratively breaks apart combinations (function applications).

Description
`strip_comb "t t1 ... tn"` returns `("t",["t1";...;"tn"])`. Note that

```
   strip_comb(list_mk_comb("t",["t1";...;"tn"]))
```

will not return `("t",["t1";...;"tn"])` if `t` is a combination.

Failure
Never fails.

Example
```
#strip_comb "x /\ y";;
("$/\", ["x"; "y"]) : (term # term list)

#strip_comb "T";;
("T", []) : (term # term list)
```

See also
`list_mk_comb`, `dest_comb`.

strip_exists

```
strip_exists : (term -> goal)
```

Synopsis
Iteratively breaks apart existential quantifications.

Description
`strip_exists "?x1 ... xn. t"` 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
`list_mk_exists`, `dest_exists`.

strip_forall

```
strip_forall : (term -> goal)
```

Synopsis
Iteratively breaks apart universal quantifications.

Description
`strip_forall "!x1 ... xn. t"` 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
`list_mk_forall`, `dest_forall`.

STRIP_GOAL_THEN

```
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

CONJ_TAC, DISCH_THEN, FILTER_STRIP_THEN, GEN_TAC, STRIP_ASSUME_TAC, STRIP_TAC.

`strip_imp`

```
strip_imp : (term -> goal)
```

Synopsis
Iteratively breaks apart implications.

Description
`strip_imp "t1 ==> ( ... (tn ==> t)...)"` 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)";;
(["T ==> F"; "T"], "F") : goal
```

See also
`list_mk_imp, dest_imp.`

`strip_pair`

```
strip_pair : (term -> term list)
```

Synopsis
Iteratively breaks apart tuples.

Description
`strip_pair("(t1,...,tn)")` returns `["t1";...;"tn"]`. A term that is not a tuple is simply returned as the sole element of a list. Note that

```
   strip_pair(list_mk_pair ["t1";...;"tn"])
```

will not return `["t1";...;"tn"]` if `tn` is a pair or a tuple.

Failure
Never fails.

Example

```
#list_mk_pair ["(1,2)";"(3,4)";"(5,6)"];;
"(1,2),(3,4),5,6" : term

#strip_pair it;;
["1,2"; "3,4"; "5"; "6"] : term list

#strip_pair "1";;
["1"] : term list
```

See also
`list_mk_pair`, `dest_pair`.

STRIP_TAC

```
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

`CONJ_TAC, DISCH_TAC, DISCH_THEN, GEN_TAC, STRIP_ASSUME_TAC, STRIP_GOAL_THEN.`

STRIP_THM_THEN

```
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
CHOOSE_THEN, CONJUNCTS_THEN, DISJ_CASES_THEN, STRIP_ASSUME_TAC.

STRUCT_CASES_TAC

```
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

`ASM_CASES_TAC`, `BOOL_CASES_TAC`, `COND_CASES_TAC`, `DISJ_CASES_TAC`.

SUBGOAL_THEN

```
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 with `‘ASSUME‘` 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 top 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 "~(0 = n)" (\th:thm. 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

```
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

```
#let thm1 = SPECL ["m:num"; "n:num"] ADD_SYM
#and thm2 = CONJUNCT1 ADD_CLAUSES;;
thm1 = |- m + n = n + m
thm2 = |- 0 + m = m
```

depending on the occurrence of free subterms

```
#SUBS [thm1; thm2] (ASSUME "(n + 0) + (0 + m) = m + n");;
. |- (n + 0) + m = n + m

#SUBS [thm1; thm2] (ASSUME "!n. (n + 0) + (0 + m) = m + n");;
. |- !n. (n + 0) + m = m + n
```

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

`ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWRITE_RULE`, `SUBST`, `SUBST_MATCH`, `SUBS_OCCS`.

subst

```
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 substitute 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 [``SUC 0`` |-> ``1``]   ``SUC(SUC 0)``;
> val it = ``SUC 1`` : term


- subst [``SUC 0`` |-> ``1``, ``SUC 1`` |-> ``2``]   ``SUC(SUC 0)``;
> val it = ``SUC 1`` : term


- subst [``SUC 0`` |-> ``1``, ``SUC 1`` |-> ``2``]
        ``SUC(SUC 0) = SUC 1``;
> val it = ``SUC 1 = 2`` : term


- subst [``b:num`` |-> ``a:num``]  ``\a:num. (b:num)``;
> val it = ``\a'. a`` : term


- subst[``flip:'a`` |-> ``foo:'a``]  ``waddle:'a``
> val it = ``waddle`` : term
```

SUBST

```
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[ui]
```

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.

`SUBST` is a complex primitive because it performs both parallel simultaneous substitution and renaming of variables. This is for efficiency reasons, but it would be logically cleaner if `SUBST` were simpler.

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
```

Uses

For substituting at selected occurrences. Often useful for writing special purpose derived inference rules.

See also

SUBS.

SUBST1_TAC

```
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

`ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST_ALL_TAC`, `SUBST_TAC`.

SUBST_ALL_TAC

```
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

ONCE_REWRITE_TAC, PURE_REWRITE_TAC, REWRITE_TAC, SUBST1_TAC, SUBST_TAC.

SUBST_CONV

```
SUBST_CONV : {var :term, thm :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 `[{var=x1, thm = A1|-t1=v1},...,{var = xn, thm = An|-`
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 [{var = x1, thm = A1|-t1=v1},...,{var = xn, thm = 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 [{var=x1,thm=th1},...,{var=xn,thm=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 (--`0`--) ADD1
= and thm1 = SPEC (--`1`--) ADD1
= and x = --`x:num`-- and y = --`y:num`--;
thm0 = |- SUC 0 = 0 + 1
thm1 = |- SUC 1 = 1 + 1
val x = (--`(x :num)`--) : term
val y = (--`(y :num)`--) : term
```

can be used to substitute selected occurrences of the terms `SUC 0` and `SUC 1`

```
- SUBST_CONV [{var=x, thm=thm0},{var=y,thm=thm1}]
=           (--`(x + y) > SUC 1`--)
=           (--`(SUC 0 + SUC 1) > SUC 1`--);
val it = |- SUC 0 + SUC 1 > SUC 1 = (0 + 1) + 1 + 1 > SUC 1 : thm
```

Uses
`SUBST_CONV` is used when substituting at selected occurrences of terms and using rewriting rules/conversions is too extensive.

See also
`REWR_CONV, SUBS, SUBST, SUBS_OCCS.`

SUBST_MATCH

```
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

```
#let thm1 = SPECL ["m:num"; "n:num"] ADD_SYM;;
thm1 = |- m + n = n + m
```

is used to apply substitutions, depending on the occurrence of free instances

```
#SUBST_MATCH thm1 (ASSUME "(n + 1) + (m - 1) = m + n");;
. |- (m - 1) + (n + 1) = m + n

#SUBST_MATCH thm1 (ASSUME "!n. (n + 1) + (m - 1) = m + n");;
. |- !n. (n + 1) + (m - 1) = m + n
```

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

`ONCE_REWRITE_RULE, PURE_REWRITE_RULE, REWRITE_RULE, SUBS, SUBST.`

subst_occs

```
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]] [{redex = --‘SUC 0‘--, residue = --‘1‘--}]
=            (--‘SUC 0 + SUC 0 = SUC(SUC 0)‘--);
val it = (--‘1 + SUC 0 = SUC 1‘--) : term

- subst_occs [[1],[1]] [{redex = --‘SUC 0‘--, residue = --‘1‘--},
=                       {redex = --‘SUC 1‘--, residue = --‘2‘--}]
=            (--‘SUC(SUC 0) = SUC 1‘--);
val it = (--‘SUC 1 = 2‘--) : term

- subst_occs [[1],[1]] [{redex = --‘SUC(SUC 0)‘--, residue = --‘2‘--},
=                       {redex = --‘SUC 0‘--, residue = --‘1‘--}]
=            (--‘SUC(SUC 0) = SUC 0‘--);
val it = (--‘2 = 1‘--) : term
```

See also

subst

SUBST_OCCS_TAC

```
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 ["m:num"; "n:num"] 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 rewriting tactics such as `REWRITE_TAC`, `PURE_REWRITE_TAC`, `ONCE_REWRITE_TAC`, `SUBST_TAC`, etc. are too extensive or would diverge.

See also

`ONCE_REWRITE_TAC`, `PURE_REWRITE_TAC`, `REWRITE_TAC`, `SUBST1_TAC`, `SUBST_TAC`.

SUBST_TAC

```
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

```
#let thm1 = SPECL ["m:num"; "n:num"] ADD_SYM
#and thm2 = CONJUNCT1 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

`ONCE_REWRITE_TAC, PURE_REWRITE_TAC, REWRITE_TAC, SUBST1_TAC, SUBST_ALL_TAC.`

SUBS_OCCS

```
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

```
#let thm = SPECL ["m:num"; "n:num"] ADD_SYM;;
thm = |- m + n = n + m
```

can be used for substituting only the second occurrence of the subterm `m + n`

```
#SUBS_OCCS [([2],thm)] (ASSUME "(n + m) + (m + n) = (m + n) + (m + n)");;
. |- (n + m) + (m + n) = (n + m) + (m + n)
```

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

`ONCE_REWRITE_RULE`, `PURE_REWRITE_RULE`, `REWRITE_RULE`, `SUBS`, `SUBST`, `SUBST_MATCH`.

subtract

```
subtract : (* list -> * list -> * list)
```

Synopsis
Computes the set-theoretic difference of two ‘sets’.

Description
`subtract l1 l2` returns a list consisting of those elements of `l1` that do not appear in `l2`.

Failure
Never fails.

Example

```
#subtract [1;2;3] [3;5;4;1];;
[2] : int list

#subtract [1;2;4;1] [4;5];;
[1; 2; 1] : int list
```

See also
`setify, set_equal, union, intersect.`

SUB_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

`ABS_CONV, RAND_CONV, RATOR_CONV.`

SWAP_EXISTS_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`.

SYM

```
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
`GSYM, NOT_EQ_SYM, REFL.`

SYM_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
`SYM.`

TAC_PROOF

```
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
`PROVE`, `prove_thm`, `VALID`.

Term

```
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 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 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 `parse_preTerm`, 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, overloaded constants or genuine variables. 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 second pre-term 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 driven in most respects 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_preTerm, preTerm, Type, overload_on, guessing_overloads, guessing_tyvars, term_grammar.`

term_grammar

```
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_from_grammars, Term`

term_lt

```
term_lt : term -> term -> unit
```

Synopsis
A total ordering function on terms.

Description
`term_lt` tells whether one term is less than another in the ordering.

Failure
Never fails.

Example

```
    - term_lt (--`\x.x = T`--) (--`3 + 4`--)
    val it = false : bool
```

Comments
If `not (term_lt tm1 tm2)` and `not (term_lt tm2 tm1)`, then `tm1 = tm2`, although it is faster to directly test for equality. Ordering of terms may be useful in implementing search trees and the like.

See also
`type_lt`

term_to_string

```
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
`print_term`

THEN

```
$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
`EVERY`, `ORELSE`, `THENL`.

THENC

`$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
`EVERY_CONV`.

THENL

`$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
`EVERY, ORELSE, THEN.`

THEN_TCL

```
$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
`EVERY_TCL`, `FIRST_TCL`, `ORELSE_TCL`.

thm_count

```
thm_count : (void -> 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`.

TOP_DEPTH_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 failure string `‘QCONV‘` may be generated and later trapped. The behaviour of the function is dependent on this use of failure. So, if the conversion given as argument happens to generate a failure with string `‘QCONV‘`, the operation of `TOP_DEPTH_CONV` will be unpredictable.

See also

`DEPTH_CONV, ONCE_DEPTH_CONV, REDEPTH_CONV.`

top_goal

```
top_goal : (void -> goal)
```

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

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

top_thm

```
top_thm : (void -> 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
`b, backup, backup_limit, e, expand, expandf, g, get_state, p, print_state, r, rotate, save_top_thm, set_goal, set_state, top_goal.`

trace

```
trace : string -> int -> unit
```

Synopsis
Sets a tracing variable to a new value.

Description
The `trace` function is used to set the value of a tracing variable. These variable control the verboseness of various tools within the system. The higher these numbers, the more verbose the tools become. 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).

A call to `trace nm v` attempts to set the tracing variable associated with the string `nm` to value `v`. A tracing variable is associated with a string by registering it using the function `register_trace`.

Failure
Fails if the name given is not associated with a registered tracing variable.

Example

```
- REWRITE_CONV [] (Term‘p /\ T /\ q‘);
> val it = |- p /\ T /\ q = p /\ q : Thm.thm
- trace "rewrite" 1;
> val it = () : unit
- REWRITE_CONV [] (Term‘p /\ T /\ q‘);
<<HOL message: Rewrite:
|- T /\ q = q.>>
> val it = |- p /\ T /\ q = p /\ q : Thm.thm
```

See also

`current_trace, register_trace, reset_trace, reset_traces, traces.`

traces

```
traces : unit -> {name : string, current_value : int, default_value : int} list
```

Synopsis

Returns a list of registered tracing variables.

Failure

Never fails.

See also

`current_trace, register_trace, reset_trace, reset_traces, trace.`

TRANS

```
$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`. Note that `TRANS` can also be used as a infix (see example below).

```
    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

The following shows identical uses of `TRANS`, one as a prefix, one an infix.

```
   #let t1 = ASSUME "a:bool = b" and t2 = ASSUME "b:bool = c";;
   t1 = . |- a = b
   t2 = . |- b = c

   #TRANS t1 t2;;
   .. |- a = c

   #t1 TRANS t2;;
   .. |- a = c
```

See also

`EQ_MP, IMP_TRANS, REFL, SYM.`

TRY

```
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
`CHANGED_TAC`, `VALID`.

tryfind

```
tryfind : ((* -> **) -> * list -> **)
```

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` succeeds.

Failure
Fails with `tryfind` if the application of the function fails for all elements in the list. This will always be the case if the list is empty.

See also
`find`, `mem`, `exists`, `forall`, `assoc`, `rev_assoc`.

TRY_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

`ALL_CONV`.

`types`

```
types : string -> {Arity : int, Name : string} 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.

Example

The theory HOL has no types declared:

```
  - types "HOL";
  > val it = [] : (int # string) list
```

but its ancestors have the following types declared:

```
   - itlist union (map types (ancestry "HOL")) [];
   > val it =
       [{Arity = 2, Name = "sum"}, {Arity = 2, Name = "prod"},
        {Arity = 0, Name = "num"}, {Arity = 1, Name = "list"},
        {Arity = 0, Name = "tree"}, {Arity = 1, Name = "ltree"},
        {Arity = 0, Name = "bool"}, {Arity = 0, Name = "ind"},
        {Arity = 2, Name = "fun"}, {Arity = 0, Name = "one"}]
       : {Arity : int, Name : string} list
}
\SEEALSO
ancestors, axioms, constants, definitions, infixes, new_type, new_type_abbrev,
new_type_definition, parents.

\ENDDOC
\DOC{type\_in}

\TYPE {\small\verb%type_in : (type -> term -> bool)%}\egroup

\SYNOPSIS
Determines whether any subterm of a given term has a particular type.

\DESCRIBE
The predicate {\small\verb%type_in%} returns {\small\verb%true%} if a subterm of the second arg
has the type specified by the first argument.

\EXAMPLE
{\par\samepage\setseps\small
\begin{verbatim}
#type_in ":num" "5 = 4 + 1";;
true : bool

#type_in ":bool" "5 = 4 + 1";;
true : bool

#type_in ":(num)list" "SUC 0";;
false : bool
```

See also

find_term, find_terms, type_in_type, type_tyvars.

type_in_type

```
type_in_type : (type -> type -> bool)
```

Synopsis
Determines whether a given type is a subtype of another.

Description
The predicate `type_in_type` returns `true` if the type given as the first argument is a subtype of the second.

Example

```
   #type_in_type ":num" ":num # bool";;
   true : bool

   #type_in_type ":num" ":(num)list";;
   true :bool

   #type_in_type ":bool" ":num + bool";;
   true : bool
```

See also
`find_term, find_terms, type_in`

type_lt

```
type_lt : hol_type -> hol_type -> unit
```

Synopsis
A total ordering function on types.

Description
`type_lt` tells whether one type is less than another in the ordering.

Failure
Never fails.

Example

```
    - type_lt (==`:bool`==) (==`:'a -> 'a`==)
    val it = true : bool
```

Comments

If `not (type_lt ty1 ty2)` and `not (type_lt ty2 ty1)`, then `ty1 = ty2`, although it is faster to directly test for equality. Ordering of types may be useful in implementing search trees and the like.

See also

`term_lt`

type_of

```
type_of : (term -> type)
```

Synopsis

Returns the type of a term.

Failure

Never fails.

Example

```
#type_of "T";;
":bool" : type
```

type_subst

```
type_subst : hol_type subst -> hol_type -> hol_type
```

Synopsis

Instantiates types in a type.

Description
If `theta = [{redex1,residue1},...,{redexn,residuen}]` is a `hol_type subst`, where the `redexi` are the types to be substituted for, and the `residuei` the replacements, and `ty` is a type to instantiate, the call

```
   type_subst theta ty
```

will appropriately instantiate the type `ty`. The instantiations will be performed in parallel. If several of the type instantiations are applicable, the choice is undefined. Each `redexi` ought to be a type variable, but if it isn't, it will never be replaced. Also, it is not necessary that any or all of the types `t1...tn` should in fact appear in `ty`.

Failure
Never fails.

Example

```
  - type_subst [{redex = (==`:'a`==), residue = (==`:bool`==)}]
               (==`:'a # 'b`==);
  > val it = (==`:bool # 'b`==) : hol_type

  - type_subst [{redex = (==`:'a # 'b`==), residue = (==`:num`==)},
                {redex = (==`:'a`==), residue = (==`:bool`==)}]
              (==`:'a # 'b`==);
  > val it = (==`:bool # 'b`==) : hol_type
```

See also
`inst, INST_TYPE.`

type_tyvars

```
type_tyvars : (type -> type list)
```

Synopsis
Determines the type variables of a given type.

Description
The function `type_tyvars` returns a list of type variables used to construct the given type.

Example

```
#type_tyvars ":bool";;
[] : type list

#type_tyvars ":(* -> **) -> (bool # ***) -> (** + num)";;
[":*"; ":**"; ":***"] : type list
```

See also

`type_abbrevs`, `type_in`, `type_in_type`.

tyvars

```
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
`tyvarsl`.

tyvarsl

```
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
`tyvars`.

uncurry

```
uncurry : ((* -> ** -> ***) -> (* # **) -> ***)
```

Synopsis
Converts a function taking two arguments into a function taking a single paired argument.

Description
The application `uncurry f` returns `\(x,y). f x y`, so that

```
   uncurry f (x,y) = f x y
```

Failure
Never fails.

See also
`curry`.

UNDISCH

```
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
`DISCH`, `DISCH_ALL`, `DISCH_TAC`, `DISCH_THEN`, `FILTER_DISCH_TAC`, `FILTER_DISCH_THEN`, `NEG_DISCH`, `STRIP_TAC`, `UNDISCH_ALL`, `UNDISCH_TAC`.

UNDISCH_ALL

```
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
`DISCH, DISCH_ALL, DISCH_TAC, DISCH_THEN, NEG_DISCH, FILTER_DISCH_TAC, FILTER_DISCH_THEN, STRIP_TAC, UNDISCH, UNDISCH_TAC.`

UNDISCH_TAC

```
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

`DISCH`, `DISCH_ALL`, `DISCH_TAC`, `DISCH_THEN`, `NEG_DISCH`, `FILTER_DISCH_TAC`, `FILTER_DISCH_THEN`, `STRIP_TAC`, `UNDISCH`, `UNDISCH_ALL`.

UNDISCH_THEN

```
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

`PAT_ASSUM`, `DISCH`, `DISCH_ALL`, `DISCH_TAC`, `DISCH_THEN`, `NEG_DISCH`, `FILTER_DISCH_TAC`, `FILTER_DISCH_THEN`, `STRIP_TAC`, `UNDISCH`, `UNDISCH_ALL`, `UNDISCH_TAC`.

union

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

Synopsis

Computes the union of two ‘sets’.

Description

If `l1` and `l2` are both sets (a list 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 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 SML/NJ library.

See also

`setify`, `set_equal`, `intersect`, `subtract`.

`variant`

```
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 (non-hidden) 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 ["y:bool"; "z:bool"] "x:bool";;
"x" : term

#variant ["x:bool"; "x':num"; "x'':num"] "x:bool";;
"x'''" : term
```

while the following shows that clashes with the names of constants are also avoided:

```
#variant [] (mk_var(`T`,":bool"));;
"T'" : 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.

Comments

The `hol90` version of `variant` differs from that of `hol88` by failing if asked to rename a constant.

See also

`genvar, hide_constant, Compat.variant (in hol88 library).`

`version`

```
Globals.version : string
```

Synopsis

The version of the `HOL` system being run.

Example

```
- Globals.version;

val it = "Athabasca" : string
```

W

`W : ((* -> * -> **) -> * -> **)`

Synopsis
Duplicates function argument : `W f x` = `f x x`.

Failure
Never fails.

See also
`#, B, C, CB, Co, I, K, KI, o, oo, S.`

words

`words : (string -> string list)`

Synopsis
Splits a string into a list of words.

Description
`words s` splits the string `s` into a list of substrings. Splitting occurs at each sequence of blanks and carriage returns (white space). This white space does not appear in the list of substrings. Leading and trailing white space in the input string is also thrown away.

Failure
Never fails.

Example

```
#words `  the cat  sat on   the mat `;;
[`the`; `cat`; `sat`; `on`; `the`; `mat`] : string list
```

Uses

Useful when wanting to map a function over a list of constant strings. Instead of using `` [`string1`;...;`stringn`] `` one can use:

```
   (words `string1 ... stringn`)
```

See also

words2, word_separators, maptok, explode.

words2

```
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/`;;
[`the`; `cat`; `sat`; `on`; `the`; `mat`] : string list

#words2 `//` `/the/cat//sat/on//the/mat/`;;
[`/the/cat//sat/on//the/mat/`] : string list
```

See also

words, word_separators, explode.

X_CASES_THEN

```
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 [["n:num"];["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

`DISJ_CASES_THENL, X_CASES_THENL, X_CHOOSE_THEN.`

X_CASES_THENL

```
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 [["n:num"];["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

DISJ_CASES_THEN, X_CASES_THEN, X_CHOOSE_THEN.

X_CHOOSE_TAC

```
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 "q:num" th)` giving the subgoal:

```
   {n < m, m = n + q} ?- ?x. m = n + (x + 1)
```

See also

`CHOOSE`, `CHOOSE_THEN`, `X_CHOOSE_THEN`.

X_CHOOSE_THEN

```
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 "q:num" SUBST1_TAC th)` giving the subgoal:

```
{n < m} ?- ?x. n + q = n + (x + 1)
```

See also
`CHOOSE`, `CHOOSE_THEN`, `CONJUNCTS_THEN`, `CONJUNCTS_THEN2`, `DISJ_CASES_THEN`, `DISJ_CASES_THEN2`, `DISJ_CASES_THENL`, `STRIP_THM_THEN`, `X_CHOOSE_TAC`.

X_FUN_EQ_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
`EXT`, `FUN_EQ_CONV`.

X_GEN_TAC

```
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
`FILTER_GEN_TAC`, `GEN`, `GENL`, `GEN_ALL`, `SPEC`, `SPECL`, `SPEC_ALL`, `SPEC_TAC`, `STRIP_TAC`.

X_SKOLEM_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

`SKOLEM_CONV`.

Index