ProofPower

Z REFERENCE MANUAL

PPTex-2.9.1w2.rda.110727

Information on the current status of ProofPower is available on the World-Wide Web, at URL:

`http://www.lemma-one.demon.co.uk/ProofPower/index.html`

This document is published by:

Lemma 1 Ltd.
2nd Floor
31A Chain Street
Reading
Berkshire
UK
RG1 2HX
e-mail: `pp@lemma-one.com`

PPTex-2.9.1w2.rda.110727 13:54:16 27/07/2011

CONTENTS

ABOUT THIS PUBLICATION

0.1 Purpose

This document, one of several making up the user documentation for the ProofPower system, is the reference manual for the system.

0.2 Readership

This document is intended to be consulted by users already acquainted with the basic principles behind ProofPower who need detailed information on the behaviour of specific facilities provided by the system. It is not a tutorial for learning the basic use of the system. A 'keyword in context' index is supplied, which is useful for identifying the full range of facilities of a particular kind, provided that the reader is familiar with the naming conventions adopted in the development of ProofPower.

0.3 Related Publications

A bibliography is given at the end of this document. Publications relating specifically to ProofPower are:

1. ProofPower Tutorial *[4]*, tutorial covering the basic ProofPower system.
2. ProofPower Z Tutorial *[6]*, tutorial covering ProofPower Z support option.
3. ProofPower Installation and Operation *[5]*;
4. ProofPower Document Preparation *[3]*.

0.4 Assumptions

It is assumed that the reader has some prior acqaintance with ProofPower either by attending a course on ProofPower or by reading the tutorial.

0.5 Acknowledgements

ICL gratefully acknowledges its debt to the many researchers (both academic and industrial) who have provided intellectual capital on which ICL has drawn in the development of ProofPower.

We are particularly indebted to Mike Gordon of The University of Cambridge, for his leading role in some of the research on which the development of ProofPower has built, and for his positive attitude towards industrial exploitation of his work.

The ProofPower system is a proof tool for Higher Order Logic which builds upon ideas arising from research carried out at the Universities of Cambridge and Edinburgh, and elsewhere. In particular the logic supported by the system is (at an abstract level) identical to that implemented in the Cambridge HOL system [1], and the paradigm adopted for implementation of proof support for the language follows that adopted by Cambridge HOL, originating with the LCF system developed at Edinburgh [2]. The functional language 'Standard ML' used both for the implementation and as an interactive metalanguage for proof development, originates in work at Edinburgh, and has been developed to its present state by an international group of academic and industrial researchers. The implementation of Standard ML on which ProofPower is based was itself originally implemented by David Matthews at the University of Cambridge, and is now commercially marketed by Abstract Hardware Limited.

The ProofPower system also supports specification and proof in the Z language, developed at the University of Oxford. We are therefore also indebted to the research at Oxford (and elsewhere) which has contributed to the development of the Z language.

UNIX INTERFACES

SML
hol_list [−*c*] [−*d database*[#*theoryname*]] [−*i scripts*] [−*v*] *theory* ...
hol_list [−*d database*[#*theoryname*]] [−*i scripts*] [−*v*]
hol_list [−*c*] [−*d database*] [−*i scripts*] [−*v*] −*a*

Description `hol_list` is used to obtain selected information from a ProofPower-HOL database. It functions in the same manner as `zed_list` except that it uses defaults appropriate to the ProofPower-HOL, and a HOL theory lister.

In the first form of use, where a list of one or more theory names is specified, `hol_list` uses ProofPower-HOL to generate on its standard output listings (in the HOL language using the function *output_theory*) of the indicated theories in a form suitable for processing by `doctex`. Any cache theory (i.e. the theory name is in the list returned by *get_cache_theories*) will be printed with most of the theory detail elided, unless the `-c` option is given.

In the second form, with no list of theory names, `hol_list` lists the names of all the theories in the database whose language is "HOL", in a sorted order, one per line on its standard output channel. The third form, with `-a`, is like the first but causes all of the theories in the database whose language is "HOL" to be listed in a sorted order.

In any of the three forms the program will start a session as if by command `hol` with the supplied `-d` and `-i` arguments (if any), and it is in this environment that the theory listing is done. The output of this startup will be suppressed, including any indication of failure to load the initialisation scripts.

Each theory is, if possible, made current, or at least in scope, when it is listed.

In any form `-v` indicates the log of the preprocessing should also be output.

Errors `hol_list` prints a message and exits (with the value 1) if the database or any of the theories does not exist. The log of the failure is sent to the standard output, the message to the error output.

See Also `pp_list`, `zed_list`, `pp`, `pp_make_database`

SML
hol [−*d database*[#*theoryname*]] [−*i files*] [−*f files* [−*n*|−*s*] [−*v*]] [−− *ml_flags*]
zed [−*d database*[#*theoryname*]] [−*i files*] [−*f files* [−*n*|−*s*] [−*v*]] [−− *ml_flags*]

Description `hol` and `zed` are identical to `pp`. q.v., except that they use default databases *hol* and *zed* respectively, and hence `-d database` is optional.

SML

pp_list [−*c*] −*d database*[#*theory*] [−*i scripts*] [−*l lang*] [−*v*] *theory ...*
pp_list −*d database*[#*theory*] [−*i scripts*] [−*l lang1* [−*l lang2 ...*]] [−*v*]
pp_list [−*c*] −*d database* [−*i scripts*] [−*l lang1* [−*l lang2 ...*]] [−*v*] −*a*

Description `pp_list` is used to obtain selected information from a ProofPower database.

In the first form of use, where a list of one or more theory names is specified, `pp_list` uses ProofPower to generate on its standard output listings of the indicated theories held in the database given by the `-d` option in a form suitable for processing by `doctex`.

If there is no `-l` option then the theory lister used will depend on the language of the theory. If the language is "HOL" then *output_theory* is used. Otherwise it will attempt to use a function named:

<language in lower case>_output_theory

and only if that doesn't exist will it use *output_theory*. All but the first language will be ignored.

If the `-l lang` option is given then it will take the language code of all theories given to be `lang`, and then work as above.

If no `-d` option is given then the function fails.

Any cache theory (i.e. the theory name is in the list returned by *get_cache_theories*) will be printed with most of the theory detail elided, unless the `-c` option is given.

In the second form, with no list of theory names, `pp_list` lists the names of all the theories in the database one per line on its standard output channel in a sorted order. If any `-l` options are given then only theories whose language is one of those listed will be noted.

The third form, with `-a`, is like the first but causes all of the theories in the database to be listed in a sorted order. If any `-l` options are given then only theories whose language is one of those given will be listed, and they will be individually printed according to their own language.

In any of the three forms, the program will start a session as if by command `pp` with the supplied `-d` and `-i` arguments (if any), and it is in this environment that the theory listing is done. The output of this startup will be suppressed, including any indication of failure to load the initialisation scripts.

Each theory is, if possible, made current, or at least in scope, when it is listed.

In any form `-v` indicates the log of the preprocessing should also be output.

Errors `pp_list` prints a message and exits (with the value 1) if the database or any of the theories does not exist. The log of the failure is sent to the standard output, the message to the error output.

See Also `zed_list`, `hol_list`, `pp`, `pp_make_database`

SML

pp_make_database \
    [−c][−v] [−f] [−p *parentdatabase*[#*parenttheory*]] *newdatabase*[#*cachetheory*]

Description `pp_make_database` makes a new child database to contain ProofPower theories. The new database initially contains a single theory, called the *cache theory* for the database, with name given by `cachetheory` (which is used by certain system functions to cache various definitions and theorems and which is used as the initial current theory when the database is used by the `pp`, `hol` and `zed` commands). If `cachetheory` is omitted then the database name, prefixed by "cache' " is taken to be the same as the name of the new cache theory.

The `-p` option may be used to indicate the database which is to be the parent of the new database and to indicate which theory in it is to be the parent of the theory `cachetheory`. The parent theory is taken to be the cache theory for the parent database if it is not given explicitly.

For portability, the parent database name should normally be given without any architecture- or compiler-specific prefixes or suffixes. Any such prefixes or suffixes will be added automatically by `pp_make_database`, If the resulting file name is an absolute path name (i.e., starts with a '/' character), then that is used as the parent database file name. If the resulting file name is not an absolute path name, `pp_make_database` looks for the parent database file first in the current directory and then in the user's search path (given in the environment variable `$PATH`).

If the `-p` option is not supplied then the database `hol` supplied with the system is used as the parent database, and the parent theory is the theory *hol*. This is an appropriate default for a ProofPower-HOL child database. An appropriate value for ProofPower-Z might be the database `zed` supplied with the system.

In interactive use, `pp_make_database` will normally ask for confirmation before overwriting the database if it already exists. The `-f` (force) option may be used to suppress the request for confirmation before overwriting an existing database.

The `-v` option produces more output which may be useful for diagnostic purposes.

Under Poly/ML, databases are subject to an adjustable size limit. By default, `pp_make_database` will adjust the size limit of the parent database to the minimum possible and adjust the size limit of the child database to the maximum allowed. The `-c` option suppresses these adjustments.

The supplied child database name will be used to create the child database file name which is derived using an algorithm specific to the Standard ML compiler being used.

Errors `pp_make_database` prints a message and exits (with value 1) if the parent database or theory does not exist, if the new database cannot be created or if the name of the cache theory clashes with the name of a theory in the parent database.

Some systems impose a limit on the depth of nesting of the database hierarchy and the command will print an error message and exit (with value 1) if this limit would be exceeded.

The environment variable `PPCOMPILER` may be used to select between the Poly/ML or SML/NJ compiler if ProofPower has been installed for both compilers. If it is set, the value of this variable must be either "`POLYML`" or "`SMLNJ`".

See Also `hol`, `zed`, `pp`.

SML

|**pp** $-d$ *database*[#*theoryname*] [$-i$ *files*] [$-f$ *files* [$-n$|$-s$] [$-v$]] [$--$ *ml_flags*]

Description pp runs ProofPower on the indicated database. If no `-d database` is provided to pp, the function fails. For portability, the database name should be given without any architecture- or compiler-specific prefixes or suffixes. Any such prefixes or suffixes will be added automatically by pp, If the resulting file name is an absolute path name (i.e., starts with a '/' character), then that is used as the database file name. If the resulting file name is not an absolute path name, pp searches for the database file using the search path given in the environment variable `$PPDATABASEPATH`, if set. If `$PPDATABASEPATH` is not set, pp searches for the database in the current directory, then in the subdirectory `db` of the user's home directory and then in the subdirectory `db` of the ProofPower installation directory.

If specified, *theoryname* gives the name of a theory to be made the current theory at the start of the session. If *theoryname* is not specified, then current theory will be set to the theory current when the database was last saved by *save_and_quit* or, if just created, to the cache theory for the database. The files identified by any `[-i files]` options are then executed in turn. *files* is a comma-separated list of files.

If `-f files` is provided, then the files specified in the list *files* are loaded in batch mode. Once loading is complete the database is saved and the batch session is terminated. The saving of the database can be suppress by providing the `-n` flag. The default action if any of the files fails to load is for the session to terminate at that point and the database is not saved. By providing the `-s` flag, the user can indicate to the system to save the database in batch mode upon failure. The `-n` and `-s` flags are mutually exclusive. If they are both provided, a warning message is issued and the `-s` flag is ignored.

By default, the production of subgoal package output in a batch load is as determined by the value of the flag *subgoal_package_quiet* stored in the database. If the `-v` flag is specified to pp, the subgoal package output is produced whereas if the `-q` flag is specified, it is suppressed.

If `-f files` is not provided, then the system then issues a prompt for user input.

Flags which appear after `--` are passed directly onto the Standard ML system for processing. This mechanism can be used to tailor the heap size under SML/NJ: e.g., `pp -d hol -- -h 32000`. The environment variable `PPCOMPILER` may be used to select between the Poly/ML or SML/NJ compiler if ProofPower has been installed for both compilers. If it is set, the value of this variable must be either "`POLYML`" or "`SMLNJ`".

The environment variable `PPLINELENGTH`, if set, determines the initial value of the string control *line_length*. This gives the line length used by various listing facilities, e.g., *print_theory* and *output_theory*. In interactive use, the `xpp` interface will set `PPLINELENGTH` automatically if it has not been set explicitly by the user.

Errors pp prints a message and exits (with status 1) if the database cannot be accessed or if the theory name specified as part of the `-d` argument does not exist in the database.

See Also `pp_make_database`, `pp_list`, `pp_read`, `hol`, `zed`

SML
xpp [*Standard X Toolkit options*] [*xpp options*]

Description The program xpp provides a convenient way to prepare, check and execute ProofPower scripts under the X Windows System. xpp combines a general purpose text editor with a command interface for operating the ProofPower specification and proof facilities. Consult the xpp help menu or the xpp *User Guide* for information on how to use it.

'Standard X toolkit options' refers to common options which are automatically supported by most X Windows applications. An example is the option '-display', which may be used to specify the X server on which you wish xpp output to be displayed.

The xpp option -f file may be used to specify a file to be loaded into the editor when xpp starts. If you omit this option, xpp will start off editing an empty file.

If you specify the xpp option -d database, xpp will run an interactive command session working on the specified ProofPower database. If you omit this option, xpp will just run as an editor.

The command line options mentioned above are the most common ones. The program has a number of other options you may wish to use. Consult the xpp *User Guide* for further details.

See Also *USR031: ProofPower - Xpp User Guide*

SML
zed_list [−*c*] [−*d database*[#*theory*]] [−*i scripts*] [−*v*] *theory ...*
zed_list [−*d database*[#*theory*]] [−*i scripts*] [−*v*]
zed_list [−*c*] [−*d database*] [−*i scripts*] [−*v*] −*a*

Description zed_list is used to obtain selected information from a ProofPower-Z database. It functions in the same manner as hol_list except that it uses defaults appropriate to the ProofPower-Z, and a Z theory lister.

In the first form of use, where a list of one or more theory names is specified, zed_list uses ProofPower-Z to generate on its standard output listings (in the Z language using the function *z_output_theory*) of the indicated theories, in a form suitable for processing by doctex. Any cache theory (i.e. the theory name is in the list returned by *get_cache_theories*) will be printed with most of the theory detail elided, unless the -c option is given.

In the second form, with no list of theory names, zed_list lists the names of all the theories whose language is Z in the database one per line on its standard output channel, in a sorted order.

The third form, with -a, is like the first but causes all of the theories in the database whose language is "Z" to be listed in a sorted order.

In any of the three forms the program will start a session as if by command zed with the supplied -d and -i arguments (if any), and it is in this environment that the theory listing is done. The output of this startup will be suppressed, including any indication of failure to load the initialisation scripts.

Each theory is, if possible, made current, or at least in scope, when it is listed.

In any form -v indicates the log of the preprocessing should also be output.

Errors zed_list prints a message and exits (with the value 1) if the database or any of the theories does not exist. The log of the failure is sent to the standard output, the message to the error output.

See Also pp_list, hol_list, zed, pp_make_database

SML

conv_ascii [−*r*] [−*K*] [−*k keyword_file_name*] <*filename*> ...
conv_extended [−*r*] [−*K*] [−*k keyword_file_name*] <*filename*> ...

Description *conv_ascii* converts ProofPower documents using the extended character set into ASCII keyword format. *conv_extended* performs the opposite conversion.

The *filename* arguments may be just the base-name, perhaps with a directory name prefix, or may include the `.doc` suffix. By default, the result of the conversion is checked by converting in the opposite direction and comparing with the input. If the check is successful, the `.doc` file is then replaced by the result of the conversion. If the conversion appears to be unsuccessful the output of the conversion is placed in a file with suffix `.asc` or `.ext` in the current directory, and the `.doc` file is left unchanged.

If −*r* is specified no check is made and the output of the conversion is placed in a file with suffix `.asc` or `.ext`.

Note that the check will always fail on a file containing a mixture of extended characters and ASCII keywords. Use −*r* and then, if all is well, overwrite the `.doc` file with the `.asc` or `.ext` file using *mv(1)* or *cp(1)* to convert a such file into a homogeneous one.

The check will also fail if the file is already in the desired format, in which case there is no need to run the conversion program.

The −*K* and −*k* options indicate the keyword files to be used as for *doctex* and *docsml* (and are only needed if fonts other than those supplied with ProofPower are being used.)

See Also *docpr*

SML

docdvi [−*v*] [−*f view_file_name*] [−*K*] [−*k keyword_file_name*]
[−*e edit_file_name*] [−*p TeX_program_name*] [− $\mathcal{N}$] <*filename*> ...

Description Shell script that combines the actions of `doctex`, `bibtex` (which is part of the basic TeX distribution) and `texdvi` with the intention of fully processing a simple document from its `.doc` form to a printable `.dvi` file.

The option −$\mathcal{N}$ controls how many times LaTeX should be invoked, the default is three (i.e., '−*3*'), the values of $\mathcal{N}$ may be in the range one to four inclusive. The other options are as for `doctex` and `texdvi`.

LaTeX and `bibtex` are run so that if they detect errors and prompt for input they will read an end of file and thus stop immediately.

In some cases an extra run of LaTeX may be required. In these cases LaTeX will output the message: '*LaTeX Warning: Label(s) may have changed. Rerun to get cross-references right.*'

See Also *doctex*, *texdvi*

SML

docpr [−*n*] [−*p*] [−*s*] [−*v*] [−*w width*] <*filename*> ...

Description Shell script that prints out files that may contain extended characters in a verbatim-like manner. Lines may be numbered in the output by using the −*n* option. Lines are folded at at 80 characters wide, or at the width given by the −*w width* option. The output may be viewed on screen with the −*s* option, the default is to print the output. By default all intermediate files are deleted, with the −*p* option the `.dvi` file will be preserved. With the −*v* option details of the files processed are listed on the standard output.

See Also *doctex*, *texdvi*

SML

doctex $[-v]$ $[-f$ *view_file_name*$]$ $[-e$ *edit_script*$]$
$[-K]$ $[-k$ *keyword_file_name*$]$ <*filename*> ...
docsml $[-v]$ $[-f$ *view_file_name*$]$ $[-K]$ $[-k$ *keyword_file_name*$]$ <*filename*> ...

Description Shell scripts that sieve each of their *filename* arguments to produce various output files. These arguments may be given just as the base-name, perhaps with a directory name prefix, or may include the `.doc` suffix. When the $-v$ option is set details of the files read and written are shown on the standard output. The default steering files are named `sieveview` and `sievekeyword` and looked for first in the current directory, second on the callers execution path (from the UNIX environment variable `$PATH`). The default viewfile may be changed with the $-f$ option. The default keyword file may be suppressed with the $-K$ option. Additional keyword files may be given with the $-k$ option which may be used several times. The $-e$ option identifies the name of a script of `ex` commands which are used to edit the `.tex` file.

The output file from `doctex` has suffix `.tex` and is intended for processing with `texdvi`. The output file from `docsml` has suffix `.sml` and is typically processed by loading it into a ProofPower database.

See Also *texdvi*, *docdvi*

SML

texdvi $[-v]$ $[-b]$ $[-p$ *TeX_program_name*$]$ <*filename*> ...

Description Shell script that runs LaTeX on each of the *filename* arguments to produce the corresponding `.dvi` file. These arguments may be just the base-name, perhaps with a directory name prefix, or may include the `.tex` suffix. When the $-v$ option is set details of the `.tex` and`.dvi` files read and written are shown on the standard output. To support indexing this script ensures that a `.sid` file exists before LaTeX is called; when LaTeX completes any `.idx` file is sorted to create a`.sid` file ready for the next time `texdvi` is used. When initially producing a `.dvi` file `texdvi` will need to be run up to four times so that the derived information such as tables of contents and inter-page references stabilise.

The LaTeX program is `latex` by default but a different program may be specified with the $-p$ option.

If the `-b` option is specified, `bibtex` is run after running `latex`.

See Also *texdvi*, *docdvi*

PROGRAMMING UTILITIES

2.1 Error Management

SML
signature **BasicError** = *sig*

Description This is the signature of the structure *BasicError*.

SML
exception **Fail** *of MESSAGE*
exception **Error** *of MESSAGE*

Description These exception are raised to report error conditions. *Fail* is for errors which may be trapped (so that the associated message is suppressed). *Error* is intended to ensure that the message will be reported and, by convention, should not be trapped.

Uses Obscure debugging situations.

SML
type **MESSAGE**

Description This type is used to pass error and other messages around in the system.

Uses Obscure debugging situations.

SML
val **area_of** : *exn* −> *string*

Description This returns the name of the function which raised an exception (provided the exception was raised with *fail* following the usual conventions). If the exception was not the one raised by *fail* then it is raised again.

Uses For use when coding new facilities to add to the system.

SML
val **divert** : *exn* −> *string* −> *string* −> *int* −> (*unit* −> *string*) *list* −> ′*a*
val **list_divert** : *exn* −> *string* −> ((*string* ∗ *int* ∗ ((*unit* −> *string*) *list*)) *list*) −> ′*a*
val **elaborate** : *exn* −> *int* −> *string* −> *int* −> (*unit* −> *string*) *list* −> ′*a*

Description These functions support a style of error handling in which, if an error is reported during evaluation of an expression, the source of the error may be checked and the error report modified if needed to give a more meaningful report to the user. Sources of errors are identified by the string passed as the first argument to the function *fail* which is used to flag trappable errors. By convention, this string gives the name of the top level function which has raised the error.

In the call *divert X from new new_msg inserters*, *X* is the exception which has been raised and *from* identifies a possible source for an error report. *inserters* is a list of functions to be used to generate insertions for the error message (as with *fail* q.v.). If an error has been reported by *from*, the call will have the same effect as if *fail new new_msg inserters* had been called.

list_divert X new triples handles the more general case in which errors from several sources are expected. *X* and *new* are as for *divert*. *triples* gives a list of triples giving possible sources of error and the corresponding new messages and insertion functions.

elaborate is similar to *divert* but makes it possible to expand on the information provided by the function that has raised the exception. In the call *elaborate X old_msg new new_msg inserters*, *old_msg* identifies an error message text. If *X* results from a call of *fail* (or equivalent) with that error message text, the effect is as if *fail new new_msg* (*inserters*′@*inserters*) had been called, with *inserters*′ the list of string-valued functions associated with *X*.

Uses For use when coding new facilities to add to the system.

SML

```
val fail : string -> int -> (unit -> string) list -> 'a
val error : string -> int -> (unit -> string) list -> 'a
```

Description These functions report a message of the corresponding class with text determined by an integer parameter and a list of string valued functions. The string parameter is intended to give the name of the top level function which has invoked the error message.

The error messages are stored in a database maintained by *new_error_message* and the integer parameter gives the key for the desired entry in the database. The list of string-valued functions allow the messages to be parameterised. When the error is printed, the functions are evaluated to produce a list of strings. Substrings of the database entry of the form "?*i*" where *i* is a decimal digit are replaced by the corresponding entries in the list (with "?*0*" corresponding to the head of the list). (If there are more than ten entries in the list, entries after the tenth are evaluated but the result of the evaluation is ignored).

fail is for unrecoverable errors which may, however, be trapped. It causes exception *Fail* to be raised.

error is for unrecoverable errors which must be reported to the user. It causes exception *Error* to be raised. As for *set_flag* etc.

Uses For use when coding new facilities to add to the system.

SML

```
val get_error_message : int -> (string list) -> string
```

Description This function returns the entry in the error message database associated with the given integer key. The second parameter gives a list of strings to be inserted into the text of the message. Substrings of the message text of the form "?*i*", where *i* is a decimal digit, indicate positions where these insertions are to be made. "?*0*" identifies the string at the head of the list etc.

Errors

2002 The error number ?0 does not identify an entry in the error message database

SML

```
val get_error_messages : unit -> {id:int, text:string} list
val set_error_messages : {id:int, text:string} list -> unit
```

Description *get_error_messages* returns the contents of the error message database as a list.

set_error_messages uses *new_error_message* to add any new error messages in a list of such into the database of error messages. It will issue a message on the standard output (and change nothing) for any messages that do not match those already present.

SML
```
val get_message_text: MESSAGE -> string
```

Description This returns a printable form of an error message text. The message text is given without the header information which is inserted by *get_message*, q.v.

Uses In constructing extensions to the system.

The error message data structure includes functions passed as arguments to *fail* or *error* that are called to generate parts of the message. If any of these functions raises *Fail*, the exception is caught and the string returned is a report on the failure.

See Also *fail*, *error*, *get_message*

Errors

2004 Failure detected formatting message: ?0
*2005 * failure ?0.?1 reported **

SML
```
val get_message: MESSAGE -> string
```

Description This returns a printable form of an error message value. The message text is followed by a trailer of the form "<*#nnnnn area*>", where *#nnnnn* is the number of the message in the error database and *area* typically gives the name of the function which gave rise to the error message.

Uses In constructing extensions to the system.

See Also *get_message_text*

SML
```
val new_error_message : {id:int, text:string} -> unit
```

Description This function adds a new entry to the database of error messages. Note that substrings of the message of the form "?*i*" where *i* is a decimal digit have special significance (see *fail* for details). "??" may be used to insert a single "?" character in a message.

If the *id* and the *text* are identical to an existing entry, then *new_error_message* has no effect. If there is an existing entry with the same *id* but a different *text* then a message is reported on the standard output and the existing entry is left unchanged.

Errors

2001 The error number ?0 is already in use for a different message

Uses For use when adding facilities to the system.

SML
```
val pass_on : exn -> string -> string -> 'a
```

Description *pass_on exn from to* is similar to *reraise*, q.v., but the function name associated with the exception is only modified if it is equal to *from*, in which case it is changed to *to*.

SML
```
val pending_reset_error_messages : unit -> unit -> unit
```

Description This function is intended for use in system initialisation and shutdown. The binding *val p = pending_reset_error_messages()*, defines *p* as a function which will set the internal state of the *BasicError* module to the value it had at the time the binding for *prcs* was made. This is used to remember the set-up for error messages introduced in a child database.

SML
```
val pp'change_error_message : {id:int, text:string} -> unit
```

Description This function changes an entry in the database of error messages. If the number does not identify an existing entry a new entry is made.

Uses ICL Use only.

SML
```
val pp'error_init : unit -> unit
```

Description This function is used to initialise certain aspects of the error reporting system. It is called automatically at the start of each session. It is harmless, but unnecessary, to call it within a session.

SML
```
val reraise : exn -> string -> 'a
```

Description This re-raises an exception. If the exception is the exception *Fail* (as raised by *fail*, q.v.) then the function name associated with the exception is changed to the name given by the second argument.

Uses For use when coding new facilities to add to the system.

2.2 Data Types

SML
```
signature UtilitySharedTypes = sig
```

Description Any new types in the Utility structures mentioned in more than one signature will be declared in this signature.

SML
```
datatype 'a OPT = Nil | Value of 'a;
```

Description A type of "optional" values.

Uses A typical use for the datatype *'a OPT* is in implementing partial functions for which raising an exception is not an appropriate action for undefined cases.

See Also *force_value*, *is_Nil*

SML
```
type 'a S_DICT;
```

Description The type of simple dictionaries: (*string* ∗ *'a*) *list*.

See Also Signature *SimpleDictionary*.

2.3 Lists

SML
```
signature ListUtilities = sig
```

Description Holds a variety of utility Standard ML list functions.

SML
```
val all_different : ''a list -> bool;
```

Description *all_different* determines whether a list has any repeated entries.

See Also *all_distinct*

SML
```
val all_distinct : ('a * 'a -> bool) -> 'a list -> bool;
```

Description *all_distinct eq list* determines whether *list* has any repeated entries using *eq* to test for equality. Each member, x of the list is tested against all the subsequent members of the list, with x being the first argument to *eq*.

See Also *all_different*

SML
```
val all : 'a list -> ('a -> bool) -> bool;
```

Description *all list cond* is true iff. all elements of *list* satisfy *cond*.

SML
```
val any : 'a list -> ('a -> bool) -> bool;
```

Description *any list cond* is true iff. some element of *list* satisfies *cond*.

SML
```
val app : ('a -> unit) -> 'a list -> unit;
```

Description Apply a function to each element of a list in turn for the side-effect.

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

Description *combine* combines a pair of lists into a list of pairs. It is the left inverse of *split*.

Errors
```
1007   Cannot combine unequal length lists
```

See Also *split*, *zip*

SML
```
val contains : ''a list -> ''a -> bool;
```

Description *contains list x* searches for a member of *list* equal to x and returns true iff. it finds one.

See Also *present*, *mem*

SML
```
val cup : ''a list * ''a list -> ''a list;
```

Description An infix binary union operation for lists, with Standard ML equality test. It has the same result ordering as *union*(q.v.).

See Also *list_cup*, *union*

SML
```
val diff : ''a list * ''a list -> ''a list;
```

Description *diff* is the set difference operator for lists.

SML
```
val drop : 'a list * ('a -> bool) -> 'a list;
```

Description *list drop cond* is the list obtained by deleting all members of *list* for which the boolean function *cond* is true.

See Also *less*

SML
```
val filter : ('a -> bool) -> 'a list -> 'a list;
```

Description *filter pred list* returns a list that is *list*, except that elements of the list that don't satisfy *pred* are dropped.

Definition
```
filter pred [] = []
| filter pred (a :: x) = (
        if pred a
        then (a :: filter pred x)
        else filter pred x);
```

SML
```
val find : 'a list -> ('a -> bool) -> 'a;
```

Description *find list cond* searches for the first member of *list* satisfying *cond*, and returns such a member if there is one.

Errors
```
1004    Element cannot be found in list
```

SML
```
val flat : 'a list list -> 'a list;
```

Description *flat* takes a list of lists and returns the result of concatenating them all.

SML
```
val fold : ('a * 'b -> 'b) -> 'a list -> 'b -> 'b;
```

Description Fold a list into a single value:

Definition
```
fold f [x1, x2, ...., xk] b = f(x1, f(x2, ... f (xk, b))...)
```

See Also revfold

SML
```
val force_value : 'a OPT -> 'a;
```

Description Force an object of type *'a OPT* (q.v) into one of type *'a*:

Definition
```
force_value (Value x) = x
```

Errors
```
1001    Argument may not be Nil
```

SML

```
val from : 'a list * int -> 'a list;
```

Description *list from n* takes the trailing slice of *list*. It uses 0-based indexing. If *n* is 0 or negative then entire list is returned, and if *n* indexes past the other end of the list then the empty list is returned.

Example

```
[0,1,2,3] from 2 = [2,3]
```

See Also *to*

SML

```
val grab : ''a list * ''a -> ''a list;
```

Description *list grab what* is the list obtained by inserting *what* at the head of *list* if it is not a member of it already, in which case *list* is returned.

See Also *insert*

SML

```
val hd : 'a list -> 'a;
val tl : 'a list -> 'a list;
```

Description *hd* returns first element of a list, *tl* returns all but the first element of a list.

Definition

```
hd (a :: x) = a
tl (a :: x) = x
```

Errors

```
1002    An empty list has no head
1003    An empty list has no tail
```

SML

```
val insert : ('a * 'a -> bool) -> 'a list -> 'a -> 'a list;
```

Description *insert eq list what*is the list obtained by inserting *what* at the head of *list* if it is not a member, by equality test *eq*, of it already, in which case *list* is returned.

See Also *grab*

SML

```
val interval : int -> int -> int list;
```

Description *interval a b* is the list $[a, a+1, a+2 \ldots, b]$. This is taken to be [] if $a > b$ and to be $[a]$ if $a = b$.

SML

```
val is_Nil : 'a OPT -> bool
```

Description Is the argument equal to *Nil* (q.v).

Definition

```
is_Nil Nil = true
| is_Nil _ = false
```

SML

```
val is_nil : 'a list -> bool;
```

Description *is_nil* tests whether a list is empty([]). It can be used for lists of types which do not admit equality.

SML
```
val lassoc1 : (''a * ''a) list -> ''a -> ''a;
```

Description *lassoc1 alist arg* is *x*, where (*arg*, *x*) is the first element of *alist* with *arg* as its left item. The function is made total by taking *arg* as the result if there is no appropriate member of the list.

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N}$ = *1 . . . 5*.

SML
```
val lassoc2 : (''a * 'b) list -> (''a -> 'b) -> ''a -> 'b;
```

Description *lassoc2 alist f arg* is *x*, where (*arg*, *x*) is the first element of *alist* with *arg* as its left item. The function is made total by returning *f arg* if there is no appropriate member of the list.

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N}$ = *1 . . . 5*.

SML
```
val lassoc3 : (''a * 'b) list -> ''a -> 'b;
```

Description *lassoc3 alist arg* is *x*, where (*arg*, *x*) is the first element of *alist* with *arg* as its left item.

Errors
```
1005 No such value in association list
```

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N}$ = *1 . . . 5*.

SML
```
val lassoc4 : (''a * 'b) list -> 'b -> ''a -> 'b;
```

Description *lassoc4 alist default arg* is *x*, where (*arg*, *x*) is the first element of *alist* with *arg* as its left item. The function is made total by returning *default* if there is no appropriate member of the list.

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N}$ = *1 . . . 5*.

SML
```
val lassoc5 : (''a * 'b) list -> ''a -> 'b OPT;
```

Description *lassoc5 alist arg* is *Value x*, where (*arg*, *x*) is the first element of *alist* with *arg* as its left item. The function is made total by returning *Nil* if there is no appropriate member of the list.

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N}$ = *1 . . . 5*.

SML
```
val length : 'a list -> int;
```

Description *length* returns the length of a list. Note that the Standard ML function *size* can be used to find the length of strings.

SML
```
val less : ''a list * ''a -> ''a list;
```

Description *list less what* is the list obtained by deleting all members of *list* which are equal to *what*.

See Also *drop*

SML
```
val list_cup : ''a list list -> ''a list;
```

Description A distributed union operation for lists, with Standard ML equality test.

Definition
```
list_cup [list0, list1, ..., listn] =
        list0 cup (list1 cup ...(listn cup [])...))
```

See Also *cup*, *list_union*

SML
```
val list_overwrite : (''a * 'b) list * (''a * 'b) list -> (''a * 'b) list;
```

Description *alist list_overwrite olist* overwrites *alist* with each element of *olist*, using *overwrite*(q.v).

Definition
```
fun alist list_overwrite olist = (
        fold (fn (l1, l2) => l2 overwrite l1) olist alist
)
```

See Also *overwrite*, *list_roverwrite*.

SML
```
val list_roverwrite : ('a * ''b) list * ('a * ''b) list -> ('a * ''b) list;
```

Description *alist list_roverwrite olist* overwrites *alist* with each element of *olist*, using *roverwrite* (q.v.).

Definition
```
fun alist list_roverwrite olist = (
        fold (fn (l1, l2) => l2 roverwrite l1) olist alist
)
```

See Also *roverwrite*, *list_overwrite*.

SML
```
val list_union : ('a * 'a -> bool) -> 'a list list -> 'a list;
```

Description A distributed union operation for lists, with parameterised equality test:

Definition
```
list_union eq [list0, list1, ..., listn] =
        union eq list0 (union eq list1 (...(union eq listn [])...))
```

See Also *union*, *list_cup*.

SML
```
val mapfilter : ('a -> 'b) -> 'a list -> 'b list;
```

Description Map a function over a list. If, when evaluating

mapfilter f (*x_1* :: ... *x_k* − *1* :: *x_k* :: *x_k* + *1* :: ...)

the evaluation of *f x_k* raises a *Fail* exception, then the result will be

(*f x_1* :: ... *f x_k* − *1* :: *f x_k* + *1* :: ...)

SML
```
val mem : ''a * ''a list -> bool;
```

Description *x mem list* searches for a member of *list* equal to *x* and returns true iff. it finds one.

See Also *contains*, *present*

SML
| *val* **nth** : $int -> 'a\ list -> 'a$;

Description Return the *n*-th element of a list. The head of the list is the *0*-th element.

Errors
| *1009* *Index past ends of list*

SML
| *val* **overwrite** : $(''a * 'b)\ list * (''a * 'b) -> (''a * 'b)\ list$;

Description *alist overwrite* (a, b) gives the list in which the first pair in *alist* that has the left item *a* is replaced with the pair (a, b). If no such pair is found in *alist* then it returns the list of (a, b) appended to the tail of *alist*.

See Also *roverwrite*, *list_overwrite*

SML
| *val* **present** : $(('a * 'a) -> bool) -> 'a -> 'a\ list -> bool$;

Description *present eq x list* searches for a member, *y*, of *list* that satisfies $eq(x, y)$ and returns true iff. it finds one.

See Also *contains*, *mem*

SML
| *val* **rassoc1** : $(''a * ''a)\ list -> ''a -> ''a$;

Description *rassoc1 alist arg* is *x*, where $(x,\ arg)$ is the first element of *alist* with *arg* as its right item. The function is made total by taking *arg* as the result if there is no appropriate member of the list.

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N} =\ 1 \ldots 5$.

SML
| *val* **rassoc2** : $('a * ''b)\ list -> (''b -> 'a) -> ''b -> 'a$;

Description *rassoc2 alist f arg* is *x*, where $(x,\ arg)$ is the first element of *alist* with *arg* as its left item. The function is made total by returning *f arg* if there is no appropriate member of the list.

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N} =\ 1 \ldots 5$.

SML
| *val* **rassoc3** : $('a * ''b)\ list -> ''b -> 'a$;

Description *rassoc3 alist arg* is *x*, where $(x,\ arg)$ is the first element of *alist* with *arg* as its right item.

Errors
| *1005* *No such value in association list*

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N} =\ 1 \ldots 5$.

SML
| *val* **rassoc4** : $('a * ''b)\ list -> 'a -> ''b -> 'a$;

Description *rassoc4 alist default arg* is *x*, where $(x,\ arg)$ is the first element of *alist* with *arg* as its right item. The function is made total by returning *default* if there is no appropriate member of the list.

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N} =\ 1 \ldots 5$.

SML

val **rassoc5** : $('a * ''b)\ list\ {-}{>}\ ''b\ {-}{>}\ 'a\ OPT$;

Description *rassoc5 alist arg* is *Value x*, where $(x,\ arg)$ is the first element of *alist* with *arg* as its right item. The function is made total by returning *Nil* if there is no appropriate member of the list.

See Also lassoc$\mathcal{N}$ and rassoc$\mathcal{N}$, where $\mathcal{N} =\ 1 \ldots 5$.

SML

val **revfold** : $('a * 'b\ {-}{>}\ 'b)\ {-}{>}\ 'a\ list\ {-}{>}\ 'b\ {-}{>}\ 'b$;

Description Fold a list into a single value:

Definition

$revfold\ f\ [x1,\ x2,\ ....,\ xk]\ b = f(xk,\ ...,\ f(x2,\ f\ (x1,\ b))...)$

See Also fold

SML

val **roverwrite** : $('a * ''b)\ list * ('a * ''b)\ {-}{>}\ ('a * ''b)\ list$;

Description *alist roverwrite* (a, b) gives the list that in which the first pair in *alist* that has the right item b is replaced with the pair (a, b). If no such pair is found in *alist* then it returns the list of (a, b) appended to the end of *alist*.

See Also *overwrite*, *list_roverwrite*

SML

val **split3** : $('a * 'b * 'c)\ list\ {-}{>}\ 'a\ list * 'b\ list * 'c\ list$;

Description Split a list of triples into a triple of lists. *split3* is the analogue of *split* for lists of triples.

See Also *split*

SML

val **split** : $('a * 'b)\ list\ {-}{>}\ 'a\ list * 'b\ list$;

Description Split a list of pairs into a pair of lists.

Definition

$split\ [(x0,\ y0),\ (x1,\ y1),\ ...\ (xk,\ yk)] = [x0,\ x1,\ ...\ ,\ xk],\ \ [y0,\ y1,\ ...\ ,\ yk]$

See Also *split3*, *combine*

SML

val **subset** : $''a\ list * ''a\ list\ {-}{>}\ bool$;

Description *l1 subset l2* is true iff. all the elements of *l1* are also elements of *l2*

See Also =

SML

val **to** : $'a\ list * int\ {-}{>}\ 'a\ list$;

Description *list to n* takes the initial slice of *list*. It uses 0-based indexing. If n is 0 or negative an empty list is returned, and if n indexes past the other end of *list* then the entire list is returned.

Example

$[0,1,2,3]\ to\ 2 = [0,1,2]$

See Also *from*

SML
```
val union : ('a * 'a -> bool) -> 'a list -> 'a list -> 'a list;
```

Synopsis A prefix binary union operation for lists, with parameterised equality test.

Description *union* is essentially a binary union operation for lists. Since we need it to work on types which are not equality types, it has a parameter giving the relation to be used to determine equality of members of the lists. In some cases it may be important for the order of members of the union to be known. The rule is that *union eq list1 list2* is the list obtained by prepending those elements of *list1* not already present in *list2*, to the list *list2*. Presence for x in the list being created being that there is a member, y, of the list being created with $eq(x, y) = true$. If *list1* contains duplicates then all but the rightmost will be eliminated, but those in *list2* will not be. Note also that if one of the lists is small it is better supplied as the first list argument if efficiency is of the essence.

Definition
```
union eq (list1 @ [a]) list2 = union eq list1 (
        if present eq a list2
        then list2
        else (a :: list2)
) | union eq [] list2 = list2
```

See Also *cup*, *list_union*

SML
```
val which : (('a * 'a) -> bool) -> 'a -> 'a list -> int OPT;
```

Description *which eq x list* returns *Value* of the position of first element, y, in *list* for which *eq x y* is true. It uses 0-based indexing. If no such y is found, then it returns *Nil*.

SML
```
val zip : ('a -> 'b)list -> 'a  list -> 'b list;
```

Description Given a list of functions, and a list of arguments, of the same length, apply each function to its corresponding argument. For the cases when the list of functions induce side effects, note that the functions are applied from the head of their list to the tail, and will be applied until there are insufficient elements of either list to continue. If there lists are not of equal length then at that point a failure will be raised.

See Also *combine*

Errors
```
1008    List lengths differ
```

SML
```
val ~<= : ''a list * ''a list -> bool;
val ~= : ''a list * ''a list -> bool;
```

Description *l1* <= *l2* is true iff. every member of *l1* is also a member of *l2*. *l1* = *l2* is true iff. the set of members of *l1* is equal to the set of members of *l2*.

See Also *subset*

2.4 Functions

SML

signature **FunctionUtilities** = *sig*

Description Holds a variety of utility Standard ML functions concerned with functions.

SML

$val\ {**}\ : ('a \ {-}{>}\ 'b) * ('c \ {-}{>}\ 'd) \ {-}{>}\ 'a * 'c \ {-}{>}\ 'b * 'd;$

Description The infix operator $**$, with precedence 4 (higher than "o"), applies the first of a pair of functions to the first of a pair, and the second of the pair of functions to the second of the pair, returning the pairing of the results.

Definition

$(f\ {**}\ g)\ x = (f\ x,\ g\ x)$

SML

val **curry** : $('a * 'b \ {-}{>}\ 'c) \ {-}{>}\ 'a \ {-}{>}\ 'b \ {-}{>}\ 'c;$

Description *curry f a b* gives $f\ (a, b)$.

See Also *uncurry*

SML

val **fst** : $'a * 'b \ {-}{>}\ 'a;$

Description *fst* is the left projection function for pairs: $fst(a, b) = a$.

See Also *snd*

SML

val **fun_and** : $(('a \ {-}{>}\ bool) * ('a \ {-}{>}\ bool)) \ {-}{>}\ 'a \ {-}{>}\ bool;$
val **fun_or** : $(('a \ {-}{>}\ bool) * ('a \ {-}{>}\ bool)) \ {-}{>}\ 'a \ {-}{>}\ bool;$
val **fun_not** : $('a \ {-}{>}\ bool) \ {-}{>}\ 'a \ {-}{>}\ bool;$
val **fun_true** : $'a \ {-}{>}\ bool;$
val **fun_false** : $'a \ {-}{>}\ bool;$

Description These functions allow a style of programming that handles predicates rather than booleans.

Definition

$(f\ fun\_and\ g)\ x = f\ x\ andalso\ g\ x$
$(f\ fun\_or\ g)\ x = f\ x\ orelse\ g\ x$
$(fun\_not\ f)\ x = not(f\ x)$
$fun\_true\ x = true$
$fun\_false\ x = false$

SML

val **fun_pow** : $int \ {-}{>}\ ('a \ {-}{>}\ 'a) \ {-}{>}\ 'a \ {-}{>}\ 'a;$

Description For non-negative n, *fun_pow n f* is f^n, i.e. the function

$$\lambda x \bullet f(f(...f(fx)....))$$

where f appears n times.

Errors

1010 First argument must not be negative

SML
```
val repeat : (unit -> 'a) -> unit;
val iterate : ('a -> 'a) -> 'a -> 'a;
```

Description *repeat* applies its argument to () until it fails (with an error generated by *fail*, q.v.), whereupon it returns (). *iterate f a* applies *f* to *a*. If this causes no failure it then calls *iterate f* on the result. If it fails (with an error generated by *fail*, q.v.) it returns *a*. Failures other than those caused by *fail* are not handled.

Definition
```
fun repeat f        = (f (); repeat f) handle (Fail _) => ()
fun iterate f a     = (iterate f (f a)) handle (Fail _) => a
```

SML
```
val snd : 'a * 'b -> 'b;
```

Description *snd* is the right projection function for pairs: $snd(a, b) = b$.

See Also *fst*

SML
```
val swap : 'a * 'b -> 'b * 'a;
```

Description *swap* interchanges the elements of a pair: $swap(a, b) = (b, a)$.

SML
```
val switch : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c;
```

Description *switch f a b* gives *f b a*.

SML
```
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c;
```

Description *uncurry f (a, b)* gives *f a b*.

See Also *curry*

2.5 Combinators

SML

signature **Combinators** = *sig*

Description Holds the three combinators S, K, I.

SML

val **I** : $'a$ −> $'a$

Description The identity combinator: $I\ x\ =\ x$.

SML

val **K** : $'a$ −> $'b$ −> $'a$

Description The deletion combinator: $K\ x\ y$ is x.

SML

val **S** : $('a$ −> $'b$ −> $'c)$ −> $('a$ −> $'b)$ −> $'a$ −> $'c$

Description The duplication combinator: $S\ f\ g\ a$ is $(f\ a)(g\ a)$.

2.6 Characters

SML

signature **CharacterUtilities** = *sig*

Description Holds a variety of utility Standard ML functions concerned with character handling.

SML

val **is_all_decimal** : *string* −> *bool*;

Description *is_all_decimal* checks whether a string consists of one or more decimal digits.

SML

val **nat_of_string** : *string* −> *int*;

Description *nat_of_string* converts a string into non-negative integer (using decimal notation).

See Also *string_of_int*

Errors

1012 ?0 is not a decimal string
1013 String is empty

SML

val **string_of_int** : *int* −> *string*;

Description *string_of_int* converts an integer into a decimal string.

See Also *nat_of_string*

2.7 Simple Dictionary

SML
signature **SimpleDictionary** = *sig*

Description Holds a set of Standard ML functions concerned with a linear search dictionary.

Uses For handling small dictionaries.

See Also *EfficientDictionary*.

SML
val **initial_s_dict** : *'a S_DICT*;

Description The empty dictionary, which gives a starting point for the use of the simple dictionary functions. It does not associate a value with any name.

SML
val **s_delete** : *string -> 'a S_DICT -> 'a S_DICT*;

Description *s_delete* deletes an element of the domain of a dictionary. If the element is not in the domain it returns the dictionary unchanged. *s_delete name dict* returns a dictionary that does not associate anything with *name*, but otherwise associates as *dict*.

SML
val **s_enter** : *string -> 'a -> 'a S_DICT -> 'a S_DICT*;

Description *s_enter* implements overwriting by a singleton function. *s_enter name value dict* returns the dictionary that associates *name* with *value*, and otherwise associates as *dict*. Overwriting is done "in place", entries not previously present will be placed at the end of the dictionary viewed as a list.

SML
val **s_extend** : *string -> 'a -> 'a S_DICT -> 'a S_DICT*;

Description *s_extend* implements extension by a singleton function, that is to say it is like *s_enter*. *s_extend name value dict* returns the dictionary that associates *name* with *value*, and otherwise associates as *dict*. It fails if *name* is already in the domain of *dict*. Entries not previously present will be placed at the head of the dictionary viewed as a list.

Errors
1014 *?0 is already in dictionary*

SML
val **s_lookup** : *string -> 'a S_DICT -> 'a OPT*;

Description *s_lookup* implements application (of the dictionary viewed as a partial function). *s_lookup name dict* returns the value that *dict* associates with *name*.

SML
val **s_merge** : *'a S_DICT -> 'a S_DICT -> 'a S_DICT*;

Description *s_merge* extends one dictionary by another. The dictionary *s_merge dict1 dict2* will associate a name with the value that either *dict1* or *dict2* associates it with.

Failure Will get the *s_extend* failure message if any element is common to the domains of both dictionaries (*dict1* and *dict2*). Duplicate keys in the first list will also cause an *s_extend* error, but will be replicated in the result if found in the second list.

2.8 Efficient Dictionary

SML
```
signature EfficientDictionary = sig
```

Description This is the signature of a structure implementing dictionaries (lookup-up tables) based on hash-search techniques.

Uses For handling large dictionaries.

See Also *SimpleDictionary*.

SML
```
type 'a E_DICT;
```

Description The type of efficient dictionaries.

SML
```
type E_KEY;
val e_get_key : string -> E_KEY;
val e_key_lookup : E_KEY -> 'a E_DICT -> 'a OPT
val e_key_enter : E_KEY -> 'a -> 'a E_DICT -> 'a E_DICT;
val e_key_extend : E_KEY -> 'a -> 'a E_DICT -> 'a E_DICT;
val e_key_delete : E_KEY -> 'a E_DICT -> 'a E_DICT;
val string_of_e_key : E_KEY -> string;
```

Description The abstract data type *E_KEY* represents the hash-keys used in the internals of the efficient dictionary (*E_DICT*) access functions. *e_get_key* computes the hash-key for a given string. This may then be used as an argument to the functions *e_key_lookup*, *e_key_enter*, *e_key_extend* and *e_key_delete* which perform the same functions as the corresponding functions without "*key_*" in the name. This approach may be used if the same string is to be used to access several efficient dictionaries to avoid the computational cost of recalculating the hash-key. *string_of_key* is the left inverse of *e_get_key*.

Failure The failures are exactly as for the corresponding string access functions. In particular, the area names in error messages are, e.g., "e_lookup" rather than "e_key_lookup" etc.

SML
```
val e_delete : string -> 'a E_DICT -> 'a E_DICT;
```

Description *e_delete* deletes an element of the domain of a dictionary. If the element is not in the domain it returns the dictionary unchanged. *e_delete name dict* returns a dictionary that does not associate anything with *name*, but otherwise associates as *dict*.

SML
```
val e_enter : string -> 'a -> 'a E_DICT -> 'a E_DICT;
```

Description *e_enter* implements overwriting by a singleton function. *e_enter name value dict* returns the dictionary that associates *name* with *value*, and otherwise associates as *dict*.

SML
```
val e_extend : string -> 'a -> 'a E_DICT -> 'a E_DICT;
```

Description *e_extend* implements extension by a singleton function, that is to say it is like *e_enter*. *e_extend name value dict* returns the dictionary that associates *name* with *value*, and otherwise associates as *dict*. It fails if *name* is already in the domain of *dict*.

Errors
```
1014   ?0 is already in dictionary
```

SML

```
val e_flatten : 'a E_DICT -> 'a S_DICT;
```

Description *e_flatten* converts an efficient dictionary into a simple one. The result will contain no duplicates, but will be in no useful order.

SML

```
val e_lookup : string -> 'a E_DICT -> 'a OPT
```

Description *e_lookup* implements application (of the dictionary viewed as a partial function). *e_lookup name dict* returns the value that *dict* associates with *name*.

SML

```
val e_merge : 'a E_DICT -> 'a E_DICT -> 'a E_DICT;
```

Description *e_merge* extends one efficient dictionary by another. The dictionary *e_merge dict1 dict2* will associate a name with the value that either *dict1* or *dict2* associates it with.

Failure Will get the *e_extend* failure message if an element is common to the domains of both dictionaries.

SML

```
val e_stats : 'a E_DICT -> {height : int, nentries : int, nnodes : int, sumweights : int};
```

Description *e_stats dict* returns statistics about the internals of the efficient dictionary *dict*. Efficient dictionaries are currently represented as binary trees whose non-leaf nodes each carry a simple dictionary of entries (in case of collisiion of hash values). The statistics currently returned are the height of the tree, the number of entries, the number of nodes and the sum over all entries of the depth of the entries (i.e, the sum of the number of entries per node weighted by node-depth).

SML

```
val initial_e_dict : 'a E_DICT;
```

Description The empty dictionary, which gives a starting point for the use of the efficient dictionary functions. It does not associate a value with any name.

SML

```
val list_e_enter : 'a E_DICT -> 'a S_DICT -> 'a E_DICT;
```

Description *list_e_merge* extends an efficient dictionary by overwriting with entries from a simple one. That is, for each association in the simple dictionary an *e_enter* is executed on the efficient dictionary.

SML

```
val list_e_merge : 'a E_DICT -> 'a S_DICT -> 'a E_DICT;
```

Description *list_e_merge* extends an efficient dictionary by merging with entries from a simple one. That is, for each association in the simple dictionary an *e_extend* is executed on the efficient dictionary.

Failure Will get the *e_extend* failure message if an element is common to the domains of both dictionaries.

2.9 Sorting

SML

signature **Sort** = *sig*

include Order;

Description This provides an efficient sort utility package. For historical reasons it includes the structure *Order*.

SML

val **sort** : *'a ORDER -> 'a list -> 'a list*
val **merge** : *'a ORDER -> 'a list -> 'a list -> 'a list*

Description *sort* sorts a list and *merge* merges two lists assumed already to be sorted. Both functions are parametrised by an ordering function of type *'a ORDER*, i.e., $'a- >' a- > int$. The integer, say n returned by an application of this function, say $f\ a\_1\ a\_2$, is interpreted as follows:

$n < 0$ *a_2* is to come after *a_1* (i.e. the arguments are in order).

$n > 0$ *a_2* is to come before *a_1* (i.e. the arguments are out of order).

$n = 0$ *a_2* is to be taken as equal to *a_1*

Sorting eliminates duplicate elements in the sense of the equality test given by the ordering. Merging includes just one copy of an element that occurs once in each of its arguments in the result. The result of merging unsorted lists is unspecified; in particular, the result is unspecified if there is duplication within one of the lists.

Example

To sort a list of integers, ilist in ascending order:

```
sort (curry (op -)) ilist
```

or

```
sort int_order ilist
```

See Also For convenient ways of constructing orderings, see, e.g. *string_order* and *list_order*.

2.10 Sparse Arrays

SML
```
signature SparseArray = sig
```

Description This is the signature of a structure implementing sparse arrays (i.e. imperative data structures representing finite partial functions on the integers). The sparse arrays also give an efficient means for handling dense (i.e. contiguous) arrays whose size varies. To facilitate their use for such dynamically sized arrays, the sparse arrays have lower and upper bound attributes which gives the smallest and largest indices into the array which identify an occupied cell.

The design of the structure is an adaptation of the library structure *Array* implementing fixed length arrays.

SML
```
type '_a SPARSE_ARRAY;
```

Description This is the type of a sparse array with entries of type *'_a*.

SML
```
val array : int -> '_a SPARSE_ARRAY;
```

Description This function creates an empty sparse array. The parameter indicates the length of an internal data structure used to represent the array. For a contiguous array or for a sparsely filled array with a random distribution of occupied cells, the average access time for an element will be proportional to n/l where n is the number of occupied cells and l is this length.

Errors
```
1102 The length parameter must be positive
```

SML
```
val lindex : '_a SPARSE_ARRAY -> int
val uindex : '_a SPARSE_ARRAY -> int
```

Description *lbound*(*array*) (resp. *ubound*(*array*)) returns the smallest (resp. largest) index of an occupied cell in the sparse array *array*. An exception is raised if the array is empty.

Errors
```
1103 the array is empty
```

SML
```
val scratch : '_a SPARSE_ARRAY -> unit;
```

Description *scratch array* empties all cells in the sparse array *array*.

SML
```
val sub_opt : ('_a SPARSE_ARRAY * int) -> '_a OPT
```

Description *sub*(*array*, *i*) returns *Value a*, where *a* is the occupant of the *i*-th cell of the sparse array *array*. If the cell is unoccupied it returns *Nil*.

SML
```
val sub : ('_a SPARSE_ARRAY * int) -> '_a
```

Description *sub*(*array*, *i*) returns the occupant of the *i*-th cell of the sparse array *array*. An exception is raised if the cell is not occupied.

Errors
```
1101 Cell with index ?0 is empty
```

SML

```
val update : ('_a SPARSE_ARRAY * int * '_a) -> unit;
```

Description *update*(*array*, *i*, *a*) makes *a* the occupant of the *i*-th cell of the sparse array *array*. The cell need not previously have been occupied (indeed, *update* is the only means by which cells become occupied).

2.11 Dynamic Arrays

SML
```
signature DynamicArray = sig
```

Description This is the signature of a structure implementing dynamic arrays with 0-based indexing (i.e. imperative data structures representing finite partial functions on the integers, whose range is an interval *1 ... n*). The implementation gives constant access time. The design of the structure is an adaptation of the library structure *Array* implementing fixed length arrays.

SML
```
type '_a DYNAMIC_ARRAY;
```

Description This is the type of a dynamic array with entries of type *'_a*.

SML
```
val array : int -> '_a DYNAMIC_ARRAY;
```

Description This function creates an empty dynamic array. The parameter indicates the length of an internal data structure used to represent the initial size of the array. The average access time for an element will be constant — the underlying array structure is grown as necessary.

Errors

1301 The initial size parameter must be positive

SML
```
val scratch : '_a DYNAMIC_ARRAY -> unit;
```

Description *scratch array* empties all cells in the sparse array *array* and reduces the underlying data structure to the initial length specified when the array was first created.

SML
```
val sub_opt : ('_a DYNAMIC_ARRAY * int) -> '_a OPT
```

Description *sub(array, i)* returns *Value a*, where *a* is the occupant of the *i*-th cell of the dynamic array *array*. If the cell is unoccupied it returns *Nil*.

SML
```
val sub : ('_a DYNAMIC_ARRAY * int) -> '_a
```

Description *sub(array, i)* returns the occupant of the *i*-th cell of the dynamic array *array*. An exception is raised if the cell is not occupied.

Errors

1101 Cell with index ?0 is empty
1303 Index ?0 is out of range

SML
```
val uindex : '_a DYNAMIC_ARRAY -> int
```

Description *lbound(array)* the largest index of an occupied cell in the dynamic array *array* or *~1* if no cells in the array are occupied.

SML
```
val update : ('_a DYNAMIC_ARRAY * int * '_a) -> unit;
```

Description *update(array, i, a)* makes *a* the occupant of the *i*-th cell of the sparse array *array*. The cell need not previously have been occupied (indeed, *update* is the only means by which cells become occupied).

2.12 Arbitrary Magnitude Integer Arithmetic

SML

```
signature Integer = sig
    eqtype INTEGER;
    val idiv : INTEGER * INTEGER -> INTEGER;
    val imod : INTEGER * INTEGER -> INTEGER;
    val @* : INTEGER * INTEGER -> INTEGER;
    val @+ : INTEGER * INTEGER -> INTEGER;
    val @- : INTEGER * INTEGER -> INTEGER;
    val @~ : INTEGER -> INTEGER;
    val iabs : INTEGER -> INTEGER;
    val @< : INTEGER * INTEGER -> bool;
    val @> : INTEGER * INTEGER -> bool;
    val @<= : INTEGER * INTEGER -> bool;
    val @>= : INTEGER * INTEGER -> bool;
    val integer_of_string : string -> INTEGER;
    val @@ : string -> INTEGER;
    val string_of_integer : INTEGER -> string;
    val int_of_integer : INTEGER -> int;
    val integer_of_int : int -> INTEGER;
    val natural_of_string : string -> INTEGER;
    val zero : INTEGER;
    val one : INTEGER;
    val string_of_float : INTEGER * INTEGER * INTEGER -> string;
    val integer_order : INTEGER -> INTEGER -> int;
```

Description This is the signature of an open structure providing arithmetic on integers of arbitrary magnitude. It is used to support HOL natural numbers and other object language numeric types. The names of the usual arithmetic operators are decorated with an initial *i* or @ as appropriate. The string conversions work with signed decimal string representations. Either '−' or '∼' may be used for unary negation and a leading '+ ' is also allowed. @@ is an abbreviation for *integer_of_string*. *natural_of_string* is a converter for non-negative numbers (it has the same error cases as *nat_of_string*).

string_of_float interprets a triple (x, p, e) as a floating point number with value $x \times 10^{e-p}$ and converts the triple into its string representation.

integer_order implements the ordering of the integers in the form used by *sort*, q.v.

Errors

1201 *the divisor is zero*
1202 *an empty string is not a valid decimal number*
1203 *the string '?0' is not a valid decimal number*
1204 *the conversion would overflow*

2.13 Order-preserving Efficient Dictionary

SML
```
type 'a OE_DICT;
val initial_oe_dict : 'a OE_DICT;
val oe_lookup : string -> 'a OE_DICT -> 'a OPT
val oe_enter : string -> 'a -> 'a OE_DICT -> 'a OE_DICT;
val oe_extend : string -> 'a -> 'a OE_DICT -> 'a OE_DICT;
val oe_delete : string -> 'a OE_DICT -> 'a OE_DICT;
val oe_key_lookup : E_KEY -> 'a OE_DICT -> 'a OPT
val oe_key_enter : E_KEY -> 'a -> 'a OE_DICT -> 'a OE_DICT;
val oe_key_extend : E_KEY -> 'a -> 'a OE_DICT -> 'a OE_DICT;
val oe_key_delete : E_KEY -> 'a OE_DICT -> 'a OE_DICT;
val oe_flatten : 'a OE_DICT -> 'a S_DICT;
val oe_key_flatten : 'a OE_DICT -> ('a * E_KEY) S_DICT;
val e_dict_of_oe_dict : 'a OE_DICT -> 'a E_DICT;
val list_oe_merge : 'a OE_DICT -> 'a S_DICT -> 'a OE_DICT;
val oe_merge : 'a OE_DICT -> 'a OE_DICT -> 'a OE_DICT;
```

Description This type and associated access functions implement order-preserving efficient dictionaries. The functions have exactly the same effect as the corresponding functions *e_lookup*, *e_enter* etc., qv., for the type *E_DICT* except that *se_flatten* returns a list which preserves the order in which entries were made (last-in, first-out). If an entry is updated (rather than added) by *oe_key_enter* or *oe_enter*, the updated entry appears in its original position.

list_oe_merge enters the list of items in the second argument into the dictionary given as its first argument in tail-first (right-to-left) order.

Failure The failures are exactly as for the corresponding *E_DICT* functions. In particular, the area names in error messages are, e.g., "e_lookup" rather than "oe_lookup" etc.

2.14 Compatibility with SML'90

```
SML
signature BasicIO = sig
        type instream;
        type outstream;
        exception Io of {cause:exn, function:string, name:string}
        val close_in : instream -> unit
        val close_out : outstream -> unit
        val end_of_stream : instream -> bool
        val input : instream * int -> string
        val lookahead : instream -> string
        val open_in : string -> instream
        val open_out : string -> outstream
        val output : outstream * string -> unit
        val std_in : instream
        val std_out : outstream
end;
signature ExtendedIO = sig
        type instream;
        type outstream;
        val can_input : instream * int -> bool
        val flush_out : outstream -> unit
        val open_append : string -> outstream
        val is_term_in : instream -> bool
        val input_line : instream -> string
        val is_term_out : outstream -> bool
        val system : string -> bool
        val get_env : string -> string
        val std_err : outstream
end;
```

Description These are the signatures of the structures that implement SML'90-style I/O. *BasicIO* is open. *ExtendedIO* is not.

ExtendedIO differs from the the original SML'90 in several respects:

- It provides *system* instead of *execute* (which cannot be implemented cleanly on UNIX implementation sof the SML'97 standard basis library, since the SML'90 signature does not give an interface for the caller to reap the executed process).
- It provides *std_err*, which was not in the SML'90 library at all (and is the same as *TextIO.stdErr* in the SML'97 standard basis library).
- It provides *get_env* which is the UNIX *get_env* with non-existent environnent variables returning an empty string.

SML
```
signature PPArray = sig
        exception Subscript
        type 'a array
        val arrayoflist: '_a list -> '_a array
        val array : int * '_a -> '_a array
        val length  : 'a array -> int
        val sub : 'a array * int -> 'a
        val tabulate : int * (int -> '_a) -> '_a array
        val update : 'a array * int * 'a -> unit
end;
```

Description This is the signature of a structure that provides an array datatype compatible with the ProofPower code (independent of the underlying compiler).

SML
```
signature PPString = sig
        val implode : string list -> string;
        val explode : string -> string list;
        exception Ord;
        val ord : string -> int;
        val chr : int -> string;
        val string_of_exn : exn -> string;
end;
```

Description This is the signature of an open structure that provides string functions compatible with the ProofPower code (independent of the underlying compiler).

SML
```
signature PPVector = sig
        exception Subscript
        exception Size
        type '_a vector
        val vector: '_a list -> '_a vector
        val length  : 'a vector -> int
        val sub : 'a vector * int -> 'a
        val tabulate : int * (int -> '_a) -> '_a vector
end;
```

Description This is the signature of a structure that provides a vector (read-only array) datatype compatible with the ProofPower code (independent of the underlying compiler).

SML

```
(*
structure SML97BasisLibrary = struct
        val explode : string -> char list; ...
        structure Array = Array; ...
end;
*)
```

Description This is a structure containing the required structures of the Standard ML '97 Basis Library together with some functions from the '97 standard for the language that are redefined by ProofPower.

It is provided so that these structures can still be accessed when ProofPower defines a structure of the same name as a basis library structure (e.g., "*Char*").

The structure *SML97BasicLibrary.Prelude* contains the functions from the '97 standard for the language that are redefined by ProofPower. If you open this structure and later wish to revert to the ProofPower versions of *explode*, *hd*, etc., open the structures *PPString* and *ListUtilities*.

SYSTEM FACILITIES

3.1 System Control

SML
```
signature SystemControl = sig
```

Description This is the signature of the structure *SystemControl*.

SML
```
val get_flags : unit -> (string * bool) list
val get_int_controls : unit -> (string * int) list
val get_string_controls : unit -> (string * string) list
val get_controls : unit ->
        ((string * bool) list * (string * int) list * (string * string) list)
```

Description These functions return the names and current values of the system flags or controls.

SML
```
val get_flag : string -> bool
val get_int_control : string -> int
val get_string_control : string -> string
```

Description These functions are used to get the values of named control variables of the corresponding types. The parameter gives the name of the control variable.

Errors
```
2011    The name ?0 is not in use as a control variable name
```

Uses This function is for use when adding new facilities to the HOL system which require global control variables.

SML
```
val new_flag :
        {name:string, control:bool ref, default:unit->bool, check:bool -> bool} -> unit
val new_int_control :
        {name:string, control:int ref, default:unit->int, check:int -> bool} -> unit
val new_string_control :
        {name:string, control:string ref, default:unit->string, check:string -> bool} -> unit
```

Description These functions are used to introduce new named control variables of the corresponding types. The *name* parameter gives the name of the new control variable. The *control* component of the parameter gives the variable itself. The *default* component of the parameter is a function which is used by *reset_flag*, *reset_int_control* or *reset_string_control* to reset the value.

After the introduction, users may update the control using one of *set_flag*, *set_int_control* or *set_string_control*.

The *check* component of the parameter is a function to check the validity of the control values, and, if desired, to notify other code of the change in the value. When one of the control setting functions, is called, an error is reported if the *check* function for the control returns *false* when applied to the new value supplied by the caller.

The following message is raised as a warning if the control variable name is already in use. If the user elects to continue, the old control variable is renamed (by decorating it with one or more prime characters) and a new control variable is introduced with the specified name.

Errors
```
2010    The name ?0 is already in use as a control variable name
```

Uses This function is for use when adding new facilities to the HOL system which require global control variables.

SML
```
val pending_reset_control_state : unit -> unit -> unit
```

Description This function is intended for use in system initialisation and shutdown. The binding *val prcs = pending_reset_control_state()*, defines *prcs* as a function which will set the internal state of the *SystemControl* module to the value it had at the time the binding for *prcs* was made. This is used to remember the set-up for controls introduced in a child database. Note that, to avoid problems with stateful user-defined check functions, this function does not attempt to set the values of the controls. The values are, after all, not part of the *SystemControl* module's internal state.

SML
```
val reset_flags : unit -> unit
val reset_int_controls : unit -> unit
val reset_string_controls : unit -> unit
val reset_controls : unit -> unit
```

Description These functions reset the current values of all the system flags or controls in the system, as by *reset_flag*, etc.

SML
```
val reset_flag : string -> bool
val reset_int_control : string -> int
val reset_string_control : string -> string
```

Description These functions are used to reset the values of named control variables of the corresponding types. The parameter gives the name of the control variable. They return the previous value of the control variable.

Errors
```
2011 The name ?0 is not in use as a control variable name
```

Uses This function is for use when adding new facilities to the HOL system which require global control variables.

SML
```
val set_flags : (string * bool) list -> unit
val set_int_controls : (string * int) list -> unit
val set_string_controls : (string * string) list -> unit
val set_controls : ((string * bool) list * (string * int) list * (string * string) list)
        -> unit
```

Description These functions set the current values of the system flags or controls named in the lists. Items that are not mentioned keep their previous values.

SML
```
val set_flag : (string * bool) -> bool
val set_int_control : (string * int) -> int
val set_string_control : (string * string) -> string
```

Description These functions are used to change the values of named control variables of the corresponding types. The first parameter gives the name of the control variable. The second parameter gives the desired new value. They return the previous value of the control variable.

Errors
```
2011 The name ?0 is not in use as a control variable name
2012 Value out of range for control variable ?0
```

Uses This function is the standard means of changing global control variables.

3.2 System Initialisation

SML
```
signature HOLSystem = sig
```

Description This is the signature of the structure *HOLSystem* which contains functions used to end a HOL session and to save the results of a HOL session, as well as two access routes to the UNIX environment to the Standard ML session.

SML
```
signature Initialisation = sig
```

Description This is the signature of the structure *HOLInitialisation* which contains functions which may be used to add and test new start of session functions. These functions are for use by those extending the system.

SML
```
        (* flag: gc_messages; default false *)
```

Description The flag *gc_messages* can be used to turn the Standard ML compiler garbage collector messages on and off (*true* meaning on) providing that facility is supported by the compiler being used. By default, garbage collection messages are turned off.

SML
```
type ICL'DATABASE_INFO_TYPE;
val pp'database_info : ICL'DATABASE_INFO_TYPE;
```

Description Private ProofPower database information, that neither contains information useful to the user, nor should be overwritten by the user. Note that it is not an assignable variable. It is set by *pp'set_database_info*.

SML
```
val get_init_funs : unit -> (unit -> unit) list;
val get_save_funs : unit -> (unit -> unit) list;
```

Description These functions returns the list of functions that have been registered with *new_init_fun* and *new_save_fun*. They are made visible because they are needed to save the state in a child database.

SML
```
val     get_shell_var : string -> string;
```

Description *get_shell_var shvar* will extract the value (as a string), if any, bound to shell environment variable *shvar*. If the variable is not set the empty string will be returned.

SML
```
val init : unit -> unit;
```

Description *init* causes the initialisation functions in the table maintained by *new_init_fun* to be executed, as they would be at the start of a session. The failure of any individual initialisation function will not affect the attempted execution of the others.

Uses Mainly for use in testing extensions to the system.

See Also *new_init_fun*.

Errors
```
36014  Exception caught by init: ?0 (?1)
```

SML
```
val load_files : string list -> bool
```

Description *load_files* takes a list of files and compiles each file (using *use_file*). A message indicating the success or failure is output as each file is processed and a summary is output when all files have been processed. If all the files loaded without any error, *load_files* returns *true* else it returns *false*.

SML
```
val new_init_fun : (unit -> unit) -> unit;
```

Description *new_init_fun* adds a new entry to a table of functions which are invoked at the start of each session. At the beginning of each session, these functions are executed in turn, with the function stored by the most recent use of *new_init_fun* executed last.

SML
```
val new_save_fun : (unit -> unit) -> unit;
```

Description *new_save_fun* adds a new entry to a table of functions which are invoked when the state of a session is saved with *save*, *save_and_quit* or *save_and_exit*. The functions are executed in turn, with the function stored by the most recent use of *new_save_fun* executed last.

SML
```
val pp'reset_database_info: bool -> ICL'DATABASE_INFO_TYPE -> unit;
```

Description This function resets the current system state to a given stored value (which will generally be given by the variable *pp'database_info*), optionally setting the current theory. It is not intended to be called other than in the system start-up code.

SML
```
val pp'set_banner : string OPT -> string;
(* string control: system_banner; default - see below *)
(* string control: user_banner; default - "" *)
```

Description *pp'set_banner* (*Value banner*) will change the core part of the system banner to *banner*, returning the old value. *pp'set_banner Nil* just returns the current value. (The value is held in the string control *system_banner* and can also be changed using *set_string_control* or read *get_string_control*).

The messages below gives the banner, which has elements which may be changed by setting the string controls *system_banner* and *user_banner*. Message 36050 is printed first with *system_banner* as the insertion (?0) followed by message 36051 with insertions giving the latest copyright year (?0) and the *user_banner* (?1). If it is not empty, *user_banner* should begin with a newline character.

Message 36000 gives the value for *system_banner* set in the HOL database, the insertion being the version string taken from the variable *system_version* defined by the make file.

Errors
```
36000 ProofPower ?0 [HOL Database]
36050 === ?0
36051 === Copyright (C) Lemma 1 Ltd. 2000-?0?1
```

SML
```
val pp'set_database_info: unit -> unit;
```

Description This function sets the value of *pp'database_info* so that it describes the current system state. The function is used by *save_and_quit*, and elsewhere, but should not be directly invoked by the user.

SML
```
val pp′theory_hierarchy : pp′Kernel.pp′HIERARCHY_OPT;
```

Description Private ProofPower database information, that neither contains information useful to the user, nor should be overwritten by the user. Note that it is not an assignable variable.

SML
```
val print_banner : unit -> unit;
```

Description Output the system startup banner.

SML
```
val print_status : unit -> unit;
```

Description This command will list:

1. Current theory name;
2. Current proof context name(s);
3. Number of distinct goals to be achieved;
4. Current subgoal label;

SML
```
val    quit : unit -> unit
val    exit : int -> unit
```

Description *quit*() is used to end a session with the HOL system. In interactive use, the user is warned that the database will not be saved, and asked whether they still wish to quit. The session will be quit if the response is "y", and otherwise the user is returned to the HOL session. If it is used non-interactively, or *use_terminal* (q.v.) is not active, then the session will end without the database being saved.

exit ends the current session of the HOL system with an exit status that is the argument to *exit*. The exit status is available to the calling environment (e.g., as documented in the UNIX manual page for sh(1)). This facility enables the user to flag errors to the outside environment from within ProofPower.

See Also *save_and_quit*, *save_and_exit* to save the database.

SML

```
val save : unit -> unit;
val save_as : string -> unit;
val save_and_quit : unit -> unit;
val save_and_exit : int -> unit;
```

Description *save*() saves the user's current work to disk using the current database name (which is initially derived from the name supplied on the command line when *ProofPower* is invoked using the supplied shell scripts). *save_as name* saves the user's current work to disk under a new name (which becomes the current name used in subsequent calls of *save*()).

save_and_quit() saves the user's current work to disk and then ends the current ProofPower session.

save_and_exit saves the user's current work and then ends the current ProofPower session with an exit status that is the argument to *save_and_exit*. The exit status is available to the calling environment (e.g., as documented in the UNIX manual page for sh(1)). This facility enables the user to flag errors to the outside environment from within ProofPower.

If these function are called from another function rather than at the top-level then the function should be the last side-effecting function call before returning to the top-level, otherwise the behaviour when a new session is started on the saved state will be compiler-dependent.

The state of subsystems such as the subgoal package is preserved between sessions by system-dependent means. The compactification cache is cleared at the end of each session in order to reduce the size of the saved database.

See Also *quit*, *exit*, *clear_compactification_cache*

Errors

```
36010 The database name has not been set
36017 STATE WAS FOUND TO BE INCONSISTENT: state should not be saved
```

Errors If the database cannot be saved then depending on the Standard ML compiler, the function may exit anyway, with a compiler-specific raised error message. The only warning of this is that the start of session text informs the user of the database is read-only at that point in time. This does not happen with Standard ML of New Jersey, which reports the error and then continues the session.

3.3 Warnings

SML

```
signature Warning = sig
```

Description This is the signature of the structure containing the function *warn* which is used to report recoverable error conditions. It also contains the function *comment* which is used to pass comments from the system to the user.

SML

```
val comment : string -> int -> (unit -> string) list -> unit
```

Description *comment* is used to report messages to the user. The parameters are exactly as for *fail* and *error* (q.v.).

Errors

```
10010 *** COMMENT ?0 raised by ?1:
```

SML

```
val warn : string -> int -> (unit -> string) list -> unit
```

Description *warn* is used to report on recoverable error conditions. The parameters are exactly as for *fail* and *error* (q.v.). The behaviour of *warn* depends on the system control flag *ignore_warnings* and on whether or not the system is running interactively, as shown in the following table:

interactive	*ignore_warnings*	Effect
yes	false	the message is reported; the system asks the user whether to continue; if the answer is 'yes' then control returns to the caller of *warn* otherwise an exception is raised.
yes	true	the message is reported and control returns to the caller of *warn*
no	false	the message is reported and an exception is raised
no	true	the message is reported and control returns to the caller of *warn*

Errors

```
10001 *** WARNING ?0 raised by ?1:
10002 Do you wish to continue (y/n)?
10003 execution of ?0 abandoned
```

3.4 Profiling

SML
```
signature Profiling = sig
```

Description The signature contains definitions that may be used to record statistics, e.g., on the number of times certain functions have been called.

SML
```
(* profiling - boolean flag declared by new_flag *)
```

Description Turns profiling on (if true) or off (if false). Default is false, but flag is true during build of ProofPower-HOL. This should be maintained via the functions of structure *SystemControl*.

SML
```
val prof : string -> unit;
val counts : string -> int OPT;
val get_stats : unit -> int S_DICT;
val set_stats : int S_DICT -> unit;
val print_stats : int S_DICT -> unit;
val init_stats : unit -> unit;
```

Description These five functions provide a simple database facility, associating each name with a count. A call to *prof name* increments, if the flag "profiling" is true, the count for *name*. A call to *counts name* returns the value of the current count for *name*. A call to *get_stats* provides the counting database as an integer dictionary, in order of first name entry into database being first in the dictionary viewed as a list. The function *print_stats* will provide a one line - one entry display of an integer dictionary, in particular the kind of dictionary provided by *get_stats*. A call to *init_stats* initialises all the counts to 0 (which is also the state in which the database starts). A call to *set_stats* will restore a statistics database to a given set of values (such as those given by *get_stats*). The input list must contain no duplicated names.

It is likely that the output of *get_stats* would be best sorted before being printed by *print_stats*.

Uses The intended use of this database is to profile function calls, with the implementer making one call to *prof* per profiled function.

Errors
```
1020 Input list is ill-formed
```

3.5 Timing

SML
```
signature Timing = sig
```

Description The signature contains definitions that can be used to measure execution time of ML code.

SML
```
datatype TIMER_UNITS = Microseconds | Milliseconds | Seconds;
type 'b TIMED = {result : 'b, time : int, units : TIMER_UNITS};
val time_app : TIMER_UNITS -> ('a -> 'b) -> 'a -> 'b TIMED;

val reset_stopwatch : unit -> unit ;
val read_stopwatch : TIMER_UNITS -> int;
```

Description The function *time_app* and the associated data types *TIMER_UNITS* and *'a TIMED* may be used to measure the execution time of a function.

In the call *time_app u f x*, *u* specifies the units in which the timing is to be measured, *f* is the function to be timed and *x* gives the argument to be passed to *f*. The return value gives the result of the application *f x* together with the time taken measured in the specified units and a reminder of what the units were.

reset_stopwatch_time and *read_stopwatch_time* give a way of timing sequences of ML commands. *read_stopwatch_time u* returns the elapsed time measured in the units specified by *u* since the last call of *reset_stopwatch_time*. *read_stopwatch_time* will either return a meaningless value or result in arithmetic overflow if *reset_stopwatch_time* has not been called in the current session.

The following points should be born in mind when using these functions:

- The times are "wall-clock" times. Garbage-collection and other overheads will be included.
- Depending on the underlying Standard ML compiler, arithmetic overflow may occur if the units are chosen inappropriately for the time period being measured.
- The functions will themselves introduce a time overhead, which may vary depending on system load and other system-dependent factors.

Errors

1021 Arithmetic overflow in time conversion

INPUT AND OUTPUT

4.1 The Reader/Writer

SML

signature **HOLReaderWriter** = *sig*

Description This structure holds the HOL specific reader writer code. All the errors that the basic reader writer may raise may also be raised by the HOL reader writer.

SML

signature **ReaderWriter** = *sig*

Description File and terminal reading and writing functions.

Errors

5001 End of file found in comment
5002 End of file found in string
5003 Unknown keyword '?0' after '?1'
5004 Unknown keyword '?0'
5005 Unknown extended character '?0' (decimal ?1) after '?2'
5006 Unknown extended character '?0' (decimal ?1)
5007 Unexpected symbol '?0' (a symbol of type $Invalid$ has been read)
5008 Bracket mismatch, '?0' found after an opening '?1'
5010 Unknown language requested by symbol '?0' with language name '?1'
5011 Unknown language requested
5014 Newline found in string after '?0'
5030 End of file in quotation
5032 End of file found in Standard ML quotation
5036 Unknown language '?0' requested

Several error messages are provided to report faults in the user's textual input to the ICL HOL system, they may be produced from all of the routines *use_file*, *use_string* and *use_terminal*. Some error messages might be associated with particular routines in the *ReaderWriterSupport* structure but that is incidental to most users, so they are all gathered here.

SML

signature **ReaderWriterSupport** = *sig*

Description A set of declarations that allows the addition of new embedded languages. The HOL language is an example of a language embedded into a basic system that understands Standard ML with extended characters and percent keywords.

SML

(* **prompt1** − *boolean flag declared by new_flag, default*: ":>" *)
(* **prompt2** − *boolean flag declared by new_flag, default*: ":#" *)

Description Prompt strings for *use_terminal*. String *prompt1* is used when the reader writer is expecting the first line of a top-level expression, *prompt2* is used for subsequent lines. The strings used here must comprise characters whose decimal codes are in the range 32 to 126 inclusive, but excluding the characters 'Q' (i.e., code 81) and '%' (37).

SML
```
(* RW_diagnostics – integer control declared by new_int_control, default: 0 *)
```

Description For reader writer diagnostic purposes.

SML
```
(* use_extended_chars – boolean flag declared by new_flag, default: true *)
```

Description Controls how the writer changes the text output from the PolyML compiler. When *true* extended characters are written, when *false* the corresponding keywords are written.

SML
```
(* use_file_non_stop_mode – boolean flag declared by new_flag, default: false *)
```

Description Makes *use_file* continue reading text (if the flag is *true*) or stop reading (if *false*) from the file after an error is reported by PolyML, this includes both syntax and execution errors. Default is to stop reading.

SML
```
datatype NAME_CLASS
	=	Simple
	|	Starting	of (READER_ENV -> (string * bool)
					-> string -> bool -> string list
					-> string list) * string
	|	Middle	of string
	|	Ending	of string
	|	Ignore
	|	Invalid;
```

Description These detail the characteristics of a symbol. *Simple* is used for symbols that may be part of identifiers. *Starting*, *Middle* and *Ending* relate to the symbols position when embedding text of other languages. The function with *Starting* is the reader routine for the particular embedded language. Details of how this function should be written (and of it arguments) are given in the implementation document corresponding to this design. *Ignore* is used for characters that are completely ignored in the input, the extended characters for indexing come in this category. *Invalid* will cause an error message.

See Also Error 5007

SML

```
datatype SYMBOL
      =     SymKnown              of string * bool
                                  * PrettyNames.PRETTY_NAME
      |     SymUnknownChar        of string
      |     SymUnknownKw          of string * bool
      |     SymDoublePercent
      |     SymWhite              of string list
      |     SymCharacter of string
      |     SymEndOfInput
      ;
```

Description *SymKnown* indicates a symbol declared via *add_new_symbols*, if a keyword was read the string hold the characters without the enclosing percents and the boolean is *true*. Otherwise, when an extended character is read the string holds the character and the boolean is *false*.

SymUnknownChar indicates an extended character not declared via *add_new_symbols*.

SymUnknownKw indicates a keyword not declared via *add_new_symbols* or a badly formed keyword with no closing percent sign. The boolean is *true* for a well-formed keyword.

SymDoublePercent indicates an empty keyword, i.e., two adjacent percent signs.

SymWhite indicates a non-empty sequence of formatting characters (space, tab, newline, and formfeed) which are passed as individual characters in reverse order in the string list.

SymEndOfInput indicates an empty string was seen.

All other cases are passed back as a single character in *SymCharacter*.

SML

```
exception TooManyReadEmpties;
```

Description Associated with the reader functions is the exception *TooManyReadEmpties* which is raised when the parser has read the end of the file and has passed the end of file character at least 100 times to the compiler. Raising this exception signifies something has gone wrong in a reader.

SML

```
structure PrettyNames : sig
```

Description A structure within *ReaderWriterSupport* that gathers all the information relating the extended characters and percent keywords understood by the system, together with the interfaces for interrogating and extending the information.

SML

```
type PRETTY_NAME (* = ( string list * string OPT * NAME_CLASS ) *) ;
```

Description Each symbol is defined in a three-element tuple of this type. Elements of the tuple are as follows. First, a non-empty list of the keywords that may be used for this symbol. These keywords exclude the enclosing percent signs. Second, an optional character for the symbol. Third, a value of datatype *NAME_CLASS* indicating the characteristics of the symbol.

The extended character field, when used, contains a single character. It may be the letter "Q" or any character with decimal code greater than 127.

See Also Function *add_new_symbols*, for details of the validation of values of this type.

Example

```
([   "fn",
     "lambda"],    Value "λ",          Simple),
```

SML

```
type READER_ENV (* = {
    advance          : unit -> string,
    look_at_next     : unit -> string,
    push_back        : string -> unit
} *) ;
val skip_and_look_at_next : READER_ENV -> unit -> string;
```

Description All of the parsing functions in the reader writer support use the functions provided in this record type to read characters from the current input stream. Attempting to read characters by any other method will have unpredictable results. The utility function *skip_and_look_at_next* combines *advance* and *look_at_next* discarding the result of *advance*. Some applications will want to use instances of this data type to count line numbers, so pushing back newlines that have not been read is not advisable.

SML

```
type READER_ENV;
type READER_FUNCTION;
```

Description These types are used for reader functions for embedded languages, they are identical to the types of the same name in signature *ReaderWriterSupport*.

See Also Signature *ReaderWriterSupport*.

SML
```
type READER_FUNCTION (*
        = READER_ENV
        -> (string * bool)      (* Starting symbol *)
        -> string               (* Language name *)
        -> string               (* Opening text *)
        -> string list          (* Left hand context *)
        -> string list *);
```

Description The type of the reader functions for embedded languages. The first string argument gives the symbol that started the quotation. For a keyword enclosing percent signs are omitted and the boolean is true. For an extended character the boolean is false. The second string holds the language name without the leading "%dntext%" or trailing "%cantext%", the default language and type are expanded to give their full names, namely "HOL" or "HOL:" for the colon form. The third string is text to be included at the start of the quoted text, in the case of a HOL quotation it is the first characters that are to be read by the HOL recogniser. The string list is the left hand context of the call and must be returned with the text of the quotation added to its head.

SML
```
val abandon_reader_writer : unit -> unit;
```

Description Only meaningfully used after *use_terminal* has been called, when it forces an exit from that routine.

SML
```
val add_error_code : int * string list -> string list;
val add_error_codes : int list * string list -> string list;
```

Description For each error number "*nn*" given as the first argument an entry of the form "␣ERROR__*nn*␣" is added to the head of the second argument. (Note that "␣" denotes aspace character.)

SML
```
val add_general_reader : string * string * string * READER_FUNCTION -> unit;
val add_specific_reader : string * string * READER_FUNCTION -> unit;
val add_named_reader : string * string * string * READER_FUNCTION -> unit;
```

Description Adds reader functions to the database of known readers. The first strings give the language name, the last string holds the name of a Standard ML constructor which is to be written before the quotation when it occurs in within languages other than Standard ML. Typical values of the last string are "Lex.Term" and "Lex.Type".

Errors

5033	*Reader already present for language '?0'*
5034	*Improper reader name '?0'*
5035	*Improper reader name '?0' and '?1'*

SML

val **add_new_symbols** : *PRETTY_NAME list −> unit*;

Description Adds details of new symbols to the data structures characterising all known symbols. There is some validation of the symbols added, the list of names should not be empty, the individual names should not contain two adjacent "Q"s and the character field should have a single character which is either a "Q" or has decimal code greater than 127.

Errors

5100 Keyword '?0' has adjacent 'Q's
5101 Empty keyword list
5102 Invalid extended character '?0' with keyword '?1'
5103 Keyword '?0' duplicated
5104 Character '?0' duplicated

Errors 5100, 5101 and 5102 are issued as warnings against particular parts of the argument value, they do not prevent the other parts from being added to the data structures.

SML

val **ask_at_terminal** : *string −> string*;

Description Asks a question at the terminal by writing out the given string then reading a single line of text which is returned. Characters are read until a newline or end of file is reached, in the first case the the returned string will end with a newline.

Any characters in the type ahead buffer of the terminal input stream before *ask_at_terminal* is called are read and saved (for later analysis by the normal reading functions) before the prompt is output and the response is read.

Errors

5012 Function 'use_terminal' is not active
5013 Input stream is not a terminal, nothing read

SML

val **diag_line** : *string −> unit*;

Description *diag_line* outputs a string to the standard output stream followed by a new line, after translating it with *translate_for_output*(q.v.). It is intended for use in printing formatted terms, theorems and the like (for which the pretty printer will have included new lines within the string if necessary).

See Also *diag_string*, *raw_diag_line*.

SML

val **diag_string** : *string −> unit*;

Description *diag_string* outputs a string on the standard output stream, after translating it with *translate_for_output*(q.v.). If the string exceeds the value of *get_line_length* it attempts to split the string into tokens, to fit within the line length. A token is taken to be an initial string of spaces, followed by exclusively non-space characters.

See Also *list_diag_string*, *diag_line*, *raw_diag_string*.

SML

val **expand_symbol** : *SYMBOL −> string*;

Description A value of type *SYMBOL* is expanded into the corresponding character string.

SML
```
        val find_name : string -> PRETTY_NAME OPT
        val find_char : string -> PRETTY_NAME OPT
```

Description Finds the characteristics of a symbol based on its keyword or character. Both functions return *Nil* if the symbol is not known. They return the tuple given to *add_new_symbols* for known symbols.

SML
```
val general_quotation : READER_ENV
        -> (string * bool)     (* Start of quotation symbol *)
        -> string              (* Opening characters *)
        -> bool                (* Context, true => in Standard ML *)
        -> string list         (* Left hand context *)
        -> string list;
val specific_quotation : READER_ENV
        -> (string * bool)     (* Start of quotation symbol *)
        -> string              (* Opening characters *)
        -> bool                (* Context, true => in Standard ML *)
        -> string list         (* Left hand context *)
        -> string list;
val named_quotation : READER_ENV
        -> (string * bool)     (* Start of quotation symbol *)
        -> string              (* Opening characters *)
        -> bool                (* Context, true => in Standard ML *)
        -> string list         (* Left hand context *)
        -> string list;
```

Description Process the text of a quotation and add it to the left hand context given. The opening quotation symbol has been read and is passed as the first string argument, a keyword is passed without its enclosing percent signs and the boolean is true, for an extended character the boolean is false. For general and named quotations the next characters to be read denote the language of the quotation. The boolean argument indicates whether the left hand context is in Standard ML text or in a quotation of another language.

Errors
```
5004    Unknown keyword ‘?0‘
5006    Unknown extended character ‘?0‘ (decimal ?1)
5010    Unknown language requested by symbol ‘?0‘ with language name ‘?1‘
5011    Unknown language requested
5030    End of file in quotation
5031    End of file in language name of general quotation
```

SML

val **get_box_braces** : (*READER_ENV −> string list −> string list*)
    −> READER_ENV −> string list −> string list;
val **get_curly_braces** : (*READER_ENV −> string list −> string list*)
    −> READER_ENV −> string list −> string list;
val **get_round_braces** : (*READER_ENV −> string list −> string list*)
    −> READER_ENV −> string list −> string list;

Description These functions assemble a section of bracketed text. The opening bracket has been read, the first unread character is the first character within the brackets. Each routine reads text upto and including the matching closing bracket. The first argument is the parsing routine for reading items of text within the brackets. The third argument is the left hand context, which is returned with the bracketed text read by these functions, and the enclosing braces. The three pairs of brackets: "[ ]", "{ }" and "( )" are handled by functions *get_box_braces*, *get_curly_braces* and *get_round_braces* respectively.

Errors

5008 Bracket mismatch, '?0' found after an opening '?1'

SML

val **get_HOL_any** : *READER_ENV −> string list −> string list*

Description Assemble a section of HOL text starting with the first unread character. Text is read up to and including the first unmatched symbol of value *Ending_*. The second argument gives the left hand context, the new text read is added to that context and returned. All the errors that the basic reader writer may raise may also be raised by the HOL reader writer.

See Also Type *READER_ENV*.

SML

val **get_ML_any** : *READER_ENV −> string list −> string list*

Description Assemble a section of Standard ML text starting with the first unread character. Text is read up to the first semi colon ';', unmatched closing bracket or ending keyword. A semi colon will be read and added to the returned text, a closing bracket or ending keyword is left unread for the calling routine. The syntax error where too many closing bracket are presented must be resolved by the outermost routine that calls function. The second argument gives the left hand context, the new text read is added to that context and returned.

Errors

5003 Unknown keyword '?0' after '?1'
5005 Unknown extended character '?0' (decimal ?1) after '?2'
5007 Unexpected symbol '?0' (a symbol of type $Invalid$ has been read)

See Also Type *READER_ENV*.

SML
```
val get_ML_string : READER_ENV -> string list -> string list * int list;
val get_primed_string : READER_ENV -> string list -> string list * int list;
```

Description Assemble a string literal and add it to the left hand context given in the second argument. On entry the opening string quote has been read, exit when the closing string quote has been read. The goal of this routine is to form an equivalent string that can be read by a Standard ML compiler, and to defer as much validation of the string as possible to that compiler. Minimal validation is performed on escape sequences. Well-formed layout sequences (i.e., the sequence "\f..f\") are removed, characters not recognised as formatting ones are retained and wrapped between "\ " and " \" for later checking by the Standard ML compiler. Extended characters are translated to their three digit decimal form.

Function *get_ML_string* reads a Standard ML string.

Function *get_primed_string* reads a string enclosed with single left-hand primes (‘). These are similar to Standard ML strings but with the meanings of the single (‘) and double (") prime characters interchanged.

An end of file found in the string indicates that there is no more input available, and so an immediate failure (error 5002) is raised. Error code 5014 is included to aid in understanding where errors occur, this error is not actually generated until the first non white-space character after the newline is processed. All other errors detected in strings are reported when found, additionally their numbers passed back in the result.

Errors

5002 *End of file found in string*
5014 *Newline found in string after ‘?0‘*

See Also Type *READER_ENV*.

SML
```
val get_percent_name : READER_ENV
        -> string * PrettyNames.PRETTY_NAME OPT * bool;
```

Description Assemble a percent keyword and look it up in the list of known keywords. On entry the opening percent (%) is the first unread character.

The tuple returned contains: (1) the keyword read, but without the percent characters; (2) the symbols entry as given to *add_new_symbols* or *Nil* for an unknown keyword; (3) a flag set true if the keyword had a closing percent character, false otherwise, error reporting is left to the calling functions. Non-alphanumeric keywords may contain the characters "! & $ # + - / : < = > ? @ \ ~ ’ ^ | *"

See Also Type *PRETTY_NAME*. Type *READER_ENV*. Function *is_special_char*.

SML
```
val get_use_extended_chars_flag : unit -> bool;
```

Description This function gives the value of the flag *use_extended_chars*.

SML
```
val HOL_lab_prod_reader : READER_FUNCTION;
```

Description This is the reader function for HOL labeled products. It is provided to allow specialised versions of the HOL language to be read, it is not intended to be called directly called by any user code. All the errors that the basic reader writer may raise may also be raised by the HOL reader writer.

SML
```
val HOL_reader : string -> bool -> READER_FUNCTION;
```

Description This is the HOL reader function, its first argument is the name of the recogniser for the particular aspect of HOL to be recognised. Its second argument indicates whether this reader is considered to be used only at outermost (i.e., Standard ML's top-level): *true* is used for outermost usage, *false* for HOL text that may be used within other expresions. This function is provided to allow specialised versions of the HOL language to be read, it is not intended to be called directly called by any user code. All the errors that the basic reader writer may raise may also be raised by the HOL reader writer.

SML
```
        val is_same_symbol : (string * string) -> bool
```

Description Compare two symbols return true if they are identical, i.e., the same string. Otherwise, look up both with *find_char* and *find_name* then if they are the same symbol return true, if either is not a known symbol or they are not the same symbol return false.

SML
```
val is_special_char : string -> bool;
```

Description Checks whether the string contains a single non-alphanumeric character that is allowed in a keyword. Returns *true* if the argument contains exactly one of the characters listed in the description of function *get_percent_name*, otherwise *false* is returned.

See Also Function *get_percent_name*.

SML
```
val is_white : string -> bool
```

Description Returns *true* if the string is a single white-space character, *false* otherwise.

SML
```
val list_diag_string : string list -> unit;
```

Description *list_diag_string* outputs a list of strings onto the standard output stream, after translating them with *translate_for_output*(q.v.). The strings in the list are concatenated (with spaces to separate them) and then output with *diag_string* (q.v).

See Also *diag_string*, *diag_line*, *list_raw_diag_string*.

SML
```
val local_error : string -> int -> (unit -> string) list -> unit;
val local_warn : string -> int -> (unit -> string) list -> unit;
```

Description An error or warning message is written to the standard output, then the function returns. The arguments are identical in form to functions *error* and *fail* of `DS/FMU/IED/DTD002`.

See Also Functions *error* and *fail*.

SML
```
val look_up_general_reader : string * string -> (READER_FUNCTION * string) OPT;
val look_up_specific_reader : string -> (READER_FUNCTION * string) OPT;
val look_up_named_reader : string * string -> (READER_FUNCTION * string) OPT;
```

Description Looks up readers in the database of known readers. The argument strings are matched against the first string given in the call of the *add_...._reader*, if the reader is known then the corresponding constructor string and reader function are returned. The value *Nil* is returned for an unknown reader.

SML
|*val* **read_symbol** : *READER_ENV −> SYMBOL*;

Description Reads one or more characters and returns a value of type *SYMBOL*. No errors are reported by this routine. The routine reads as many characters as necessary to form a symbol. End of file is returned as a *SymEndOfInput*.

SML
|*val* **reset_use_terminal** : *unit −> unit*;

Description Restores the state that controls *use_terminal* to its default values. N.b., this bypasses the check that *use_terminal* makes on recursive calls (and so could cause a small memory leak if not used with care).

SML
|*val* **skip_comment** : *READER_ENV −> unit*;

Description Skip over a comment which comprises a sequence of characters within which the comment braces '`(*`' and '`*)`' are properly balanced. This routine is entered when the opening round bracket of the comment has been read, the opening asterisk is the first unread character. Note that Standard ML comments separate lexical items thus the calling routine should not simply discard the comment, it might replace the comment with a space character to ensure the lexical items remain separated.

Errors
|*5001 End of file found in comment*

See Also Type *READER_ENV*.

SML
|*val* **SML_recogniser** : *string * string * 'a * string −> 'a*;

Description This routine is not intended to be directly called by any user code, it is provided to allow the quotation of Standard ML text. The context of use of this routine is that the "macro processing" of the Standard ML quotation "`%<%%dntext%SML%cantext% 42 %>%`" yields the text "`(ReaderWriterSupport.SML_recogniser ("%<%", "SML", 42 , "%>%"))`" which is read by the Standard ML compiler.

Errors
|*5032 End of file found in Standard ML quotation*
|*5050 Incorrect symbols starting or ending of Standard ML quotation*: '*?0*', '*?1*', '*?2*'

SML
|*val* **string_of_int3** : *int −> string*

Description The string representation of small positive integers is needed in various places, particularly within Standard ML strings where some characters are denoted by their decimal code in three digits, preceded by a backslash. Function *string_of_int3* gives a three character with leading zeros representation of small positive numbers. In general the routine *PolyML.makestring* cannot be used, if the value last passed to *PolyML.print_depth* is zero then *PolyML.makestring* converts numbers into three dots. The intended use of this function is in building reader writer extensions for other languages. In such places it is intended that the caller only supply suitable arguments, getting this wrong indicates something wrong in the design of the caller. The text of the message anticipates this usage.

Errors
|*5040 DESIGN ERROR*:*Number ?0 is too big or is negative*

SML
```
val to_ML_string : string -> string
```

Description Converts characters which are to form part of a string literal into another string which may be read by a Standard ML compiler and which has the same meaning. This is intended to form the string representation of extended characters for passing them through to a Standard ML compiler. Characters other than space, tab and newline which are outside the range 32 to 126 (decimal) inclusive are converted to their four character equivalent of a backslash followed by a three digit decimal number with leading zeroes.

SML
```
val translate_for_output : string -> string;
```

Description Translates a string according to the macro processing rules used when outputting text. The output produced depends on the setting of the control flag *use_extended_chars*, when false the result will have no extended characters, the keyword forms will be used.

SML
```
val use_file : string -> unit;
val use_file1 : string -> unit;
```

Description Both of these functions compile and execute ProofPower-ML (i.e., Standard ML extended to allow mathematical symbols) from the named file. If the file does not exist then the it will read the file with the given name and suffix "`.ML`", if that file does not exist it will try the suffix "`.sml`".

use_file passes the file name string through *translate_for_output* before using it as an operating system file name which is appropriate for file names given as ProofPower-ML strings. The variant *use_file1* uses the string exactly as given.

See Also Error messages given with signature for *ReaderWriter*. Flag *use_file_non_stop_mode*.

Errors
```
5009 Cannot read file '?0' or '?0.ML' or '?0.sml'
```

SML
```
val use_string : string -> unit;
```

Description Read Standard ML with extended characters allowed, from the given string.

See Also Error messages given with signature for *ReaderWriter*.

SML
```
val use_terminal : unit -> unit;
```

Description Read Standard ML with extended characters allowed, from the terminal. This routine takes over the terminal, it handles all exceptions as the outermost level of the ML system. To return to the default PolyML terminal reader use *abandon_reader_writer*.

This routine prompts to the conventions of PolyML but uses the strings "`:> `" and "`:# `", the PolyML prompts do not have the colon. These strings are held as the string controls '*prompt1*' and '*prompt2*' and thus may be altered.

Typing two control-D characters to the terminal prompt, or reading the end-of-file, causes the function *PolyML.quit* to be called.

See Also Error messages given with signature for *ReaderWriter*. Control strings'*prompt1*' and '*prompt2*'.

4.2 Output

SML
signature **SimpleOutput** = *sig*

Description Holds a variety of utility Standard ML functions concerned with simple output. Related facilities may be found in structure ReaderWriter. Function *ask_at_terminal* (q.v) provides for prompted input of text from the terminal.

Strings containing extended characters and strings derived from HOL types and terms should be passed through the ReaderWriter function *translate_for_output* (q.v) before being output. This allows the proper output of keywords and extended characters on both graphic and simple ASCII terminals.

SML
(∗ **line_length** − *integer control declared by new_int_control* ∗)

Description An integer control dictating the output's length of line available for printing.

See Also *set_line_length*, *get_line_length*

SML
val **format_list** : (*'a −> string*) *−> 'a list −> string −> string*;

Description *format_list formatter items seperator* is used to format a list of items for printing as a string, perhaps for printing. Given *formatter*, a function to format a single item, *items*, a list of items, and *seperator*, a string to separate elements of a multi-element list, the resulting string is the contatenation of the formatted items with interposed separators. The formatted head element of the list becomes the left hand end of the result string.

Example
```
format_list string_of_term [⌜1⌝,⌜x⌝,⌜a ∧ b⌝] ", ";
  −−−>
val it = "⌜1⌝, ⌜x⌝, ⌜a ∧ b⌝" : string
```

SML
val **get_line_length** : *unit −> int*

Description Returns current output line length.

See Also *set_line_length*

SML
val **list_raw_diag_string** : *string list −> unit*;

Description *list_raw_diag_string* outputs a list of strings onto the standard output stream. The strings in the list are concatenated (with spaces to separate them) and then output with *raw_diag_string* (q.v).

See Also *raw_diag_string*, *raw_diag_line*, *list_diag_string*.

SML
val **raw_diag_line** : *string −> unit*;

Description *raw_diag_line* outputs a string to the standard output stream followed by a new line. It is intended for use in printing formatted terms, theorems and the like (for which the pretty printer will have included new lines within the string if necessary).

See Also *raw_diag_string*, *diag_line*.

SML
```
val raw_diag_string : string -> unit;
```

Description *raw_diag_string* outputs a string on the standard output stream. If the string exceeds the value of *get_line_length* it attempts to split the string into tokens, to fit within the line length. A token is taken to be an initial string of spaces, followed by exclusively non-space characters.

See Also *list_raw_diag_string*, *raw_diag_line*, *diag_string*.

SML
```
val set_line_length : int -> int
```

Description Set the output line length, returning the previous line length. Default length is 80, minimum length 20.

See Also *get_line_length*

Errors
```
1015   line length must be at least 20
```

4.3 HOL Lexical Analysis

SML

```
signature Lex = sig
```

Description This is the signature of the structure which contains the lexical analyser for ICL HOL.

Uses For use by those who wish to extend the system to handle languages other than HOL which have a similar lexical structure.

SML

```
datatype ASSOC =  LeftAssoc
               |  RightAssoc;
datatype FIXITY =  Nonfix
               |  Binder
               |  Infix     of ASSOC * int
               |  Prefix    of int
               |  Postfix   of int;
```

Description These data types are used in the symbol table and elsewhere to give the syntactic status of a name. *Nonfix* means no special status. The integer components are the precedences for infix, prefix or postfix status.

SML

```
datatype HOL_TOKEN = HTAqTm     of TERM
                   | HTAqTy     of TYPE
                   | HTName     of string
                   | HTNumLit   of string
                   | HTString   of string
                   | HTChar     of string
                   | HTBinder   of string
                   | HTInOp     of {name:string, is_type_op:bool,
                                    is_term_op:bool, prec : ASSOC * int}
                   | HTPostOp   of {name:string, prec : int}
                   | HTPreOp    of {name:string, prec : int}
                   | HTAnd
                   | HTBlob
                   | HTColon
                   | HTElse
                   | HTIf
                   | HTIn
                   | HTLbrace
                   | HTLbrack
                   | HTLet
                   | HTLsqbrack
                   | HTRbrace
                   | HTRbrack
                   | HTRsqbrack
                   | HTSemi
                   | HTThen
                   | HTVert
                   | HTEos;
```

Description This is the data type of the output from the HOL lexical analyser.

Uses For use by those who wish to extend the system to handle languages other than HOL which have a similar lexical structure.

SML

```
datatype INPUT = Text      of string
               | String    of string
               | Char      of string
               | Type      of TYPE
               | Term      of TERM
               | Separator of string
               | Error     of int;
```

Description This is the data type of the input to the HOL lexical analyser.

Uses For use by those who wish to extend the system to handle languages other than HOL which have a similar lexical structure.

SML
```
val is_alnum : string -> bool
val is_copula : string -> bool
val is_digit : string -> bool
val is_macro : string -> bool
val is_punctuation : string -> bool
val is_space : string -> bool
val is_symbolic : string -> bool
```

Description These functions classify character strings according to their first character. They all return false if the argument is an empty string. The characters for which the various functions return true are shown in the following table.

is_alnum	a letter or a number or the prime character '′'
is_copula	an underscore or the subscription, or superscription characters
is_digit	a decimal digit
is_macro	the character '%' which introduces preprocessor macros
is_punctuation	'(', ')', '{', }', '[', ']', ':', ';', ',', '|', '•' or '$'
is_space	a formatting character, i.e., space, tab, newline etc.
is_symbolic	any character which is not does not fall into any of the above classes

SML
```
val lex : (string list list) -> (string -> FIXITY) ->
                    INPUT list -> HOL_TOKEN list
```

Description This is the HOL lexical analyser.

The first parameter is the list of (exploded) strings which are to be taken as terminator symbols. Terminators are recognised by looking for the first match in the list, so that if one terminator is a leading substring of another the longer one must come first. No punctuation symbol should appear in a terminator. For HOL this parameter is always obtained by calling the symbol table function *get_terminators*, which maintains the list of terminators sorted in order of decreasing length.

The second parameter is used to classify names as binder, infix, prefix, postfix or nonfix.

The third parameter is the input to be lexically analysed.

Uses For use by those who wish to extend the system to handle languages other than HOL which have a similar lexical structure.

Errors
```
15001 antiquotation not allowed after '$'
15002 '$' not allowed at end of quotation
15003 lexical analyser or reader/writer error detected (?0)
15004 ill-formed keyword symbol
15005 ?0 is not a valid character literal (must contain exactly one character)
15006 error code ?0 reported by reader/writer
```

The last of these error messages occurs, e.g., when a keyword symbol has been entered incorrectly and is preceded by a more comprehensive error message from the reader/writer.

SML

```
val num_lit_of_string : string -> (INTEGER * (INTEGER * INTEGER) OPT) OPT;
```

Description The argument to this function should be a string representing a numeric literal (either a natural number, N, or a floating point number with optional, optionally signed, exponent part, $X.Y$ or $X.YeZ$. The result value is *Nil* if the string cannot be interpreted as a numeric literal. Otherwise, the result value is N, or $(XY, P, 0)$ or (XY, PZ), where XY stands for the natural number obtained by concatenating the digit sequences X and Y and P is the number of digits in Y.

4.4 Pretty Printing

SML
signature **PrettyPrinter** = *sig*

SML
(∗ *Flag* **pp_top_level_depth** : *integer control*, *default −1* ∗)
(∗ *Flag* **pp_format_depth** : *integer control*, *default −1* ∗)

Description These control the depth to which HOL types and terms are printed. Control *pp_top_level_depth* applies to values printed as part of Standard ML top-level expressions. Control *pp_format_depth* applies to values printed by the "*format_ ...*" routines. When these controls are negative, types and terms are fully printed, otherwise the value indicates how deeply the expression is printed where zero indicates suppressing the whole type or term. Suppressed types and terms, or parts thereof, are shown as three dots.

See Also Functions *format_term*, *format_term1*, *format_thm*, *format_thm1*, *format_type* and *format_type1*.

SML
(∗ *Flag* **pp_print_assumptions** : *boolean control*, *default true* ∗)

Description This controls whether the assumptions of values of type *THM* are printed. The default is to print assumptions. If assumptions are not printed then each is shown as three dots.

See Also Functions *format_thm* and *format_thm1*.

SML
end (∗ *of signature PrettyPrinter* ∗);

SML
val **format_term** : *bool −> TERM −> string list*;
val **format_term1** : *bool −> int −> TERM −> string list*;

Description Produce a number of lines, one string per line, containing a pretty printing of the given HOL Term. The text is suitable for directly outputting via the *diag_line* and *diag_string* routines *BasicIO.output*. If the boolean argument is set *false* then the strings produced from terms whose language is the same as that of the current theory will not include the term quotation symbols, in all other cases the term quotation symbols will be included. Line width is given by the integer in *format_term1*, or for *format_term* the current line width (as maintained by *set_line_length*, *q.v.*) is used.

See Also Pretty printer controls: *pp_add_brackets*, *pp_show_HOL_types*, *pp_types_on_binders* and *pp_let_as_lambda*.

SML
val **format_thm** : *THM −> string list*;
val **format_thm1** : *int −> THM −> string list*;

Description Produce a number of lines, one string per line, containing a pretty printing of the given HOL theorem. The text is suitable for directly outputting via the *diag_line* and *diag_string* routines The theorem is printed with a comma separated list of terms for the assumptions, a turnstile and finally the term representing the conclusion. Assumptions in the same language as the conclusion are not enclosed with the term quotation symbols. Other assumptions have term quotation symbols. Line width is given by the integer in *format_term1*, or for *format_term* the current line width (as maintained by *set_line_length*, *q.v.*) is used.

See Also Pretty printer controls: *pp_add_brackets*, *pp_show_HOL_types*, *pp_types_on_binders* and *pp_let_as_lambda*.

SML
```
val format_type : bool -> TYPE -> string list;
val format_type1 : bool -> int -> TYPE -> string list;
```

Description Produce a number of lines, one string per line, containing a pretty printing of the given HOL type. The text is suitable for directly outputting via the *diag_line* and *diag_string* routines If the boolean argument is set *true* then type quotation symbols will be included in the returned strings, when *false* they are excluded. Line width is given by the integer in *format_term1*, or for *format_term* the current line width (as maintained by *set_line_length*, *q.v.*) is used.

See Also Pretty printer control: *pp_add_brackets*.

SML
```
val pp_init : unit -> unit;
```

Description Initialise the pretty printing system so that values of types *TERM*, *TYPE* and *THM* will be prettily printed out as "top level" Standard ML values.

SML
```
val show_type : bool -> int OPT -> OppenFormatting.OPPEN_FUNS
        -> TYPE -> unit;
val show_term : bool -> int OPT -> OppenFormatting.OPPEN_FUNS
        -> TERM -> unit;
val show_thm : int OPT -> OppenFormatting.OPPEN_FUNS
        -> THM -> unit;
```

Description These functions enable programming of Oppen-style pretty-printing for data types that contain embedded types, terms and theorems.

4.5 Theory Lister

SML
```
signature Lister = sig
```

Description This is the signature of the structure *Lister* which contains functions for listing theories.

SML
```
signature ListerSupport = sig
datatype LISTER_SECTION =
        LSBanner     |     LSParents        |     LSChildren
|       LSConsts     |     LSAliases        |     LSUndeclaredAliases
|       LSTypes      |     LSTypeAbbrevs    |     LSUndeclaredTypeAbbrevs
|       LSFixity     |     LSTerminators    |     LSUndeclaredTerminators
|       LSAxioms     |     LSDefns          |     LSThms
|       LSTrailer
|       LSADString of string -> (string list * string)
|       LSADStrings of string -> (string list * string list)
|       LSADThms of string -> (string list * THM) list
|       LSADTerms of string -> (string list * TERM) list
|       LSADTypes of string -> (string list * TYPE) list
|       LSADTables of string -> (string list * string list) list
|       LSADSection of string -> string
|       LSADNestedStructure of string -> (string * LISTER_SECTION list);
```

Description *ListerSupport* is the signature of a structure containing a functions, *gen_theory_lister* and *gen_theory_lister1* for creating variant theory listers, e.g. for languages other than ProofPower-HOL. The data type *ListerSupport.LISTER_SECTION* controls what is listed. Each constructor of this type determines an element of the listing. The first block of constructors for the type *LISTER_SECTION* cause sections of the listing like those produced by the HOL theory lister to be included (except that *LSBanner* uses the first argument to *print*, *output*, or *output1* to compute the contents of the banner heading.) The second block of constructors are for creating application-defined sections of the listing and in each case the constructor takes as its operand a function which is passed the name of the theory being listed as argument. *LSADSection* produces a section header containing the result of applying the argument function to the theory name unless that result is an empty string, in which case it has no effect. The others are for printing (labelled) individual strings (*LSADString*) or columns of strings (*LSADStrings*), or (labelled) lists of theorems, terms, types or rows of strings (*LSADTables*). In each case the first component of (each element of) the result is used as a list of labels for the elements and is printed in the left margin and the second component is indented.

SML
```
(* sorted_listings          - flag - default false *)
(* listing_indent           - integer - control default 2 *)
```

Description These two system control variables influence the behaviour of the functions *list_theory* and *output_theory* which are used to generate theory listings. If *sorted_listings* is false (the default) then items are unsorted, otherwise they are sorted according to *string_order* (q.v.). *listing_indent* sets the indent level of the listings in terms of a number of tabstops, and its default is 2.

Errors
```
33052 integer control `?0` must be greater than zero
```

See Also *output_theory*

SML
```
val gen_theory_lister : LISTER_SECTION list ->
	{	print: (string -> string) -> string -> unit,
		out: (string -> string) -> {theory: string, out_file: string} -> unit,
		out1: (string -> string) -> {theory: string, out_file: string} -> unit};
val gen_theory_lister1 : LISTER_SECTION list ->
	{	print: (string -> string) -> string -> unit,
		out: (string -> string) -> {theory: string, out_file: string} -> unit,
		out1: (string -> string) -> {theory: string, out_file: string} -> unit};
end (* of structure ListerSupport *) (* of structure ListerSupport *);
```

Description The functions *ListerSupport.gen_theory_lister* and *ListerSupport.gen_theory_lister1* are used to create customised theory listers and can also be used to create formatted listings of other kinds.

They return a triple of functions each of which has as its first argument a function to compute the contents of the banner line in the listing from the name of the theory name. Given such an argument, the three components, *print*, *out*, and *out1* deliver results which behave very much like *print_theory*, *output_theory* and *output_theory1*, respectively, as regards where they send the listing and whether or not they insert LaTeX formatting controls in it, but what they put in the listing is determined by the argument to *gen_theory_lister*. This argument is a list of elements of type *LISTER_SECTION*, q.v.

The integer control *listing_indent*and the flag *sorted_listings* control the print of labelled lists of theorems, terms etc. *listing_indent* gives the number of spaces of indent from the left margin of the lists. If *sorted_listings* is true, the lists will be sorted using the concatenation of the labels as the sort key otherwise they are printed in the order supplied.

gen_theory_lister1 is just like *gen_theory_lister* except that it does not check whether the theory exists or whether it is in scope.

SML
```
val output_theory1 : {theory:string, out_file:string} -> unit
```

Description *output_theory1*{*theory* = *thy*, *out_file* = *file*} causes a listing of the theory *thy* to be output to the file *file*. The listing is in a format suited for display on the screen or for viewing with a text editor. The theory must be in scope, i.e. it must be the current theory or one of its ancestors.

See Also *output_theory print_theory*

Errors
```
33050 The theory ?0 is not in scope
33051 There is no theory called ?0
33101 i/o failure on file ?0 (?1)
33102 the theory ?0 does not exist
```

SML

```
val output_theory : {theory:string, out_file:string} -> unit
```

Description *output_theory{theory = thy, out_file = file}* causes a listing of the theory *thy* to be output to the file *file*. The listing is in a format suited for printing using the ICL HOL document preparation system. The theory must be in scope, i.e. it must be the current theory or one of its ancestors.

See Also *output_theory1 print_theory*

Errors

33050 The theory ?0 is not in scope
33051 There is no theory called ?0
33101 i/o failure on file ?0 (?1)
33102 the theory ?0 does not exist

SML

```
val print_theory : string -> unit
```

Description *print_theory thy* causes a listing of the theory *thy* to be written to the standard output. The listing is in a format suited for display on the screen. The theory must be in scope, i.e. it must be the current theory or one of its ancestors.

Errors

33050 The theory ?0 is not in scope
33051 There is no theory called ?0

See Also *output_theory output_theory1*

4.6 Z Theory Lister

SML
```
signature ZLister = sig
```

Description This is the signature of the structure *ZLister* which contains functions for listing ProofPower-Z theories.

SML
```
val z_output_theory : {theory:string, out_file:string} -> unit
```

Description *z_output_theory{theory = thy, out_file = file}* causes a listing of the theory *thy* to be output to the file *file*. The listing is in a format suited for printing using the ProofPower document preparation system. The theory must be in scope, i.e. it must be the current theory or one of its ancestors.

See Also *output_theory z_print_theory z_output_theory1*

Errors As for *output_theory*.

SML
```
val z_output_theory1 : {theory:string, out_file:string} -> unit
```

Description *z_output_theory1{theory = thy, out_file = file}* causes a listing of the theory *thy* to be output to the file *file*. The listing is in a format suited for display on the screen or for viewing with a text editor. The theory must be in scope, i.e. it must be the current theory or one of its ancestors.

See Also *output_theory1 z_print_theory z_output_theory*

Errors As for *output_theory1*.

SML
```
val z_print_fixity : string -> unit
```

Description If *id* has been defined as an infix operator, or other kind of fancy-fix symbol, *z_print_fixity id* prints out a Z fixity paragraph showing the template or templates in which *id* appears.

Errors
```
65100 there are no fixity paragraphs in scope containing ?0
```

SML
```
val z_print_theory : string -> unit
```

Description *z_print_theory thy* causes a listing of the ProofPower-Z theory *thy* to be written to the standard output. The listing is in a format suited for display on the screen. The theory must be in scope, i.e. it must be the current theory or one of its ancestors.

Errors As for *print_theory*.

See Also *print_theory z_output_theory z_output_theory1*

HOL TYPES AND TERMS

5.1 Syntactic Manipulations

It should be noted that the functions documented in this section are drawn from two signatures, *TypesAndTerms* and *icl′TypesAndTerms*.

Since the former includes the latter, an object available under the name *xxx*, say, or in full *TypesAndTerms.xxx*, is also available under the name *icl′TypesAndTerms.xxx*. It is the intention that users should not access objects by names of the form *icl′TypesAndTerms.xxx*. In practice, since the structure *TypesAndTerms* is open, the unqualified name *xxx* will do unless you have redefined the name *xxx*.

SML

signature **pp′TypesAndTerms** = *sig*

Description This provides the type of HOL types: *TYPE*, of HOL terms: *TERM*, and some functions upon them. A user should access all the elements of this signature through signature *DerivedTerms* (q.v).

SML

signature **TypesAndTerms** = *sig*

Description This provides various functions on derived *TERM*s, which are not considered necessary to create the abstract data type *THM*. It also contains, by inclusion, the types, and functions on the types *TERM* and *TYPE* from structure *pp′TypesAndTerms*(q.v.).

SML

```
datatype DEST_SIMPLE_TYPE =
        Vartype of string
|       Ctype of (string * TYPE list);
```

Description This is the type of simple destroyed types, related to the type *TYPE* by *dest_simple_type*(q.v) and *mk_simple_type*(q.v.). The value constructors correspond to type variables and compound types.

SML

```
datatype DEST_SIMPLE_TERM =
        Var of string * TYPE
|       Const of string * TYPE
|       App of TERM * TERM
|       Simpleλ of TERM * TERM;
```

Description This is the simple type of destroyed terms, related to the type *TERM* by *dest_simple_term*(q.v) and *mk_simple_term*(q.v.). The four value constructors represented destroyed variables, constants, applications and simple λ-abstractions respectively.

Uses In writing pattern-matching functions upon HOL terms.

See Also *DEST_TERM*.

SML

```
datatype DEST_TERM = DVar of string * TYPE |
        DConst of string * TYPE |
        DApp of TERM * TERM |
        Dλ of TERM * TERM |
        DEq of TERM * TERM |
        D⇒ of TERM * TERM |
        DT |
        DF |
        D¬ of TERM |
        DPair of TERM * TERM |
        D∧ of TERM * TERM |
        D∨ of TERM * TERM |
        D⇔ of TERM * TERM |
        DLet of ((TERM * TERM)list * TERM) |
        DEnumSet of TERM list |
        D∅ of TYPE |
        DSetComp of TERM * TERM |
        DList of TERM list |
        DEmptyList of TYPE |
        D∀ of TERM * TERM |
        D∃ of TERM * TERM |
        D∃1 of TERM * TERM |
        Dϵ of TERM * TERM |
        DIf of (TERM * TERM * TERM) |
        Dℕ of INTEGER |
        DFloat of INTEGER * INTEGER * INTEGER |
        DChar of string |
        DString of string;
```

Description This type is that of a term destroyed using the appropriate derived destructor functions (e.g. *dest_eq*) as well as the primitive ones. The type given to *D∅* and *DEmptyList* is the type of an element of the associated set or list. The type is related to *TERM* by *mk_term* (q.v.) and *dest_term* (q.v)

See Also *DEST_SIMPLE_TERM*

SML

```
eqtype TERM;
```

Description This is the type of well-formed HOL terms. Objects of this type are manipulated by term constructor, destructor and recogniser functions, such as *mk_app*, *dest_λ* and *is_var*.

SML

```
eqtype TYPE;
```

Description All HOL terms will be "typed", by associating them with an object of type *TYPE*. A type may either be a type variable or a compound type.

This is not an equality type (i.e. = cannot be used in tests for equality - see =: instead.).

SML

```
val =$ : (TERM * TERM) -> bool;
```

Description This is the (infix) equality test for HOL terms. It is retained for backwards compatibility — the type of HOL terms is now an equality type.

Instead of equality it is often preferable to test for α-convertibility, using ~=$

SML
```
val =: : (TYPE * TYPE) -> bool
```

Description This is the (infix) equality test for HOL types. It is retained for backwards compatibility — the type of HOL types is now an equality type.

SML
```
val bin_bool_op : string -> TYPE -> TYPE -> TERM;
```

Description Returns a constant with the given name, and type

⌜:BOOL → BOOL → BOOL⌝

The type arguments are dummies, present only to make the function have an acceptable signature for certain other functions.

SML
```
val BOOL : TYPE;
```

Description The HOL type of truth values:

Definition
```
val BOOL = ⌜:BOOL⌝;
```

See Also Theory "min".

SML
```
val CHAR : TYPE;
```

Description This is the HOL type of single characters.

Definition
```
val CHAR = ⌜:CHAR⌝;
```

See Also Theory "char".

SML
```
val dest_app: TERM -> (TERM * TERM);
```

Description Destroys a function application into the function and argument. Note that many derived term constructs, e.g. all quantifications, are also applications.

Definition
```
dest_app ⌜f t⌝ = (⌜f⌝, ⌜t⌝)
dest_app ⌜∀ x • t⌝ = (⌜$∀⌝, ⌜λ x • t⌝)
```

Errors
```
3010 ?0 is not of form: ⌜t1 t2⌝
```

SML
```
val dest_binder : string -> int -> string -> TERM -> TERM * TERM;
```

Description A generic method of implementing binder destructor functions:

Definition
```
dest_binder area msg binder_nm ⌜binder(λ varstruct• body)⌝ =
        (⌜varstruct⌝, ⌜body⌝)
```

where *binder* is a constant whose name is *binder_nm*. The *varstruct* may be any allowed variable structure.

See Also *dest_simple_binder*

Failure If the term cannot be destroyed, then the error will be from *area*, with a message indexed by *msg*.

SML
```
val dest_bin_op : string -> int -> string -> TERM -> (TERM * TERM);
```

Description *dest_bin_op area msg rator_nm term* first assumes that *term* is of the form ⌜*rator t1 t2*⌝, where *rator* is a constant with name *rator_nm*, and attempts to return the pair (*t1*, *t2*).

Example
```
dest_bin_op "dest_∧" 4032 "∧" ⌜a ∧ b⌝ = (⌜a⌝, ⌜b⌝)
```

If the function fails it will fail with message *msg*, area *area* and with the string form of *term*.

SML
```
val dest_char : TERM -> string;
```

Description Destroy a character literal.

Example
```
dest_char ⌜'a'⌝ = "a"
```

Errors
```
3024 ?0 is not a character literal
```

SML
```
val dest_const: TERM -> (string * TYPE);
```

Description This destroys a constant into its name and type.

Errors
```
3009 ?0 is not a constant
```

SML
```
val dest_ctype : TYPE -> string * TYPE list;
```

Description Extract the components of a compound type.

Definition
```
dest_ctype ⌜:(ty1,ty2,...)tc⌝ = ("tc", [⌜:ty1⌝,⌜:ty2⌝,...])
dest_ctype ⌜:ty tc⌝ = ("tc", [⌜:ty⌝])
dest_ctype ⌜:tc⌝ = ("tc", [])
```

Errors
```
3001 ?0 is not a compound type
```

SML
```
val dest_empty_list : TERM -> TYPE;
```

Description A derived term destructor function for empty lists.

Definition
```
dest_list ⌜[]:ty LIST⌝ = ⌜:ty⌝
```

Errors
```
4034 ?0 is not of form: ⌜[]⌝
```

SML
```
val dest_enum_set : TERM -> (TERM list);
```

Description A derived term destructor function for enumerated sets.

Definition
```
dest_enum_set ⌜{a; b; ...}⌝ = [⌜a⌝, ⌜b⌝, ...]
```

Errors
```
4011 ?0 is not of form: ⌜{t1, ...}⌝
```

SML

val **dest_eq** : *TERM* −> (*TERM* ∗ *TERM*);

Description A derived term destructor function for equations.

Definition

$dest\_eq \ulcorner a = b \urcorner = (\ulcorner a \urcorner, \ulcorner b \urcorner)$
$dest\_eq \ulcorner a \Leftrightarrow b \urcorner = (\ulcorner a \urcorner, \ulcorner b \urcorner)$

Errors

3014 ?0 is not of form: $\ulcorner t = u \urcorner$

SML

val **dest_float** : *TERM* −> *INTEGER* ∗ *INTEGER* ∗ *INTEGER*;

Description Destroy a floating point literal.

Definition

$dest\_float \ulcorner XXYY. \urcorner = (\ulcorner x \urcorner, \ulcorner 0 \urcorner, \ulcorner 0 \urcorner)$
$dest\_float \ulcorner XX.YYeZZ \urcorner = (\ulcorner x \urcorner, \ulcorner p \urcorner, \ulcorner 0 \urcorner)$
$dest\_float \ulcorner XX.YYeZZ \urcorner = (\ulcorner x \urcorner, \ulcorner p \urcorner, \ulcorner z \urcorner)$

where x is the natural number with decimal representation $XXYY$, p is the number of digits after the point in $XX.YY$ and z is the integer represented by ZZ (with $p = z = 0$ in the first case and $z = 0$ in the second).

Errors

4042 ?0 is not a floating point literal

SML

val **dest_f** : *TERM* −> *unit*;

Description This will return () if given the term $\ulcorner F \urcorner$, and otherwise fail.

Errors

4037 ?0 is not: $\ulcorner F \urcorner$

SML

val **dest_if** : *TERM* −> (*TERM* ∗ *TERM* ∗ *TERM*);

Description Destroy a conditional.

Definition

$dest\_if \ulcorner if\ c\ then\ y\ else\ n \urcorner = (\ulcorner c \urcorner, \ulcorner y \urcorner, \ulcorner n \urcorner)$

Errors

4006 ?0 is not of form: $\ulcorner if\ c\ then\ y\ else\ n \urcorner$

SML
```
val dest_let : TERM -> ((TERM * TERM)list * TERM);
```

Description A derived term destructor function for *let*-terms. See *mk_let* for details of format. The distinction between a local function definition, and a variable structure bound to an abstraction is lost, with both being destroyed to the second form.

Example
```
dest_let(mk_let([(⌜f x⌝, ⌜y⌝)],⌜bdy⌝)) =
        ([(⌜f⌝, ⌜λ x• y⌝)], ⌜bdy⌝)
dest_let(mk_let([(⌜f⌝, ⌜λ x• y⌝)],⌜bdy⌝)) =
        ([(⌜f⌝, ⌜λ x• y⌝)], ⌜bdy⌝)
```

Errors
```
4009  ?0 is not of form: ⌜let ... in ...⌝
```

dest_let (*mk_let*([], *term*)) will actually fail (unless *term* is already a *let*-term), as apply *mk_let* to ([], *term*)) will just return *term*.

SML
```
val dest_list : TERM -> (TERM list);
```

Description A derived term destructor function for list-terms.

Definition
```
dest_list ⌜[a; b; ...]⌝ = [⌜a⌝, ⌜b⌝, ...]
dest_list ⌜[]⌝ = []
```

Errors
```
4015  ?0 is not of form: ⌜[t1,...]⌝
```

SML
```
val dest_mon_op : string -> int -> string -> TERM -> TERM;
```

Description *dest_mon_op area msg rator_nm term* assumes that *term* is of the form ⌜*rator t*⌝, where *rator* is a constant with name *rator_nm*, and the function attempts to return *t*.

Example
```
dest_mon_op "dest_¬" 4029 "¬" ⌜¬ t⌝ = ⌜t⌝
```

Failure The failure message for failing to destroy the term will be from area *area*, and will have the text indexed by *msg*, and will have as argument the string form of *term*.

SML
```
val dest_multi_¬ : TERM -> (int * TERM);
```

Description *dest_multi_¬ t* will strip ¬ from *t*, returning the number of times, as well as the result. It will return (*0*, *t*) if *t* is either not boolean, or has no negations.

Example
```
dest_multi_¬ ⌜¬(¬ T)⌝ = (2, ⌜T⌝)
```

SML
```
val dest_pair : TERM -> (TERM * TERM);
```

Description A derived term destructor function for pairs.

Definition
```
dest_pair ⌜(t1, t2)⌝ = (⌜t1⌝, ⌜t2⌝)
```

Errors
```
4003  ?0 is not of form: ⌜(t1,t2)⌝
```

SML

|*val* **dest_set_comp** : *TERM* −> (*TERM* ∗ *TERM*);

Description A derived term destructor function for set comprehensions.

Example

|*dest_set_comp* ⌜{ *x* | *x* > *5*}⌝ = (⌜*x*⌝, ⌜*x* > *5*⌝)

Errors

|*4013* *?0 is not of form*: ⌜{*v* | *p*}⌝

SML

|*val* **dest_simple_binder** : *string* −> *int* −> *string* −> *TERM* −> *TERM* ∗ *TERM*;

Description Executing *dest_simple_binder area msg binder_nm* ⌜*binder*(λ *var* • *body*)⌝, where *binder* is a constant with the name *binder_nm*, will give (⌜*var*⌝, ⌜*body*⌝).

Example

|*dest_simple_binder* "*dest_simple_*∀" *3032* "∀" ⌜∀ *x* • *t*⌝ = (⌜*x*⌝,⌜*t*⌝)

See Also *dest_binder*

Failure If the term cannot be destroyed, then the error will be from *area*, with a message indexed by *msg*, and argument the string form of *term*.

SML

|*val* **dest_simple_term** : *TERM* −> *DEST_SIMPLE_TERM*;

Description An injective function, that destroys a term, returning its top-level structure, and the associated constituent parts.

See Also *DEST_SIMPLE_TERM*

SML

|*val* **dest_simple_type** : *TYPE* −> *DEST_SIMPLE_TYPE*;

Description This function destroys a HOL type into something of type *SIMPLE_DEST_TYPE* (q.v).

SML

|*val* **dest_simple_∀** : *TERM* −> (*TERM* ∗ *TERM*);

Description A derived term destructor function for ∀-terms. It cannot destroy paired abstraction ∀-terms, being the inverse of *mk_simple_*∀.

Definition

|*dest_simple_*∀ ⌜∀ *var* • *body*⌝ = (⌜*var*⌝, ⌜*body*⌝)

See Also *dest_*∀

Errors

|*3032* *?0 is not of form*: ⌜∀ *var* • *body*⌝

SML

|*val* **dest_simple_∃₁** : *TERM* −> (*TERM* ∗ *TERM*);

Description A derived term destructor function for simply abstracted ∃_*1*-terms. It may destroy only simple abstraction ∃_*1*-terms, being the inverse of *mk_simple_*∃_*1*.

Definition

|*dest_simple_*$\exists_1$ ⌜$\exists_1$ *var*• *body*⌝ = (⌜*var*⌝, ⌜*body*⌝)

Errors

|*4019* *?0 is not of form*: ⌜$\exists_1$ *v*• *t*⌝

See Also *dest_*∃_*1*

SML
```
val dest_simple_∃ : TERM -> (TERM * TERM);
```

Description A derived term destructor function for ∃-terms. It cannot destroy paired abstraction ∃-terms, being the inverse of *mk_simple_∃*.

Definition
```
dest_simple_∃ ⌜∃ var • body⌝ = (⌜var⌝, ⌜body⌝)
```

See Also *dest_∃*

Errors
```
3034  ?0 is not of form: ⌜∃ var • body⌝
```

SML
```
val dest_simple_λ: TERM -> (TERM * TERM);
```

Description Destroys a simple λ-abstraction. It cannot destroy paired λ-abstractions, being a inverse of *mk_simple_λ*.

Definition
```
dest_simple_λ ⌜λ v • t⌝ = (⌜v⌝, ⌜t⌝)
```

See Also *dest_λ*

Errors
```
3011  ?0 is not of form: ⌜λ var • t⌝
```

SML
```
val dest_string : TERM -> string;
```

Description Destroy a string literal.

Example
```
dest_string ⌜"abc"⌝ = "abc"
```

Errors
```
3025  ?0 is not a string literal
```

SML
```
val dest_term : TERM -> DEST_TERM
```

Description This function returns the "best" interpretation of a term in the form of an object of type *DEST_TERM*. E.g. it will return *DEq*(*1 2*) rather than *DComb*((\$ = *1*), *2*). It will also use the paired abstraction forms of functions in preference to the simple forms, e.g., it uses *dest_λ* not *dest_simple_λ*.

The function assumes that the name of a constant is sufficient to identify it without checking the type, as with, e.g., *dest_bin_op*(q.v).

See Also *mk_term*

SML
```
val dest_t : TERM -> unit;
```

Description This will return () if given the term ⌜*T*⌝, and otherwise fail.

Errors
```
4036  ?0 is not: ⌜T⌝
```

SML
```
val dest_vartype : TYPE -> string;
```

Description Extract the name of a type variable:

Definition
```
dest_vartype ⌜:'tv⌝ = "'tv"
```

Errors
```
3019   ?0 is not a type variable
3027   STRING STORE ERROR: cannot translate internal id (?0) to string
```

SML
```
val dest_var: TERM -> (string * TYPE);
```

Description This destroys a term variable into its name and type.

Errors
```
3007   ?0 is not a term variable
```

SML
```
val dest_∅ : TERM -> TYPE;
```

Description A derived term destructor function for empty enumerated sets.

Definition
```
dest_∅ ⌜∅:ty SET⌝ = ⌜:ty⌝
```

Errors
```
4035   ?0 is not of form: ⌜∅⌝
```

SML
```
val dest_⇔ : TERM -> (TERM * TERM);
```

Description A derived term destructor function for bi-implications. N.B. this may be successfully applied to boolean equalities.

Definition
```
dest_⇔ ⌜t1 ⇔ t2⌝ = (⌜t1⌝, ⌜t2⌝)
```

Errors
```
4031   ?0 is not of form: ⌜t1 ⇔ t2⌝
```

SML
```
val dest_→_type : TYPE -> (TYPE * TYPE);
```

Description Extract the two constituent types of a function type.

Definition
```
dest_→_type ⌜:ty1 → ty2⌝ = (⌜:ty1⌝, ⌜:ty2⌝)
```

Errors
```
3022   ?0 is not of form: ⌜:ty1 → ty2⌝
```

SML
```
val dest_∧ : TERM -> (TERM * TERM);
```

Description A derived term destructor function for conjunctions.

Definition
```
dest_∧ ⌜t1 ∧ t2⌝ = (⌜t1⌝, ⌜t2⌝)
```

Errors
```
4032   ?0 is not of form: ⌜t1 ∧ t2⌝
```

SML
```
val dest_∨ : TERM -> (TERM * TERM);
```

Description A derived term destructor function for disjunctions.

Definition
```
dest_∨ ⌜t1 ∨ t2⌝ = (⌜t1⌝, ⌜t2⌝)
```

Errors
```
4027   ?0 is not of form: ⌜t1 ∨ t2⌝
```

SML
```
val dest_¬ : TERM -> TERM;
```

Description A derived term destructor function for negations.

Definition
```
dest_¬ ⌜¬ t⌝ = ⌜t⌝
```

Errors
```
4029   ?0 is not of form: ⌜¬ t⌝
```

SML
```
val dest_⇒ : TERM -> (TERM * TERM);
```

Description A derived term destructor function for implications, returning the antecedent and consequent.

Definition
```
dest_⇒ ⌜a ⇒ b⌝ = (⌜a⌝,⌜b⌝)
```

Errors
```
3016   ?0 is not of form: ⌜t ⇒ u⌝
```

SML
```
val dest_∀ : TERM -> (TERM * TERM);
```

Description A derived term destructor function for ∀-terms. It may destroy a paired abstraction ∀-term, being the inverse of *mk_∀*.

Definition
```
dest_∀  ⌜∀ varstruct• body⌝ = (⌜varstruct⌝, ⌜body⌝)
```

Errors
```
4017   ?0 is not of form: ⌜∀ vs• t⌝
```

SML
```
val dest_∃1 : TERM -> (TERM * TERM);
```

Description A derived term destructor function for ∃_1-terms. It may destroy paired abstraction ∃_1-terms, being the inverse of *mk_∃_1*.

Definition
```
dest_∃1 ⌜∃1 varstruct• body⌝ =
        (⌜varstruct⌝, ⌜body⌝)
```

Errors
```
4021   ?0 is not of form: ⌜∃1 vs• t⌝
```

See Also *dest_simple_∃_1*

SML

val **dest_∃** : *TERM* −> (*TERM* ∗ *TERM*);

Description A derived term destructor function for ∃-terms. It may destroy paired abstraction ∃-terms, being the inverse of *mk_∃*.

Definition

dest_∃ ⌜∃ *varstruct*• *body*⌝ = (⌜*varstruct*⌝, ⌜*body*⌝)

Errors

4020 ?0 is not of form: ⌜∃ *vs*• *t*⌝

See Also *dest_simple_∃*

SML

val **dest_×_type** : *TYPE* −> (*TYPE* ∗ *TYPE*)

Description *dest_×_type* ⌜:*ty_1* × *ty_2*⌝ returns (⌜:*ty_1*⌝,⌜:*ty_2*⌝).

Errors

4018 ?0 is not of the form ⌜:*ty1* × *ty2*⌝

SML

val **dest_ϵ** : *TERM* −> (*TERM* ∗ *TERM*);

Description A derived term destructor function for ϵ-terms.

Definition

dest_ϵ ⌜ϵ *varstruct*• *body*⌝ = (⌜*varstruct*⌝, ⌜*body*⌝)

Errors

4023 ?0 is not of form: ⌜ϵ *vs*• *t*⌝

SML

val **dest_λ**: *TERM* −> (*TERM* ∗ *TERM*);

Description Destroys a λ-abstraction. It can destroy paired λ-abstractions, being an inverse of *mk_λ*.

Definition

dest_λ ⌜λ *vs*• *t*⌝ = (⌜*vs*⌝, ⌜*t*⌝)

See Also *dest_simple_λ*

Errors

4002 ?0 is not of form: ⌜λ *vs*• *t*⌝

Further details of the errors will be given, before the above exceptions are raised.

SML

val **dest_ℕ** : *TERM* −> *INTEGER*;

Description Destroy a numeric literal.

Example

dest_ℕ ⌜*5*⌝ = *5*;

Errors

3026 ?0 is not a numeric literal

SML

val **equality** : *TYPE* −> *TYPE* −> *TERM*;

Description Returns the constant ⌜$ = ⌝ upon terms with the first type argument. The second type is a dummy argument, present only to make the function have an acceptable signature for certain other functions.

SML

val **frees** : *TERM −> TERM list*;

Description Extract the free term variables within the term argument. The resulting variables will be in reverse order of first occurrence (for a term viewed without fixity properties, such as infix variables).

See Also *dest_frees*

SML

val **gen_vars** : *TYPE list −> TERM list −> TERM list*;

Description *gen_vars tyl tml* generates a list of differently named term variables, with the types in *tyl*, whose names are not present within any of the terms in *tml* as variable names.

It will be much faster to make one call to this function with a list of types, than to make the equivalent number of individual calls.

SML

val **get_variant_suffix** : *unit −> string*;

Description Returns the string control *variant_suffix* used to create variant names in *string_variant* (q.v.) and its relatives. The string is set by *set_variant_suffix* (q.v.).

SML

val **inst_type** : *((TYPE ∗ TYPE) list) −> TYPE −> TYPE*;

Description *inst_type alist type* recursively descends through *type*, replacing any type variables by whatever the association list *alist* associates with them. If the association list does not contain a type variable found in *type*, then that type variable will not be changed. Replaced types are **not** recursively processed by this function.

Errors

3019 ?0 is not a type variable

SML

val **inst** : *TERM list −> (TYPE ∗ TYPE) list −> TERM −> TERM*;

Description *inst avlist slist term* instantiates the type variables of *term* with the associated types found in *slist*. An element of *slist* will be (*return*, *tv*), where *tv* is a type variable that is to be instantiated to *return*. It will rename bound variables as necessary to prevent name capture problems. It will also not allow free variables to become the same as those in the avoidance list, *avlist*, or to become bound.

It partially evaluates with two arguments.

Errors

3007 ?0 is not a term variable
3019 ?0 is not a type variable
3020 Internal error in type instantiation (?0 would become bound)

SML

val **is_app** : *TERM −> bool*;

Description Return true only when the term is a function application (i.e. of form ⌜*f x*⌝), and false otherwise: no exceptions can be raised. Note that many derived term constructs, e.g. all quantifications, are also applications. Thus *is_app* ⌜∀ *x* • *t*⌝ will return *true*.

SML
```
val is_binder : string -> TERM -> bool;
```

Description *is_binder binder_nm tm* is true only when *tm* is of the form ⌜*binder*(λ *vs*• *body*)⌝, where *binder* is a constant whose name is *binder_nm*, and *vs* an allowed variable structure, and false otherwise. It cannot raise an exception.

See Also *is_simple_binder*

SML
```
val is_bin_op : string -> TERM -> bool;
```

Description *is_bin_op rator_nm term* returns true iff. *term* is of the form ⌜rator t1 t2⌝, and *rator* is a constant with name *rator_nm*. It cannot raise an exception.

Example
```
is_bin_op "∧" ⌜a ∧ b⌝ = true
```

SML
```
val is_char : TERM -> bool;
```

Description Return true only when the term is a character literal (e.g. ⌜'a'⌝), and false otherwise: no exceptions can be raised.

SML
```
val is_const : TERM -> bool;
```

Description Return true only when the term is a constant, and false otherwise: no exceptions can be raised. Note that even if the constant has not been declared, or has an inappropriate type it will still satisfy this predicate.

SML
```
val is_ctype : TYPE -> bool;
```

Description Return true only when the type is a compound type, and false otherwise: no exceptions can be raised. If the argument isn't a compound type then it must be a type variable.

SML
```
val is_empty_list : TERM -> bool;
```

Description Return true only when the term is an empty list-term, ⌜[]⌝, and false otherwise: no exceptions can be raised.

SML
```
val is_enum_set : TERM -> bool;
```

Description Return true only when the term is an enumerated set (i.e. of form ⌜{a; b; ...}⌝), and false otherwise: no exceptions can be raised.

SML
```
val is_eq : TERM -> bool;
```

Description Return true only when the term is an equation (i.e. of form ⌜a = b⌝or ⌜a ⇔ b⌝), and false otherwise: no exceptions can be raised.

SML
```
val is_float : TERM -> bool;
```

Description Return true when the term is a floating point literal. and false otherwise: no exceptions are raised.

SML

val **is_free_in** : *TERM* −> *TERM* −> *bool*;

Description *is_free_in v term* returns true iff. there is a free occurrence of *v* in *term*. It will raise an exception if the first argument is not a term variable.

Errors

3007 ?0 is not a term variable

SML

val **is_free_var_in** : (*string* ∗ *TYPE*) −> *TERM* −> *bool*;

Description Given a destroyed term variable, return true only when it is free within the term supplied as a second argument, and false otherwise: no exceptions can be raised.

SML

val **is_f** : *TERM* −> *bool*;

Description Return true only when the term is ⌜*F* : *BOOL*⌝, and false otherwise: no exceptions can be raised.

SML

val **is_if** : *TERM* −> *bool*;

Description Return true only when the term is a conditional (i.e. of form ⌜*if a then b else c*⌝), and false otherwise: no exceptions can be raised.

SML

val **is_let** : *TERM* −> *bool*;

Description Return true only when the term is a *let*-term (i.e. of form ⌜*let x* = *y in z*⌝), and false otherwise: no exceptions can be raised.

SML

val **is_list** : *TERM* −> *bool*;

Description Return true only when the term is a list-term (i.e. of form ⌜[*a*; *b*; ...]⌝), and false otherwise: no exceptions can be raised.

SML

val **is_mon_op** : *string* −> *TERM* −> *bool*;

Description *is_mon_op rator_nm term* returns true iff. *term* is of the form *rator t*, where *rator* is a constant with name *rator_nm*. It cannot raise an exception.

Example

is_mon_op "¬" ⌜¬ *t*⌝ = ⌜*t*⌝

SML

val **is_pair** : *TERM* −> *bool*;

Description Return true only when the term is a pair (i.e. of the form ⌜(*a*, *b*)⌝), and false otherwise: no exceptions can be raised.

SML

val **is_set_comp** : *TERM* −> *bool*;

Description Return true only when the term is a set comprehension (i.e. of form ⌜{*v* | *p*}⌝), and false otherwise: no exceptions can be raised.

SML
| *val* **is_simple_binder** : *string* −> *TERM* −> *bool*;

Description *is_simple_binder binder_nm term* returns true iff. argument *term* is of the form ⌜*binder*(λ *var* • *body*)⌝, where *binder* is a constant with the name *binder_nm*.

See Also *is_binder*

SML
| *val* **is_simple_∀** : *TERM* −> *bool*;

Description A derived term test for simple ∀-terms (i.e. of form ⌜∀ x • t⌝), not formed with paired abstractions.

See Also *is_∀*

SML
| *val* **is_simple_∃$_1$** : *TERM* −> *bool*;

Description Return true only when the term is a ∃_1-term (i.e. of form ⌜$\exists_1$ x • t⌝), formed only by simple abstraction, and false otherwise: no exceptions can be raised.

See Also *is_∃_1*

SML
| *val* **is_simple_∃** : *TERM* −> *bool*;

Description A derived term test for ∃-terms (i.e. of form ⌜∃ x • t⌝), not formed with paired abstractions.

See Also *is_∃*

SML
| *val* **is_simple_λ** : *TERM* −> *bool*;

Description Is the term a simple λ-abstraction (i.e. of form ⌜λ x • t⌝).

See Also *is_λ*

SML
| *val* **is_string** : *TERM* −> *bool*;

Description Return true only when the term is a string literal (e.g. ⌜"*abc*"⌝), and false otherwise: no exceptions can be raised.

SML
| *val* **is_type_instance** : *TYPE* −> *TYPE* −> *bool*;

Description *is_type_instance ty_1 ty_2* returns true iff *ty_1* is an instance of *ty_2*. It cannot raise an exception.

SML
| *val* **is_t** : *TERM* −> *bool*;

Description Return true only when the term is ⌜*T* : *BOOL*⌝, and false otherwise: no exceptions can be raised.

SML
| *val* **is_vartype** : *TYPE* −> *bool*;

Description Return true only when the type is a type variable, and false otherwise: no exceptions can be raised. If the argument isn't a type variable then it must be a compound type.

SML
```
val is_var : TERM -> bool;
```

Description Return true only when the term is a variable, and false otherwise: no exceptions can be raised.

SML
```
val is_∅ : TERM -> bool;
```

Description Return true only when the term is an empty enumerated set, ⌜∅⌝, and false otherwise: no exceptions can be raised.

SML
```
val is_⇔ : TERM -> bool;
```

Description Return true only when the term is a bi-implication (i.e. of form ⌜$a \Leftrightarrow b$⌝), and false otherwise: no exceptions can be raised. N.B. this may be successfully applied to boolean equations.

SML
```
val is_→_type : TYPE -> bool;
```

Description Return true only when the type is a function type, i.e. of form ⌜:*ty1* → *ty2*⌝, and false otherwise: no exceptions can be raised.

SML
```
val is_∧ : TERM -> bool;
```

Description Return true only when the term is a conjunction (i.e. of form ⌜$a \land b$⌝), and false otherwise: no exceptions can be raised.

SML
```
val is_∨ : TERM -> bool;
```

Description Return true only when the term is a disjunction (i.e. of form ⌜$a \lor b$⌝), and false otherwise: no exceptions can be raised.

SML
```
val is_¬ : TERM -> bool;
```

Description Return true only when the term is a negation (i.e. of form ⌜$\neg x$⌝), and false otherwise: no exceptions can be raised.

SML
```
val is_⇒ : TERM -> bool;
```

Description Return true only when the term is an implication (i.e. of form ⌜$a \Rightarrow b$⌝), and false otherwise: no exceptions can be raised.

SML
```
val is_∀ : TERM -> bool;
```

Description Return true only when the term is a ∀-term (i.e. of form ⌜∀ *vs* • *t*⌝), possibly formed with paired abstraction, and false otherwise: no exceptions can be raised.

See Also *is_simple_∀*

SML
```
val is_∃₁ : TERM -> bool;
```

Description Return true only when the term is a ∃_*1*-term (i.e. of form ⌜$\exists_1$ *vs* • *t*⌝), possibly formed with paired abstraction, and false otherwise: no exceptions can be raised.

See Also *is_simple_∃_1*

SML
```
val is_∃ : TERM -> bool;
```

Description Return true only when the term is a ∃-term (i.e. of form ⌜∃ *vs* • *t*⌝), possibly formed with paired abstraction, and false otherwise: no exceptions can be raised.

See Also *is_simple_∃*

SML
```
val is_×_type : TYPE -> bool;
```

Description Return true only when the type is a pair type, i.e. of the form: ⌜:*ty_1* × *ty_2*⌝, and false otherwise: no exceptions can be raised.

SML
```
val is_ϵ : TERM -> bool;
```

Description Return true only when the term is a ϵ-term (i.e. of form ⌜ϵ *vs* • *t*⌝), possibly formed with paired abstraction, and false otherwise: no exceptions can be raised.

SML
```
val is_λ : TERM -> bool;
```

Description This function returns true iff. the term is of the form ⌜λ *vs* • *t*⌝. It cannot raise exceptions.

See Also *is_simple_λ*

SML
```
val is_ℕ : TERM -> bool;
```

Description Return true only when the term is a numeric literal (e.g. ⌜*5*⌝), and false otherwise: no exceptions can be raised.

SML
```
val key_mk_const : (E_KEY * TYPE) -> TERM;
val key_dest_const : TERM -> E_KEY * TYPE;
```

Description Internally, the names of constants are represented using efficient dictionary keys. These functions allow the creation and destruction of constants by key rather than by name.

SML
```
val key_mk_ctype : E_KEY * TYPE list -> TYPE;
val key_dest_ctype : TYPE -> E_KEY * TYPE list;
```

Description Internally, the names of type constructors are represented using efficient dictionary keys. These functions allow the creation and destruction of compound types by key rather than by name.

SML
```
val list_mk_app : (TERM * TERM list) -> TERM;
```

Description Applies a function to multiple arguments.

Definition
```
list_mk_app (⌜t⌝, [⌜t1⌝,⌜t2⌝,⌜t3⌝,...]) = ⌜t t1 t2 t3 ...⌝
```

Failure May give rise to the error message from *mk_app*.

SML
```
val list_mk_binder : (TERM * TERM -> TERM) -> (TERM list * TERM)
        -> TERM;
```

Description If *maker* (⌜vs⌝,⌜b⌝) makes an abstraction ⌜bind vs • b⌝, then

```
list_mk_binder maker ([⌜vs_1⌝, ⌜vs_2⌝, ...],⌜body⌝)
```

returns ⌜*bind vs_1• bind vs_2• ... • body*⌝Notice that this can be used for implementing both simple and paired abstractions, with the *vs_i* being variable structures when so allowed, and otherwise variables.

SML
```
val list_mk_bin_op : string -> int -> int ->
        (TYPE -> TYPE -> TERM) -> TERM list -> TERM;
```

Description This function combines a list of terms using the given operator, as if by *mk_bin_op* (q.v). Notice the bracketing in the example.

Example
```
list_mk_bin_op area msg ∧_fun [⌜a⌝, ⌜b ∧ c⌝, ⌜d⌝] =
                ⌜a ∧ ((b ∧ c) ∧ d)⌝
```

where ∧*_fun* takes two (dummy) arguments and returns ⌜$∧⌝.

Errors
```
3017   An empty list argument is not allowed
```

Failure The failure message for failing to combine its arguments will be as *mk_bin_op* for the offending two arguments. If given an empty list the error will be from area *area*, but with message 3017.

SML
```
val list_mk_let : (((TERM * TERM)list)list * TERM) -> TERM
```

Description This generates a nested *let*-term.

Example
```
list_mk_let ([[(⌜x⌝,⌜1⌝)],[(⌜y⌝,⌜2⌝)]],⌜x+y⌝) =
        ⌜let x = 1 in let y = 2 in x+y⌝
```

SML
```
val list_mk_simple_λ : (TERM list * TERM) -> TERM;
```

Description λ-abstract a list of variables from a term.

Definition
```
list_mk_simple_λ ([⌜x1⌝,⌜x2⌝,...],⌜t⌝) = ⌜λ x1 x2 ... • t⌝
```

This function will be implemented using *mk_simple_λ*(q.v), not *mk_λ*.

See Also *list_mk_λ*

Failure May give rise to the error message from *mk_simple_λ*.

SML

|*val* **list_mk_simple_∀** : *TERM list* ∗ *TERM* −> *TERM*;

Description Universally quantify a term with a list of variables.

Definition

|*list_mk_simple_∀* ([⌜*x1*⌝,⌜*x2*⌝,...], ⌜*body*⌝) = ⌜∀ *x1 x2* ... • *body*⌝

This uses *mk_simple_∀* (q.v) to generate its result. Note that giving an empty list paired with a non-boolean will return that term, rather than fail.

See Also *list_mk_∀*

Failure This may give *mk_simple_∀* error messages.

SML

|*val* **list_mk_simple_∃** : *TERM list* ∗ *TERM* −> *TERM*;

Description Existentially quantify a term with a list of variables.

Definition

|*list_mk_simple_∃* ([⌜*x1*⌝,⌜*x2*⌝,...], ⌜*body*⌝) = ⌜∃ *x1 x2* ... • *body*⌝

This uses *mk_simple_∃* (q.v) to generate its result. Note that giving an empty list paired with a non-boolean will return that term, rather than fail.

See Also *list_mk_∃*

Failure This may give *mk_simple_∃* error messages.

SML

|*val* **list_mk_→_type** : *TYPE list* −> *TYPE*;

Description Create the type of a multi-argument function.

Definition

|*list_mk_→_type* [⌜:*ty1*⌝,...,⌜*tyn*⌝] =
| ⌜:*ty1* → ... → *tyn*⌝

The supplied list may not be empty.

Errors

|*3017 An empty list argument is not allowed*

SML

|*val* **list_mk_∧** : *TERM list* −> *TERM*;

Description Conjoin a list of terms:

Definition

|*list_mk_∧* [⌜*a*⌝, ⌜*b*⌝, ⌜*c*⌝,...] = ⌜*a* ∧ *b* ∧ *c* ...⌝

Errors

|*3017 An empty list argument is not allowed*
|*3031 ?0 is not of type* ⌜:*BOOL*⌝

SML

|*val* **list_mk_∨** : *TERM list* −> *TERM*;

Description A function to make a disjunction of a list of terms.

Definition

|*list_mk_∨* [⌜*a*⌝, ⌜*b*⌝, ⌜*c*⌝,...] = ⌜*a* ∨ *b* ∨ *c* ...⌝

Errors

|*3017 An empty list argument is not allowed*
|*3031 ?0 is not of type* ⌜:*BOOL*⌝

SML
val **list_mk_⇒** : *TERM list* −> *TERM*;

Description Makes a multiple implication term, using *mk_⇒* (q.v.).

Definition
list_mk_⇒ [⌜*t1*⌝,⌜*t2*⌝,...,⌜*tn*⌝] = ⌜*t1* ⇒ *t2* ⇒ ... ⇒ *tn*⌝

Note that giving a singleton list containing a non-boolean will return that term, rather than fail.

Errors
3015 ?1 is not of type ⌜:*BOOL*⌝
3017 An empty list argument is not allowed
3031 ?0 is not of type ⌜:*BOOL*⌝

SML
val **list_mk_∀** : *TERM list* ∗ *TERM* −> *TERM*;

Description Repeatedly universally quantify a term.

Definition
list_mk_∀ ([⌜*a*⌝,⌜*b*⌝,⌜*c*⌝,...], ⌜*body*⌝) = ⌜∀ *a b c* ...• *body*⌝

This uses *mk_∀* to generate its result.

Failure This may give the errors of *mk_∀*.

SML
val **list_mk_∃** : *TERM list* ∗ *TERM* −> *TERM*;

Description Repeatedly existentially quantify a term.

Definition
list_mk_∃ ([⌜*a*⌝,⌜*b*⌝,⌜*c*⌝,...], ⌜*body*⌝) = ⌜∃ *a b c* ...• *body*⌝

This uses *mk_∃* to generate its result.

Failure This may give the errors of *mk_∃*.

SML
val **list_mk_ϵ** : *TERM list* ∗ *TERM* −> *TERM*;

Description Repeatedly apply ϵ to a term.

Definition
list_mk_ϵ ([⌜*a*⌝,⌜*b*⌝,⌜*c*⌝,...], ⌜*body*⌝) = ⌜ϵ *a b c* ...• *body*⌝

Failure This may give the errors of *mk_ϵ*.

SML
val **list_mk_λ** : (*TERM list* ∗ *TERM*) −> *TERM*;

Description Repeatedly λ-abstract from a term.

Definition
list_mk_λ ([⌜*a*⌝,⌜*b*⌝,⌜*c*⌝,...], ⌜*body*⌝) = ⌜λ *a b c* ...• *body*⌝

This function is implemented using *mk_λ*, not *mk_simple_λ*.

See Also *list_mk_simple_λ*

Failure May give rise to the error message from *mk_λ*.

SML
```
val list_term_union : (TERM list list) -> TERM list;
```

Description Take the union of a number of lists of terms viewed as sets, removing any α-convertible duplicates.

See Also *list_union* for precise ordering of result.

SML
```
val list_variant : TERM list -> TERM list -> TERM list;
```

Description *list_variant stoplist vlist* returns a list of variants of the list of variables *vlist*, whose names are not present in the *stoplist*, which is also a list of term variables. No names are duplicated, the function returning one new variable for each member of *vlist*. The variants are generated by sufficient appending of the variant string (see *set_variant_string*).

Errors
```
3007 ?0 is not a term variable
```

SML
```
val mk_app : (TERM * TERM) -> TERM;
```

Description This produces a function application.

Definition
```
mk_app (⌜f⌝, ⌜t⌝) = ⌜f t⌝
```

Note that many derived term constructs, e.g. all quantifications, are also applications. Thus

Example
```
mk_app (⌜$∀⌝, ⌜λ x • t⌝) = ⌜∀ x • t⌝
```

Errors
```
3005 Cannot apply ?0 to ?1 as types are incompatible
3006 Type of ?0 not of form ⌜:ty1 → ty2⌝
```

SML
```
val mk_binder : string -> int -> (TYPE -> TYPE -> TERM) ->
        (TERM * TERM) -> TERM;
```

Description A generic method of implementing binder constructor functions:

Definition
```
mk_binder area msg binder_nm (⌜varstruct⌝, ⌜body⌝) =
        ⌜binder′(λ varstruct• body)⌝ =
        ⌜binder′ varstruct• body⌝
```

binder′ is formed by applying *binder* to the types of the *varstruct* and *body*. *varstruct* may be any allowed variable structure.

See Also *mk_simple_binder*

Errors
```
4016 ?0 is not an allowed variable structure
```

Failure If the term cannot be made, then the error will be from *area*, with a message indexed by *msg*. If the first term argument is not an allowed variable structure then failure 4016 is raised from area *area*.

SML
```
val mk_bin_op : string -> int -> int -> (TYPE -> TYPE -> TERM) ->
        (TERM * TERM) -> TERM;
```

Description *mk_bin_op area msg1 msg2 rator_fn* (t_1, t_2) attempts to form ⌜*t_1 rator t_2*⌝. *rator′* is gained by applying *rator_fn* to the types of *t_1* and *t_2*.

Example
```
mk_bin_op "mk_∧" 3031 3015 (fn _ => fn _ => ⌜$∧⌝) (⌜a⌝, ⌜b⌝) = ⌜a ∧ b⌝
```

Failure The failure message for failing to apply *rator* to the first term will be from area *area*, and will have the text indexed by *msg1*, with the two terms as strings for arguments. If the failure is from applying the rators plus first term to the second term the error message will be from area *area*, and will have the text indexed by *msg2*, with the two terms as strings for arguments. It is not unusual for one of these strings of terms to be thrown away by the message *msg2* provided by the caller of this function.

SML
```
val mk_char : string -> TERM;
```

Description Construct a character literal.

Example
```
mk_char "a" = ⌜‘a‘⌝
```

Errors
```
3023  String ?0 is not a single character
```

SML
```
val mk_const : (string * TYPE) -> TERM;
```

Description This produces a constant.

Definition
```
mk_const("c",⌜:ty⌝) = ⌜c : ty⌝
```

The function makes no checks against the declaration of the constant, the declaration of the type constructors of the type supplied, or the appropriateness of the type supplied: see *get_const_info* (q.v.). However it will not form constants whose types clash with those constants required by the implementation of the abstract data type *THM* (q.v.). These are =, ⇒, ∀, and ∃.

Errors
```
3002  Type of constant with name "=" must be of form:⌜ty1 → ty1 → BOOL⌝
3003  Type of constant with name "⇒" must be of form:⌜BOOL → BOOL → BOOL⌝
3004  Type of constant with name ?0 must be of form:⌜(ty1 → BOOL) → BOOL⌝
```

SML
```
val mk_ctype : string * TYPE list -> TYPE;
```

Description Create a compound type from a type constructor and sufficient arguments. The function makes no checks against the declaration or arity of the type constructor or the type arguments: see *get_type_info* (q.v.).

Definition
```
mk_ctype ("tc", [⌜:ty1⌝,⌜:ty2⌝,...]) = ⌜:(ty1,ty2,...)tc⌝
mk_ctype ("tc", [⌜:ty⌝]) = ⌜:ty tc⌝
mk_ctype ("tc", []) = ⌜:tc⌝
```

SML
```
val mk_empty_list : TYPE -> TERM
```

Description A derived term constructor function for generating an empty list term with elements of a given type.

Definition
```
mk_empty_list ⌜: ty⌝ = ⌜[] : ty LIST⌝
```

See Also *mk_list*

SML
```
val mk_enum_set : TERM list -> TERM
```

Description A derived term constructor function for generating enumerated sets. The argument is a list of the members of the set. The type of a set of elements of type ⌜:*TY*⌝ is ⌜:*TY SET*⌝. If the term list is empty the function will fail (see *mk_∅*). The set must be of terms with the same HOL type.

Definition
```
mk_enum_set [⌜a⌝, ⌜b⌝, ...] = ⌜{a; b; ...}⌝
```

Errors
```
3012   ?0 and ?1 do not have the same types
3017   An empty list argument is not allowed
```

SML
```
val mk_eq : (TERM * TERM) -> TERM;
```

Description A derived term constructor function for generating equations.

Definition
```
mk_eq (⌜a⌝,⌜b⌝) = ⌜a = b⌝
mk_eq (⌜a:BOOL⌝,⌜b:BOOL) = ⌜a ⇔ b⌝
```

Errors
```
3012   ?0 and ?1 do not have the same types
```

SML
```
val mk_float : INTEGER * INTEGER * INTEGER -> TERM;
```

Description Make a floating point literal.

Definition
```
mk_float (⌜x⌝, 0, ⌜0⌝) = ⌜XX.⌝
mk_float (⌜x⌝, ⌜p⌝, ⌜0⌝) = ⌜XX.YY⌝
mk_float (⌜x⌝, ⌜p⌝, ⌜z⌝) = ⌜XX.YYeZZ⌝
```

where *XX.YY* is the decimal representation of $x \times 10^{-p}$ and *ZZ* is the decimal representation of z (with $p = z = 0$ in the first case and $z = 0$ in the second).

Errors
```
4041   the mantissa of a HOL floating point literal must be non−negative
```

SML
```
val mk_f : TERM;
```

Description The term ⌜*F* : *BOOL*⌝.

SML
```
val mk_if : (TERM * TERM * TERM) -> TERM;
```

Description Make a conditional.

Definition
```
mk_if (⌜c⌝, ⌜y⌝, ⌜n⌝) = ⌜if c then y else n⌝
```

Errors
```
3012  ?0 and ?1 do not have the same types
3031  ?0 is not of type ⌜:BOOL⌝
```

SML
```
val mk_let : ((TERM * TERM)list * TERM) -> TERM
```

Description A derived term constructor function for generating *let*-terms. The arguments may have any form allowed by ICL HOL Concrete Syntax. Thus they may be variable structures formed by pairing, or single clause, non-recursive functions, whose arguments may only be variable structures formed by pairing.

Example
```
mk_let ([],⌜x⌝) = ⌜x⌝
mk_let([(⌜x⌝, ⌜1⌝)], ⌜x+1⌝) = ⌜let x = 1 in x + 1⌝
mk_let([(⌜x⌝, ⌜1⌝),(⌜y⌝, ⌜2⌝)], ⌜x+y⌝) =
        ⌜let x = 1 and y = 2 in x + y⌝
mk_let([(⌜(x,y)⌝, ⌜(1,2)⌝)], ⌜x+y⌝) =
        ⌜let (x,y) = (1,2) in x + y⌝
mk_let([(⌜(x,y)⌝, ⌜(1,2)⌝)], ⌜x+y⌝) =
         ⌜let (x,y) = (1,2) in x + y⌝
mk_let([(⌜f (x,y)⌝, ⌜(1,2)⌝)], ⌜x+y⌝) =
        ⌜let f = λ (x,y)• (1,2) in x + y⌝
```

Errors
```
3012  ?0 and ?1 do not have the same types
4007  ?0 is not a well−formed LHS for mk_let
```

SML
```
val mk_list : TERM list -> TERM
```

Description A derived term constructor function for generating list-terms. The argument is a list of the members of the list. If the term list is empty the function will fail (see *mk_empty_list*). The list must be of terms with the same HOL type.

Definition
```
mk_list [⌜a⌝, ⌜b⌝, ...] = ⌜[a; b; ...]⌝
```

Errors
```
3012  ?0 and ?1 do not have the same types
3017  An empty list argument is not allowed
```

SML

val **mk_mon_op** : *string* −> *int* −> (*TYPE* −> *TERM*) −>
TERM −> *TERM*;

Description *mk_mon_op area msg rator_fn* ⌜*rand*⌝ attempts to form the term ⌜*rator rand*⌝. ⌜*rator*⌝ is gained by applying *rator_fn* to the type of ⌜*rand*⌝.

Example

mk_mon_op "*mk_¬*" *3031* (*fn* _ => ⌜$¬⌝) ⌜*t*:*BOOL*⌝ = ⌜¬ *t*⌝

Failure The failure message for failing to apply *rator* to its arguments will be from area *area*, and will have the text indexed by *msg*.

SML

val **mk_multi_¬** : (*int* ∗ *TERM*) −> *TERM*;

Description *mk_multi_¬* (*n*, *t*) will apply the constructor *mk_¬* n times to *t*.

Example

mk_multi_¬ (*2*, ⌜*T*⌝) = ⌜¬(¬ *T*)⌝

Errors

3031 ?*0 is not of type* ⌜:*BOOL*⌝
4030 ?*0 is negative*

SML

val **mk_pair** : (*TERM* ∗ *TERM*) −> *TERM*;

Description A derived term constructor function for generating pairs.

Definition

mk_pair(⌜*t1*⌝, ⌜*t2*⌝) = ⌜(*t1*, *t2*)⌝

SML

val **mk_set_comp** : (*TERM* ∗ *TERM*) −> *TERM*

Description A derived term constructor function for generating set comprehensions.

Example

mk_set_comp (⌜*x*⌝, ⌜*x* > *5*⌝) = ⌜{ *x* | *x* > *5*}⌝

Errors

3015 ?*1 is not of type* ⌜:*BOOL*⌝
4016 ?*0 is not an allowed variable structure*

SML

val **mk_simple_binder** : *string* −> *int* −> (*TYPE* −> *TYPE* −> *TERM*) −>
(*TERM* ∗ *TERM*) −> *TERM*;

Description *mk_simple_binder area msg binder_fn* (*var*, *body*) generates the term:

⌜*binder*(λ*var* • *body*)⌝

where *binder* is *binder_fn* applied to the types of *var* and *body*. *var* must be a term variable.

See Also *mk_binder*

Errors

3007 ?*0 is not a term variable*

Failure If the term cannot be made, then the error will be from *area*, with a message indexed by *msg*, and the two terms as string arguments. If the first of the pair of terms is not a variable then error 3007 will be given from area *area*.

SML
```
val mk_simple_term : DEST_SIMPLE_TERM -> TERM;
```

Description Create a well-formed TERM from a statement of a top-level structure, and the associated constituent parts.

It makes the same checks as *mk_const*, *mk_app*, etc(q.v.), and gives the same error messages as these if there is a failure.

See Also *DEST_SIMPLE_TERM*

Errors
```
3005   Cannot apply ?0 to ?1 as types are incompatible
3006   Type of ?0 not of form ⌜:ty1 → ty2⌝
3007   ?0 is not a term variable
```

SML
```
val mk_simple_type : DEST_SIMPLE_TYPE -> TYPE;
```

Description This function constructs a HOL type from something of type *SIMPLE_DEST-_TYPE* (q.v).

SML
```
val mk_simple_∀ : (TERM * TERM) -> TERM;
```

Description A derived term constructor function for generating simple ∀-terms.

Definition
```
mk_simple_∀ (⌜var⌝, ⌜body⌝) = ⌜∀ var • body⌝
```

var must be a term variable.

See Also *mk_∀*

Errors
```
3007   ?0 is not a term variable
3015   ?1 is not of type ⌜:BOOL⌝
```

SML
```
val mk_simple_∃₁ : (TERM * TERM) -> TERM;
```

Description A derived term constructor function for generating simply abstracted ∃_1-terms.

Definition
```
mk_simple_∃₁ (⌜var⌝, ⌜body⌝) =
        ⌜∃₁ var• body⌝
```

var must be a variable.

Errors
```
3007   ?0 is not a term variable
3015   ?1 is not of type ⌜:BOOL⌝
```

See Also *mk_∃_1*

SML

|*val* **mk_simple_∃** : (*TERM* ∗ *TERM*) −> *TERM*;

Description A derived term constructor function for generating simple ∃-terms.

Definition

|*mk_simple_∃* (⌜*var*⌝, ⌜*body*⌝) = ⌜∃ *var* • *body*⌝

var must be a term variable.

See Also *mk_∃*

Errors

|*3007* *?0 is not a term variable*
|*3015* *?1 is not of type* ⌜:*BOOL*⌝

SML

|*val* **mk_simple_λ** : (*TERM* ∗ *TERM*) −> *TERM*;

Description This produces a simple λ-abstraction. It may only abstract variables.

Definition

|*mk_simple_λ* (⌜*v*⌝, ⌜*t*⌝) = ⌜λ *v* • *t*⌝

See Also *mk_λ*

Errors

|*3007* *?0 is not a term variable*

SML

|*val* **mk_string** : *string* −> *TERM*;

Description Construct a string literal.

Example

|*mk_string* "*abc*" = ⌜"*abc*"⌝

SML

|*val* **mk_term** : *DEST_TERM* −> *TERM*

Description Create a term from a derived term. It is an inverse to *dest_term* (q.v), and therefore understands how to handle paired abstractions.

The function is implemented using the individual primitive and derived term constructors (e.g. *mk_const* and *mk_∀*), with what checks they use.

Failure This function will fail with the same messages as the appropriate term constructor functions.

SML

|*val* **mk_t** : *TERM*;

Description The term ⌜*T* : *BOOL*⌝.

SML

|*val* **mk_vartype** : *string* −> *TYPE*;

Description Create a HOL type variable from a string:

Definition

|*mk_vartype* "′*tv*" = ⌜: ′*tv*⌝

SML

```
val mk_var : (string * TYPE) -> TERM;
```

Description This produces a term variable.

The function makes no checks against the declaration of the subtypes of the type supplied.

Definition

```
mk_var("v",⌜:ty⌝) = ⌜v : ty⌝
```

SML

```
val mk_∅ : TYPE -> TERM;
```

Description A derived term constructor function for generating an empty (enumerated) set with elements of a given type.

Definition

```
mk_∅ ⌜: ty⌝ = ⌜∅ : ty SET⌝
```

See Also *mk_enum_set*

SML

```
val mk_⇔ : (TERM * TERM) -> TERM;
```

Description A derived term constructor function for generating bi-implications.

Definition

```
mk_⇔ (⌜t1⌝, ⌜t2⌝) = ⌜t1 ⇔ t2⌝
```

Errors

```
3015 ?1 is not of type ⌜:BOOL⌝
3031 ?0 is not of type ⌜:BOOL⌝
```

SML

```
val mk_→_type : (TYPE * TYPE) -> TYPE;
```

Description Create a function type from two types. A function type is just a kind of compound type.

Definition

```
mk_→_type (⌜:ty1⌝, ⌜:ty2⌝) =
        mk_ctype("→",[⌜:ty1⌝,⌜:ty2⌝]) = ⌜:ty1 → ty2⌝
```

SML

```
val mk_∧ : (TERM * TERM) -> TERM;
```

Description A derived term constructor function for generating conjunctions.

Definition

```
mk_∧ (⌜t1⌝, ⌜t2⌝) = ⌜t1 ∧ t2⌝
```

Errors

```
3015 ?1 is not of type ⌜:BOOL⌝
3031 ?0 is not of type ⌜:BOOL⌝
```

SML

```
val mk_∨ : (TERM * TERM) -> TERM;
```

Description A derived term constructor function for generating disjunctions.

Definition

```
mk_∨ (⌜t1⌝, ⌜t2⌝) = ⌜t1 ∨ t2⌝
```

Errors

```
3015 ?1 is not of type ⌜:BOOL⌝
3031 ?0 is not of type ⌜:BOOL⌝
```

SML

val **mk_¬** : *TERM* −> *TERM*;

Description A derived term constructor function for generating negations.

Definition

mk_¬ ⌜*t*⌝ = ⌜¬ *t*⌝

Errors

3031 ?0 is not of type ⌜:*BOOL*⌝

SML

val **mk_⇒** : (*TERM* ∗ *TERM*) −> *TERM*;

Description A derived term constructor function for generating implications. It takes two arguments: the antecedent and the consequent.

Definition

mk_⇒ (⌜*a*⌝,⌜*b*⌝) = ⌜*a* ⇒ *b*⌝

Errors

3015 ?1 is not of type ⌜:*BOOL*⌝
3031 ?0 is not of type ⌜:*BOOL*⌝

SML

val **mk_∀** : (*TERM* ∗ *TERM*) −> *TERM*;

Description A derived term constructor function for generating ∀-terms.

Definition

mk_∀ (⌜*varstruct*⌝, ⌜*body*⌝) = ⌜∀ *varstruct*• *body*⌝

varstruct may be any allowed variable structure.

Errors

3015 ?1 is not of type ⌜:*BOOL*⌝
4016 ?0 is not an allowed variable structure

See Also *mk_simple_∀*

SML

val **mk_∃₁** : (*TERM* ∗ *TERM*) −> *TERM*;

Description A derived term constructor function for generating ∃_*1*-terms.

Definition

*mk_*$\exists_1$ (⌜*varstruct*⌝, ⌜*body*⌝) =
⌜$\exists_1$ *varstruct*• *body*⌝

varstruct may be any allowed variable structure.

Errors

3015 ?1 is not of type ⌜:*BOOL*⌝
4016 ?0 is not an allowed variable structure

See Also *mk_∃_1*

SML

$|$ *val* **mk_∃** : (*TERM* ∗ *TERM*) −> *TERM*;

Description A derived term constructor function for generating ∃-terms.

Definition

$|$ *mk_∃* (⌜*varstruct*⌝, ⌜*body*⌝) = ⌜∃ *varstruct*• *body*⌝

varstruct may be any allowed variable structure.

Errors

$|$ *3015* *?1 is not of type* ⌜:*BOOL*⌝

$|$ *4016* *?0 is not an allowed variable structure*

See Also *mk_simple_∃*

SML

$|$ *val* **mk_×_type** : (*TYPE* ∗ *TYPE*) −> *TYPE*

Description *mk_×_type* (⌜:*ty_1*⌝,⌜:*ty_2*⌝) returns a pair type: ⌜:*ty_1* × *ty_2*⌝.

SML

$|$ *val* **mk_ϵ** : (*TERM* ∗ *TERM*) −> *TERM*;

Description A derived term constructor function for generating ϵ-terms.

Definition

$|$ *mk_ϵ* (⌜*varstruct*⌝, ⌜*body*⌝) = ⌜ϵ *varstruct*• *body*⌝

varstruct may be any allowed variable structure.

Errors

$|$ *3015* *?1 is not of type* ⌜:*BOOL*⌝

$|$ *4016* *?0 is not an allowed variable structure*

SML

$|$ *val* **mk_λ** : *TERM* ∗ *TERM* −> *TERM*

Description This creates a λ-abstraction of an allowed variable structure from a term.

Example

$|$ *mk_λ* (⌜*x*⌝ , ⌜*x* + *y*⌝) = ⌜λ *x*• *x* + *y*⌝

$|$ *mk_λ* (⌜(*x*, *y*)⌝, ⌜*x* + *y*⌝) = ⌜λ (*x*, *y*)• *x* + *y*⌝

$|$ *mk_λ* (⌜((*x1*,*x2*), (*y1*,*y2*))⌝, ⌜*x2* + *y2*⌝) = ⌜λ ((*x1*,*x2*), (*y1*,*y2*))• *x2* + *y2*⌝

See Also *mk_simple_λ*

Errors

$|$ *4016* *?0 is not an allowed variable structure*

SML

$|$ *val* **mk_ℕ** : *INTEGER* −> *TERM*;

Description Construct a numeric literal: the argument may not be negative.

Example

$|$ *mk_ℕ 5* = ⌜*5*⌝

Errors

$|$ *3021* *?0 should be 0 or positive*

SML

$|$ *val* **quantifier** : *string* −> *TYPE* −> *TYPE* −> *TERM*;

Description *quantifier name type dummy* returns a constant, with the given name, and type ⌜:(*type*→*BOOL*)→*BOOL*⌝, This is an appropriate type for binders. The dummy is present only to make the function have an acceptable signature for certain other functions.

SML
| *val* **rename** : (*string* ∗ *TYPE*) −> *string* −> *TERM* −> *TERM*;

Description *rename* (*oname*, *type*) *cname term* returns a term based on *term*, but with any free variables with name *oname*, and type *type* renamed to *cname*.

SML
| *val* **set_variant_suffix** : *string* −> *string*;

Description Sets the string control *variant_suffix* used to create variant names in *string-_variant* (q.v.) and its relatives. The string is initially a single prime character. The function returns the previous setting of the control.

Errors
| *3028* *string may not be empty*

SML
| *val* **string_of_term** : *TERM* −> *string*;

Description This returns a display of a term in the form of a string, with no inserted new lines, suitable for use with *diag_string* and *fail*.

See Also *format_term* is a formatted string display of a term.

SML
| *val* **string_of_type** : *TYPE* −> *string*;

Description This returns a display of a type in the form of a string, with no inserted new lines, suitable for use with *diag_string* and *fail*.

See Also *format_type* is a formatted string display of a type.

SML
| *val* **string_variant** : *string list* −> *string* −> *string*;

Description *string_variant vlist name* returns a string that is a different from any name in *vlist*. Variants are formed by repeatedly appending the variant string(see *set_variant_string*) to the *name*. Note that *string_variant* [] *name* gives *name*.

Uses Somewhat faster than *variant* if term variables are already destroyed, and their names and types are directly accessible.

See Also *variant*

SML
| *val* **STRING** : *TYPE*;

Description This is the HOL type of strings, a type abbreviation for lists of objects of type *CHAR*.

Definition
| *val STRING* = ⌜:*CHAR LIST*⌝;

See Also Theory “char”.

SML
| *val* **strip_app** : *TERM* −> *TERM* ∗ *TERM list*;

Description Splits a term into a head term, that is not an application, and the list of argument terms, if any, to which that head term was applied.

Example
| *strip_app* ⌜*t t1 t2 t3* ...⌝ = (⌜*t*⌝, [⌜*t1*⌝,⌜*t2*⌝,⌜*t3*⌝,...])
| *strip_app* ⌜*T*⌝ = (⌜*T*⌝,[])

SML
```
val strip_binder : string -> TERM -> TERM list * TERM;
```

Description *strip_binder binder* applied to

⌜*binder*(λ*vs_1*•*binder*(λ*vs_2*•...•*body*)...)⌝

will return

[⌜*vs_1*⌝,⌜*vs_2*⌝,...],⌜*body*⌝

where the *vs_i* are allowed variable structures. The function acts as *dest_binder* (q.v), and will handle paired abstraction terms. It will return an empty list and the original term if the supplied term is not formed using the binder.

See Also *strip_simple_binder*

SML
```
val strip_bin_op : string -> TERM -> TERM list
```

Description This function strips a binary operator, attempting to destroy its term argument, and recursively stripping to the right, as if by *dest_bin_op*. A term not formed from the operator is returned unchanged, as a singleton list.

Example
```
strip_bin_op "∧" ⌜a ∧ (b ∧ c) ∧ d⌝ = [⌜a⌝, ⌜b ∧ c⌝, ⌜d⌝]
```

SML
```
val strip_leaves : ('a -> 'a * 'a) -> 'a -> 'a list;
```

Description Given a function that destroys an object into a pair of objects (and here we are thinking of, for example, *dest_*∧), recursively descend the results of destruction down both branches, destroying until failure.

Example
```
strip_leaves dest_∧ ⌜(a ∧ b) ∧ c ∧ d⌝ =
        [⌜a⌝, ⌜b⌝, ⌜c⌝, ⌜d⌝]
```

SML
```
val strip_let : TERM -> ((TERM * TERM)list)list * TERM
```

Description This destroys a sequence of nested *let* constructs.

Example
```
strip_let ⌜let x = 1 in let y = 2 in x+y⌝ =
        ([[(⌜x⌝,⌜1⌝)],[(⌜y⌝,⌜2⌝)]],⌜x+y⌝)
```

SML
```
val strip_simple_binder : string -> TERM -> TERM list * TERM;
```

Description *strip_simple_binder binder* applied to

⌜*binder*(λ*v_1*•*binder*(λ*v_2*•...•*body*)...)⌝

will return

[⌜*v_1*⌝,⌜*v_2*⌝,...],⌜*body*⌝

where the *v_i* are simple variables. The function acts as *dest_simple_binder* (q.v), and will not handle paired abstraction terms. It will return an empty list and the original term if the supplied term is not formed using the binder.

See Also *strip_binder*

SML

val **strip_simple_∀** : *TERM* −> (*TERM list* ∗ *TERM*);

Description Strip a multiply universally simply quantified term.

Definition

strip_simple_∀ ⌜∀ *a b c* ...• *body*⌝ = [⌜*a*⌝,⌜*b*⌝,⌜*c*⌝,...], ⌜*body*⌝

SML

val **strip_simple_∃** : *TERM* −> (*TERM list* ∗ *TERM*);

Description Strip a repeatedly existentially simply quantified term.

Definition

strip_simple_∃ ⌜∃ *a b c* ...• *body*⌝ = [⌜*a*⌝,⌜*b*⌝,⌜*c*⌝,...], ⌜*body*⌝

SML

val **strip_spine_left** : ('*a* −> '*a* ∗ '*a*) −> '*a* −> '*a list*;

Description Given a function that destroys an object into a pair of objects (and here we are thinking of, for example, *dest_∧*), recursively descend the left results of destruction, destroying until failure.

Example

strip_spine_left dest_∧ ⌜(*a* ∧ *b*) ∧ *c* ∧ *d*⌝ =
[⌜*a*⌝, ⌜*b*⌝, ⌜*c* ∧ *d*⌝]

SML

val **strip_spine_right** : ('*a* −> '*a* ∗ '*a*) −> '*a* −> '*a list*;

Description Given a function that destroys an object into a pair of objects (and here we are thinking of, for example, *dest_∧*), recursively descend the right results of destruction, destroying until failure.

See Also *strip_bin_op* for stripping terms formed by binary (constant) term operators.

Example

strip_spine_left dest_∧ ⌜(*a* ∧ *b*) ∧ *c* ∧ *d*⌝ =
strip_∧ ⌜(*a* ∧ *b*) ∧ *c* ∧ *d*⌝ =
[⌜*a* ∧ *b*⌝, ⌜*c*⌝, ⌜*d*⌝]

SML

val **strip_→_type** : *TYPE* −> *TYPE list*;

Description Strip the type of a multi-argument function into its constituent types, only descending into the right hand result of *dest_→_type*.

Definition

strip_→_type ⌜:*ty1* → ... → *tyn*⌝ =
[⌜:*ty1*⌝,...,⌜*tyn*⌝]

SML

val **strip_∧** : *TERM* −> *TERM list*

Description Break a term into its constituent conjuncts, descending recursively only to the right.

Example

strip_∧ ⌜*a* ∧ (*b* ∧ *c*) ∧ *d*⌝ = [⌜*a*⌝,⌜*b* ∧ *c*⌝, ⌜*d*⌝]

SML

$|$*val* **strip_∨** : *TERM* −> *TERM list*

Description Break a term into its constituent disjuncts, descending recursively only to the right.

Example

$|strip\_\vee \ulcorner a \vee (b \vee c) \vee d\urcorner = [\ulcorner a\urcorner,\ulcorner b \vee c\urcorner, \ulcorner d\urcorner]$

SML

$|$*val* **strip_⇒** : *TERM* −> *TERM list*;

Description Strip a multiple implication into a list of antecedents appended to the singleton list of the innermost consequent.

Definition

$|strip\_\Rightarrow\ \ulcorner t1 \Rightarrow t2 \Rightarrow ... \Rightarrow tn\urcorner = [\ulcorner t1\urcorner,\ulcorner t2\urcorner,...\ulcorner tn\urcorner]$

Note that stripping a non-boolean will result in a singleton list containing that term, not a fail.

SML

$|$*val* **strip_∀** : *TERM* −> (*TERM list* ∗ *TERM*);

Description Strip a multiply universally quantified term (perhaps with paired abstractions).

Definition

$|strip\_\forall \ulcorner\forall\ a\ b\ c\ ...\bullet\ body\urcorner = [\ulcorner a\urcorner,\ulcorner b\urcorner,\ulcorner c\urcorner,...], \ulcorner body\urcorner$

SML

$|$*val* **strip_∃** : *TERM* −> (*TERM list* ∗ *TERM*);

Description Strip a repeatedly existentially quantified term, possibly formed with paired abstractions.

Definition

$|strip\_\exists \ulcorner\exists\ a\ b\ c\ ...\bullet\ body\urcorner = [\ulcorner a\urcorner,\ulcorner b\urcorner,\ulcorner c\urcorner,...], \ulcorner body\urcorner$

SML

$|$*val* **strip_ϵ** : *TERM* −> (*TERM list* ∗ *TERM*);

Description Strip multiple ϵ's.

Definition

$|strip\_\epsilon \ulcorner\epsilon\ a\ b\ c\ ...\bullet\ body\urcorner = [\ulcorner a\urcorner,\ulcorner b\urcorner,\ulcorner c\urcorner,...], \ulcorner body\urcorner$

SML

$|$*val* **strip_λ** : *TERM* −> (*TERM list* ∗ *TERM*);

Description Strip a multiple λ-abstraction.

Definition

$|strip\_\lambda \ulcorner\lambda\ a\ b\ c\ ...\bullet\ body\urcorner = [\ulcorner a\urcorner,\ulcorner b\urcorner,\ulcorner c\urcorner,...], \ulcorner body\urcorner$

This uses *dest_λ* (q.v.) rather than *dest_simple_λ*.

SML

val **subst** : $(TERM * TERM)$ *list* $->$ $TERM$ $->$ $TERM$;

Description *subst* $[(t\_1, u\_1), (t\_2, u\_2), \ldots]$ t returns the term formed from t by parallel substition of the t_i for the u_i. The u_i can be variables or arbitrary terms but only "free" occurrences of a u_i will be changed (i.e., only occurrences in which no free variable of u_i becomes a bound variable in t). Bound variables in t are renamed as neccessary to prevent bound variable capture.

If some u_i appears more than once in the substitution list, say $u\_i = u\_j$ for $i < j$, then the later pair (t_j, u_j) is ignored.

subst does not perform type instantiation: each t_i must have the same type as the corresponding u_i..

Definition

subst $[(\ulcorner t_1 \urcorner, \ulcorner u_1 \urcorner), (\ulcorner t_2, \ulcorner u_2 \urcorner), \ldots]\ \ulcorner t \urcorner = \ulcorner t[t_1/u_1,\ t_2/u_2,\ \ldots] \urcorner$

See Also *var_subst*

Errors

3012 ?0 and ?1 do not have the same types

SML

val **term_any** : $(TERM$ $->$ *bool*$)$ $->$ $TERM$ $->$ *bool*;

Description Given a predicate on terms, tests to see if any sub-term of some term (or the term itself) satisfies the predicate. The search ceases on the first satisfaction, rather than all the tests being done and the results combined.

SML

val **term_consts** : $TERM$ $->$ $($*string* $*$ $TYPE)$ *list*;

Description This function extracts the subterms of a term which are constants, giving destroyed constants in each case (duplicates are eliminated)

SML

val **term_diff** : $(TERM$ *list* $*$ $TERM$ *list*$)$ $->$ $TERM$ *list*;

Description Remove any terms in the first list that are α-convertible to any in the second. An infix function.

SML

val **term_fail** : *string* $->$ *int* $->$ $TERM$ *list* $->$ $'a$;

Description *term_fail area msg tml* first creates a list of functions from *unit* to *string*, using *string_of_term* (q.v.) providing displays of the list of terms. It then calls *fail* with the *area*, *msg* and this list of functions. This allows terms to be presented in error messages.

SML

val **term_fold** : $(($$TERM$ *list*$)$ $->$ $(TERM * 'a)$ $->$ $'a)$ $->$ $(TERM * 'a)$ $->$ $'a$;

Description *term_fold tmfun* (*tm*, *e*) traverses *tm* (depth first) and folds *tmfun* on the subterms for which it does not fail. *term_fold* does not traverse a subterm on which *tmfun* did not fail. *tmfun* has as its first argument a list giving the bound variables which are in scope at the point of use. It does not attempt to apply *tmfun* to a bound variable of an abstraction.

SML

val **term_grab** : $(TERM$ *list* $*$ $TERM)$ $->$ $TERM$ *list*;

Description If the given term is not α-convertible to any member of the list, then add it to the list. An infix function.

SML
val **term_less** : (*TERM list* ∗ *TERM*) −> *TERM list*;

Description Remove any terms in the list that are α-convertible to the given term. An infix function.

SML
val **term_map** : ((*TERM list*) −> *TERM* −> *TERM*) −> *TERM* −> *TERM*;

Description *term_map tmfun tm* traverses *tm* (breadth first) looking for subterms for which the application *tmfun tm* does not fail and replaces such subterms with *tmfun tm*. It does not traverse the resulting subterms. *tmfun* has as its first argument a list giving the bound variables which are in scope at the point of use. It does not attempt to apply *tmfun* to a bound variable of an abstraction.

SML
val **term_match** : *TERM* −> *TERM* −> (*TYPE* ∗ *TYPE*) *list* ∗ (*TERM* ∗ *TERM*)*list*;

Description *term_match tm_1 tm_2* attempts to find if *tm_1* is an instance of *tm_2*, up to α-convertibility. If so, then it returns two lists. The first gives the correspondence between types in *tm_1* with type variables in *tm_2*. The second gives the correspondence between (type instantiated) terms in *tm_1* with free variables in *tm_2*. Trivial (i.e. (x, x)) correspondences are not noted.

Errors
3054 ?0 is not a term instance of ?1

SML
val **term_mem** : (*TERM* ∗ *TERM list*) −> *bool*;

Description Is the given term α-convertible to any term in the list? An infix function.

SML
val **term_tycons** : *TERM* −> (*string* ∗ *int*) *list*;

Description Returns the set of type constructors and their arity present in types present within a term (represented as a list).

SML
val **term_types** : *TERM* −> *TYPE list*;

Description Gives a list of all the types of constants, variables or λ-abstraction variables within the term argument.

SML
val **term_tyvars** : *TERM* −> *string list*;

Description Returns the list of type variable names present in types present within a term.

SML
val **term_union** : (*TERM list* ∗ *TERM list*) −> *TERM list*;

Description Take the union of two term lists viewed as sets, removing any α-convertible duplicates. An infix function.

See Also *union* for precise ordering of result.

SML
val **term_vars** : *TERM* −> (*string* ∗ *TYPE*) *list*;

Description This function extracts the subterms of a term which are variables (including abstraction variables), giving destroyed variables in each case.

SML

$\mid$ *val* **type_any** : (*TYPE* −> *bool*) −> *TYPE* −> *bool*;

Description Given a predicate on types, tests to see if any sub-type of some type (or the type itself) satisfies the predicate. The search ceases on the first satisfaction, rather than all the tests being done and the results combined.

SML

$\mid$ *val* **type_fail** : *string* −> *int* −> *TYPE list* −> *'a*;

Description *type_fail area msg tyl* first creates a list of functions from *unit* to *string*, using *string_of_type* (q.v.) providing displays of the list of types. It then calls *fail* with the *area*, *msg* and this list of functions. This allows types to be presented in error messages.

SML

$\mid$ *val* **type_map** : (*TYPE* −> *TYPE*) −> *TYPE* −> *TYPE*;

Description *type_map tyfun ty* traverses *ty* (breadth first) looking for subtypes, *st*, for which the application *tyfun st* does not fail and replaces such subtypes with *tyfun st*. It does not traverse the resulting subtypes.

SML

$\mid$ *val* **type_match1** : (*TYPE* ∗ *TYPE*) *list* −> *TYPE* −> *TYPE* −> (*TYPE* ∗ *TYPE*)*list*;

Description *type_match1* is similar to *type_match*, q.v., but has an additional context parameter representing an instantiation; *type_match1* will fail unless the supplied context can be extended to give the required match. For example, the first line below evaluates true, but the second fails.

$\mid$ *type_match1* [(⌜:'b⌝, ⌜:'b⌝)] ⌜:('a → ℕ) → 'b⌝ ⌜:'a → 'b⌝ = [(⌜:'a → ℕ⌝, ⌜:'a⌝), (⌜:'b⌝, ⌜:'b⌝)];
$\mid$ *type_match1* [(⌜:'b → ℕ⌝, ⌜:'a⌝)] ⌜:('a → ℕ) → 'b⌝ ⌜:'a → 'b⌝;

Trivial associations are included in the result so that they can be passed as the context in subsequent calls. The second element of each pair in the context must be a type variable.

See Also *type_match*

Errors

$\mid$ *3055 ?0 is not a type instance of ?1 in the supplied context*
$\mid$ *3019 ?0 is not a type variable*

SML

$\mid$ *val* **type_match** : *TYPE* −> *TYPE* −> (*TYPE* ∗ *TYPE*)*list*;

Description *type_match ty_1 ty_2* attempts to match *ty_1* with *ty_2*, i.e., to determine if *ty_1* can be obtained from *ty_2* by instantiating type variables. If so, it returns a representation of the type instantation as an association list suitable for use as an argument to *inst_type* q.v. Trivial (i.e. (x, x)) associations are not included. For example:

$\mid$ *type_match* ⌜:('a → ℕ) → 'b⌝ ⌜:'a → 'b⌝ = [(⌜:'a → ℕ⌝, ⌜:'a⌝)];

See Also *type_match1*, *inst_type*

Errors

$\mid$ *3053 ?0 is not a type instance of ?1*

SML

$\mid$ *val* **type_of** : *TERM* −> *TYPE*;

Description This gives the HOL type of a term.

SML
```
val type_tycons : TYPE -> (string * int) list;
```

Description This returns a list of names of type constructors, and the arity of their use, within a type.

SML
```
val type_tyvars : TYPE -> string list;
```

Description Returns the list of type variable names present in a type.

SML
```
val variant : TERM list -> TERM -> TERM;
```

Description *variant stoplist v* returns a variant of variable *v* whose name is not used for any variable in *stoplist* (which must be only variables). The variants are generated by sufficient appending of the variant string (see *set_variant_string*).

Errors
```
3007 ?0 is not a term variable
```

See Also *string_variant*, *list_variant*

SML
```
val var_subst : (TERM * TERM) list -> TERM -> TERM;
```

Description *var_subst alist term* returns the term formed by, for each pair in *alist*, substituting in *term* all free instances of the term variable which is the second of the pair with the first of the pair. The pair of the first matching term variable in the list will be used, duplicates later in the list will be ignored. Renaming may occur to prevent bound variable capture.

Note that the term variables must have the same types as the terms that are to replace them.

Definition
```
var_subst [(⌜t1⌝,⌜x1⌝),(⌜t2⌝,⌜x2⌝),...] ⌜t⌝ =
        ⌜t[t1/x1, t2/x2,...]⌝
```

Errors
```
3007 ?0 is not a term variable
3012 ?0 and ?1 do not have the same types
```

See Also *subst*

SML
```
val ~=$ : (TERM * TERM) -> bool;
```

Description An infix equality test that returns true only when its two term arguments are α-convertible, and false otherwise: no exceptions can be raised. Equality of terms is gained by using =$

SML
```
val ℕ : TYPE;
```

Description This is the HOL type of the natural numbers, $0, 1, \ldots$.

Definition
```
val ℕ = ⌜:ℕ⌝;
```

See Also Theory "ℕ".

5.2 Discrimination Nets

SML
signature **NetTools** = *sig*

Description This provides the discrimination net tools that will be used to maintain and use databases of values indexed by term form.

SML
type 'a **NET**;

Description This is the type of a discrimination net, its type parameter being the type of values that are handled by the net.

SML
val **empty_net** : *'a NET*;

Description This is the starting discrimination net, which returns an empty list of values, regardless of term form.

SML
val **list_net_enter** : (*TERM* ∗ *'a*) *list* −> (*'a NET*) −> (*'a NET*);

Description This enters a list of values and indexing terms into a discrimination net, returning the resulting net.

SML
val **make_net** : (*TERM* ∗ *'a*) *list* −> (*'a NET*);

Description This enters a list of values and indexing terms into an empty discrimination net, returning the resulting net.

SML
val **net_enter** : (*TERM* ∗ *'a*) −> (*'a NET*) −> (*'a NET*);

Description This enters a value and its indexing term into a discrimination net, returning the resulting net.

SML
val **net_lookup** : (*'a NET*) −> *TERM* −> (*'a list*);

Description *net_lookup net term* will return a list of **at least** all the values entered into *net* that were indexed by terms which can be matched (by *term_match*, q.v.) to *term*. I.e. *term* can be produced by type and term variable instantiation from the indexing term.

A principal purpose of *net_lookup* is to make the process of rewriting a term using a list of equations and conversions more efficient by quickly filtering out items which are not applicable. Consequently speed is more important than accuracy: to use the wrong metaphor, it is not important if some inapplicable equations "slip through the net" provided all the applicable ones do as well.

The discrimination net actually returns all values whose indexing terms have the same structure as the term matched, ignoring types and variables. Thus only the pattern of constant names, combinations and abstractions will be considered, with variables in the indexing term being presumed to match any term form, regardless of type.

If *net_lookup* returns more than one value, then the only ordering on the resulting values specified is that if two entries are made into the net with the same index term, then if the *net_lookup* term matches the index term then the second entered value will be returned before the first in the list of matches.

THE MANAGEMENT OF THEORIES AND THEOREMS

6.1 Standard ML Type Definitions

SML
```
datatype THEORY_STATUS =
        TSNormal | TSLocked | TSAncestor | TSDeleted;
```

Description Objects of this datatype indicate the status of a theory within a hierarchy, being:

Constructor	Description
TSNormal	Theory is present and may be written to.
TSLocked	Theory is present, and cannot be written to as it is locked.
TSAncestor	Theory is present, and cannot be written to as it is in an ancestor for some hierarchy.
TSDeleted	Theory has been deleted: the theory name may be reused for a new theory.

SML
```
datatype USER_DATUM =
        UD_Term of TERM * (USER_DATUM list)
|       UD_Type of TYPE * (USER_DATUM list)
|       UD_String of string * (USER_DATUM list)
|       UD_Int of int * (USER_DATUM list);
```

Description This provides a monomorphic type of trees whose nodes are labelled by terms, types, strings or integers.

Uses This type is used in the type *USER_DATA*, and may be used elsewhere, as a means of storing data that may be represented in a "reasonably general" structure for ProofPower related purposes, which also is not polymorphic.

SML
```
type CONV;
```

Description This is the type name conventionally used for conversions, that is, inference rules whose last argument is a term, and whose result is an equation whose LHS is precisely that term (no α-conversion). Though it would be type correct, we conventionally do not use this type name for other functions of type ... −> *TERM* −> *THM*.

Definition
```
type CONV = TERM -> THM;
```

SML
```
type SEQ;
```

Description This is the type of sequents, consisting of a list of assumptions and a conclusion.

Definition
```
type SEQ = (TERM list) * TERM;
```

=#provides a strict equality test on sequents, ∼=#provides an equality test on the sequents up to α-convertibility and order of assumptions.

SML

|*type* **THEORY_INFO**;

Description This is a labelled record type containing certain information associated with a theory.

Label	Type	Description
status	*THEORY_STATUS*	Current status of the theory.
inscope	*bool*	True if the theory is currently in scope (i.e. can its theorems, types and constants be usefully referred to).
contents	*THEORY*	The theory contents.
children	*int list*	List of the immediate children of the theory.
name	*string*	The name of the theory, as a string.

SML

|*type* **THEORY**;

Description A theory is a named collection of type names, constant names, axioms, definitions and theorems. In the abstract data type of theorems, the "names" of theories are represented as integers. For each type name the arity of the type is recorded and for each constant name its type is recorded. In order to allow deletion of types, constants, axioms and definitions. So-called level numbers are used to enables theorems that may depend on deleted material to be identified and rejected. In order for non-critical information such as operator fixity to be stored, a theory also includes a user-data slot which may be used to encode such information.

A theory is represented as a labelled record type, as follows:

Label	Type	Description
name	*int*	Internal representation of theory name.
ty_env	$\{arity : int,\ level : int\}$ *OE_DICT*	A dictionary indexed by type constructor names, returning arity, and definition level.
con_env	$\{ty : TYPE,\ level : int\}$ *OE_DICT*	A dictionary indexed by constant name, returning the type and definition level.
parents	*int list*	Internal representations of names of parents of theory.
del_levels	$(int * int)$ *list*	A list of ranges of deleted definition levels — if empty then no levels have been deleted.
axiom_ dict	*THM OE_DICT*	A dictionary of axioms.
defn_dict	*THM OE_DICT*	A dictionary of definitions.
thm_dict	*THM OE_DICT*	A dictionary of theorems.
current_ level	*int*	The current definition level.
user_data	*USER_DATA ref*	The user data stored in the theory.

SML

|*type* **THM**;

Description This is the abstract data type of theorems in ProofPower, whose primitive constructors are the inference rules and extensional mechanisms of the abstract data type. =|−provides a strict equality test on the conclusion and assumptions of theorems, ~=|−provides an equality test on the conclusion and assumptions of theorems up to α-convertibility and order of assumptions.

SML

```
type USER_DATA;
```

Description This is the type of a store for objects of type *USER_DATUM*. It is implemented as:

ML

```
type USER_DATA = USER_DATUM S_DICT;
```

Uses Within the type *THEORY* it is used to include such details as the fixity of types and constants.

6.2 Symbol Table

SML
```
signature SymbolTable = sig
```

Description This is the signature for the structure which contains the symbol table and its access functions. This structure contains private functions which are invoked as one navigates around the theory database. These private functions may give rise to error 20001 if the theory database user data has been corrupted (e.g. by explicit and incorrect use of the lower level interfaces).

Any of the functions in the structure which update the current theory may give rise to error 20002

Errors
```
20001 A symbol table entry in theory ?0 is corrupt (use restore_defaults to clear)
20002 The current theory, ?0, is not open for writing
20003 Internal error: ?0
```

SML
```
val declare_alias : (string * TERM) -> unit;
```

Description *declare_alias* (s, c) declares s as an alias for the constant c. s must comply with the HOL lexical rules for an identifier.

Errors
```
20301 The term ?0 is not a constant
20302 The string ?0 is already in use as an alias for ?1
20305 The constant ?0 is not in scope
20306 The string ?0 is not an identifier
```

SML
```
val declare_binder : string -> unit;
```

Description *declare_binder* s declares s to have the syntactic status of a binder in the current context. s must comply with the HOL lexical rules for an identifier and must not be the string ",".

See Also *undeclare_fixity*

Errors
```
20201 A fixity declaration is not allowed for ?0 (which is not an identifier)
20202 Cannot change the fixity of ‘,‘
```

SML
```
val declare_const_language : string * string -> unit;
```

Description *declare_const_language* (s, l) adds the language indicator l to those associated with the name s when used as a constant in the current context.

Errors
```
20501 There is no constant called ?0 in the current context
```

SML

```
val declare_left_infix : (int * string)-> unit;
val declare_right_infix : (int * string)-> unit;
val declare_infix : (int * string)-> unit;
```

Description *declare_left_infix* (p, s) declares s to have the syntactic status of an left associative infix operator with precedence p in the current context. s must comply with the HOL lexical rules for an identifier.

Similarly, *declare_right_infix* is used to declare right associative operators. *declare_infix* is provided for compatibility with earlier versions of the system and is the same as *declare_right_infix*.

See Also *undeclare_fixity*

Errors

20201 A fixity declaration is not allowed for ?0 (which is not an identifier)

SML

```
val declare_nonfix : string -> unit;
```

Description *declare_nonfix s* undoes the effect of a declaration of s to have special syntactic status (using *declare_binder*, *declare_infix*, *declare_prefix* or *declare_postfix*).

The effect of *declare_nonfix s* depends on the theory in which the special status for s was declared: if it was declared in the current theory, then the declaration is just removed; if in an ancestor theory then a declaration for s as a nonfix is inserted in the current theory. (Thus in the first case, the syntactic status for s reverts to what it was before the earlier declaration, whereas in the second case the syntactic status will be suppressed.)

s must must not be the string ",".

See Also *undeclare_fixity*

Errors

20201 A fixity declaration is not allowed for ?0 (which is not an identifier)
20202 Cannot change the fixity of ‘,‘
20203 There is no fixity declaration for ?0 in the current context

SML

```
val declare_postfix : (int * string) -> unit;
```

Description *declare_postfix* (p, s) declares s to have the syntactic status of a postfix operator with precedence p in the current context. s must comply with the HOL lexical rules for an identifier and must not be the string ",".

See Also *undeclare_fixity*

Errors

20201 A fixity declaration is not allowed for ?0 (which is not an identifier)
20202 Cannot change the fixity of ‘,‘

SML
```
val declare_prefix : (int * string) -> unit;
```

Description *declare_prefix* (p, s) declares s to have the syntactic status of a prefix operator with precedence p in the current context. s must comply with the HOL lexical rules for an identifier and must not be the string ",".

See Also *undeclare_fixity*

Errors
20201 A fixity declaration is not allowed for ?0 (which is not an identifier)
20202 Cannot change the fixity of ','

SML
```
val declare_terminator : string -> unit
```

Description *declare_terminator* s checks that s is a valid terminator, and if so declares that s is to be used as a lexical terminator in the current context.

Errors
20101 The string ?0 is not a valid terminator. Terminators must start with a symbolic character, must not contain spaces, and must not end with underscore, ⅄ or ϒ
20102 The string ?0 is already declared as a terminator

SML
```
val declare_type_abbrev : (string * string list * TYPE) -> unit;
```

Description *declare_type_abbrev* $(s, [\alpha\_1, \ldots, \alpha\_k], \tau)$ declares $(\alpha\_1, \ldots, \alpha\_k)s$ as a type abbreviation for the type τ. The identifier s may not already have been declared as a type abbreviation or be the name of a type constructor defined in the present context, in which cases a warning message is issued. s must comply with the HOL lexical rules for an identifier.

Errors
20401 The identifier ?0 is already declared as a type abbreviation
20402 The identifier ?0 is already declared as a type constructor
20407 The formal parameter list ?0 contains duplicate type variable names
20408 The string ?0 is not an identifier

SML
```
val expand_type_abbrev : (string * TYPE list) -> TYPE;
```

Description *expand_type_abbrev* $s, [\tau\_1, \ldots, \tau\_k]$ is the expansion of the type abbreviation s with respect to the arguments $[\tau\_1, \ldots, \tau\_k]$.

Errors
20404 The identifier ?0 is not declared as a type abbreviation
20405 The type abbreviation ?0 should have ?1 argument not ?2
20406 The type abbreviation ?0 should have ?1 arguments not ?2

SML
```
val get_aliases : string -> (string * TERM) list;
```

Description *get_aliases thy* returns information about identifiers which have have been declared as aliases in the theory *thy*. The return value is a list of pairs. Each pair contains a name and a constant for which that name is an alias. The same name may be used as an alias for several different constants, and if this happens there will be multiple entries for that alias in the list.

Errors
20601 There is no theory called ?0

SML
```
val get_alias_info : string -> (string * TYPE)list OPT;
```

Description *get_alias_info c* returns the list of aliases for the constant with name *c*, or *Nil* if *c* is not the name of a constant. For each pair (a, τ) in the result, *a* is an alias for *c* at instances of the type τ.

SML
```
val get_alias : (string * TYPE) -> string;
```

Description $get\_alias(c, \tau)$ returns the most appropriate alias for the constant with name *c* at the type τ. If no aliases for the name *c* have been declared then *c* is returned otherwise the most recent alias *s* associated with a type τ' which can be instantiated to τ is returned.

SML
```
val get_binders : string -> string list;
```

Description *get_binders thy* returns the list of identifiers which have been declared as binders in the theory *thy*.

Errors
```
20601 There is no theory called ?0
```

SML
```
val get_const_info : string -> (TYPE * ((string * TYPE)list)) OPT;
```

Description *get_const_info a* returns the information, (τ, cs), associated with the name *a* used as a constant name or an alias for a constant, if any. *cs* is the list of names and types of constants to which *a* might refer (as an alias or as the actual constant name). τ is the type to use for this name during type inference, namely, the antiunifier of the types in *cs*.

SML
```
val get_const_language : string -> string list;
```

Description *get_const_language s* returns the language indicators associated with the name *s* when used as a constant in the current context. If there is no constant called *s*, then *get_const_language s* returns the language indicator associated with the current theory. The language indicator is "*HOL*" for all identifiers supplied as part of the ICL HOL system. The head element of the list returned is the language indicator associated with the constant's declaring theory.

SML
```
val get_current_language : unit -> string;
```

Description *get_current_language* () returns the language indicator associated with the current theory.

SML
```
val get_current_terminators : unit -> string list list;
```

Description *get_current_terminators*() returns the list of identifiers which have been declared as terminators in the current context using *new_terminator*. The names are returned in exploded form, i.e. as a list of strings each containing one character.

SML
```
val get_fixity : string -> Lex.FIXITY;
```

Description *get_fixity s* returns the syntactic status of *s* in the current context.

SML

val **get_language** : *string* −> *string*;

Description *get_language thy* returns the language indicator associated with the theory *thy*.

Errors

20601 There is no theory called ?0

SML

val **get_left_infixes** : *string* −> (*int* ∗ *string*) *list*;
val **get_right_infixes** : *string* −> (*int* ∗ *string*) *list*;

Description *get_left_infixes thy* (resp. *get_right_infixes thy*) returns the list of identifiers (and associated precedences) which have been declared as left (resp. right) associative infix operators in the theory *thy*.

Errors

20601 There is no theory called ?0

SML

val **get_nonfixes** : *string* −> *string list*;

Description *get_nonfixes thy* returns the list of identifiers which are declared as binder, infix, prefix or postfix in an ancestor of the theory *thy*, but have had that special status suppressed (using *declare_nonfix*) in the theory *thy* itself.

Errors

20601 There is no theory called ?0

SML

val **get_postfixes** : *string* −> (*int* ∗ *string*) *list*;

Description *get_postfixes thy* returns the list of identifiers (and associated precedences) which have been declared as postfix operators in the theory *thy*.

Errors

20601 There is no theory called ?0

SML

val **get_prefixes** : *string* −> (*int* ∗ *string*) *list*;

Description *get_prefixes thy* returns the list of identifiers (and associated precedences) which have been declared as prefix operators in the theory *thy*.

Errors

20601 There is no theory called ?0

SML

val **get_terminators** : *string* −> *string list*;

Description *get_terminators thy* returns the list of identifiers which have been declared as terminators in the theory *thy*.

Errors

20601 There is no theory called ?0

SML

val **get_type_abbrev** : *string* −> (*string list* ∗ *TYPE*);

Description *get_type_abbrev s* returns the formal argument list and type associated with the type abbreviation *s*.

Errors

20404 The identifier ?0 is not declared as a type abbreviation

SML

val **get_type_abbrevs** : *string* −> (*string* ∗ (*string list* ∗ *TYPE*))*list*;

Description *get_type_abbrevs thy* returns information about the type abbreviation declarations which have been made in the theory *thy*. The return value is a list of pairs. Each pair contains the name of the corresponding type abbreviation together with its formal arguments and the type for which it is an abbreviation.

Errors

20601 There is no theory called ?0

SML

val **get_type_info** : *string* −> (*int* ∗ (*string list* ∗ *TYPE*) *OPT*) *OPT*;

Description *get_type_info s* returns the type information, if any, associated with *s*. See DS/FMU/IED/DTD020 for more information.

SML

val **get_undeclared_terminators** : *string* −> *string list*;

Description *get_undeclared_terminators thy* returns the list of identifiers whose status as terminators has been suppressed (with *undeclare_terminator*) in the theory *thy*.

Errors

20601 There is no theory called ?0

SML

val **get_undeclared_type_abbrevs** : *string* −> *string list*;

Description *get_undeclared_type_abbrevs thy* returns the list of identifiers which have have had their status as type abbreviations suppressed in the theory *thy*.

Errors

20601 There is no theory called ?0

SML

val **get_undeclared_aliases** : *string* −> (*string* ∗ *TERM*) *list*;

Description *get_undeclared_aliases thy* returns information about aliases which have been suppressed (with *undeclare_alias*) in the theory *thy*. The return value is a list of pairs. Each pair contains a name and a constant for which that name is no longer to be used as an alias. There may be more than one entry for a given name in the list (since several *undeclare_alias* commands may apply to one name).

Errors

20601 There is no theory called ?0

SML

val **is_type_abbrev** : *string* −> *bool*;

Description *is_type_abbrev s* returns *true* iff. *s* is declared as a type abbreviation

SML

val **resolve_alias** : (*string* ∗ *TYPE*) −> *TERM*;

Description $resolve\_alias(s, \tau)$ returns a term of the form $mk\_const(c, \tau)$ where c is the "best' resolution for the identifier s. This best resolution will be s if s has been introduced as a constant of type τ' where τ' is an instance of τ. If s is an alias then c is taken from the alias declaration for s in which the aliased constant has a type τ' which can be instantiated to τ. If more than one such declaration is applicable the most recent one is used.

Errors

20304 The identifier ?0 is not a valid constant name (or alias) at this type

SML
```
val restore_defaults : unit -> unit;
```

Description *restore_defaults*() may be used to clear corrupted symbol table information in the current theory. It does this by restoring the theory to the state it would have if no terminator, fixity, alias, type abbreviations or language declarations had been performed. A warning message is issued (and the interactive user is prompted as to whether to continue) before the operation is performed.

Errors

20703 This operation will delete all symbol table information from theory ?0

SML
```
val set_current_language : string -> unit;
```

Description *set_current_language s* sets the language indicator associated with the current theory to *s*.

SML
```
val undeclare_alias : (string * TERM) -> unit;
```

Description *undeclare_alias* (*s*, *c*) reverses the effect of a declaration of *s* as an alias for the constant *c* in the current context. This includes the possibility that *s* is the name of *c* itself.

The precise effect depends on the theory in which the alias was declared: if it was declared in the current theory, then the declaration is just removed (so that if *s* is declared as an alias for *c* in an ancestor theory, *s* will still act as an alias for *c* in the current theory); if in an ancestor theory then arrangements are made in the current theory to prevent *s* acting as an alias for *c*.

If *s* is the name of *c* itself, the type inferrer will no longer recognise *s* as a reference to *c*. In this case, *c* may be accessed either via an alias or via an ML quotation. This gives a work-around for the potential problem when a theory contains a constant whose name is needed as a variable name in some application using the theory.

Errors

20301 The term ?0 is not a constant
20303 The identifier ?0 is not declared as an alias for ?1

SML
```
val undeclare_terminator : string -> unit
```

Description *undeclare_terminator s* removes *s* from the list of identifiers which act as terminators for parsing purposes in the current context.

Errors

20103 ?0 is not in the list of terminators in the current context

SML
```
val undeclare_type_abbrev : string -> unit;
```

Description *undeclare_type_abbrev* $(s, [\alpha\_1, \ldots, \alpha\_k], \tau)$ reverses the effect of a declaration of *s* as a type abbreviation.

The precise effect depends on the theory in which the type abbreviation was declared: if it was declared in the current theory, then the declaration is just removed (so that if *s* is declared as a type abbreviation in an ancestor theory, *s* will revert to whatever that declaration said); if in an ancestor theory then arrangements are made in the current theory to prevent *s* being treated as a type abbreviation.

Errors

20403 The identifier ?0 is not declared as a type abbreviation

6.3 The Kernel Interface

SML
```
signature KernelInterface = sig
```

Description This is the signature of the structure that gives the standard interface to the logical kernel. This interface adds a layer of additional services to the kernel functionality. E.g., it is used to notify the parser and type-inferrer so that they operate correctly when the current theory changes. The functions in the structure *KernrlInterface* should always be used in preference to direct use of the functions in the structure *pp'Kernel* except in coding extensions to the system that need to bypass these services.

Errors
```
6013 ?0 is ill−formed in current theory: type name ?1 is not declared
6014 ?0 is ill−formed in current theory: type name ?1 does not have arity used
6015 ?0 is ill−formed in current theory: constant name ?1 not declared
6038 ?0 is ill−formed in current theory: constant name ?1 cannot have type used
```

The above are error messages various kinds of well-formedness check failures. A well-formedness check occurs on any types, terms and theorems saved in a theory, and thus these errors may occur for any function in this signature which saves types, terms or theorems in a theory.

SML
```
datatype KERNEL_INFERENCE =
		KISubstRule of (THM * TERM) list * TERM * THM * THM
	|	KISimpleλEqRule of TERM * THM * THM
	|	KIInstTypeRule of (TYPE * TYPE) list * THM * THM
	|	KI⇒Intro of TERM * THM * THM
	|	KI⇒Elim of THM * THM * THM
	|	KIAsmRule of TERM * THM
	|	KIReflConv of TERM * THM
	|	KISimpleβConv of TERM * THM
	|	KISucConv of TERM * THM
	|	KIStringConv of TERM * THM
	|	KICharConv of TERM * THM
	|	KIEqSymRule of THM * THM
	|	KIListSimple∀Elim of TERM list * THM * THM
	|	KIEqTransRule of THM * THM * THM
	|	KIMkAppRule of THM * THM * THM
	|	KI⇔MPRule of THM * THM * THM
	|	KISimple∀Intro of TERM * THM * THM
	|	KIInstTermRule of (TERM * TERM) list * THM * THM
	|	KIPlusConv of TERM * THM;

val on_kernel_inference : (KERNEL_INFERENCE -> unit) -> unit;
```

Description The call *on_kernel_inference f* registers the function *f* to be called whenever a kernel inference rule is called successfully. Several functions may be registered and they will be called in order of registration.

A value of type *KERNEL_INFERENCE* is passed to represent the instance of the rule that has been called. The tuple forming the argument to each constructor of the type gives the arguments and result of the corresponding rule.

SML

```
(* compactification_mask : integer control: default: compiler-dependent *)
val get_compactification_cache : unit -> TYPE list;
val set_compactification_cache : TYPE list -> unit;
val clear_compactification_cache : unit -> unit;
```

Description These functions and associated control value support compactification of objects stored in the theory database.

set_compactification_cache and *get_compactification_cache* may be used at the beginning and end of a ProofPower session to preserve the contents of the cache of type information which is used to implement compactification. Internally, the cache is held as a rather more complex, and much larger, data structure than a simple list of types and so *clear_compactification_cache* is used automatically to empty the cache at the end of a session, thereby avoiding saving the data structure in the database file. Restoring the cache from the list returned by *get_compactification_cache* using *set_compactification_cache* is time-consuming and is not done automatically; however, doing this using, e.g., the following lines of ML, may improve the space-saving in applications which are built up in several sessions:

SML Example - End of Every Session

```
val saved_compactification_cache = get_compactification_cache();
```

SML Example - Beginning of Second and Later Sessions

```
set_compactification_cache saved_compactification_cache;
```

ML functions which compute terms can often be coded so as to produce terms in which common subterms are shared. The compactification algorithm may actually increase the space occupied by such terms. Producers of such functions may therefore wish to suppress the compactification when the computed terms are stored in the theory database.

compactification_mask is an integer control which is treated as a bit-mask and may used to suppress selected aspects of the compactification algorithm. The default value of *0* should be correct for most normal specification and proof work. The significance of the bits in the mask is as follows:

1	Suppress compactification in *new_axiom*
2	Suppress compactification in *new_const*
4	Suppress compactification in *new_type_defn*
8	Suppress compactification in *new_spec*
16	Suppress compactification in *save_thm*
32	Suppress compactification in *simple_new_defn*

So, for example, if the mask is set to *47* (= *1* + *2* + *4* + *8* + *32*), then compactification will only be performed when *save_thm* is called. The default value depends on the Standard ML compiler: 63 (i.e., no compactification) for Poly/ML and 0 (i.e., full compactification) for Standard ML of New Jersey.

SML

```
datatype KERNEL_STATE_CHANGE
        =   OpenTheory of string * ((string list) * (string list))
        |   DeleteTheory of string
        |   NewTheory of string
        |   NewParent of string * (string list)
        |   LockTheory of string
        |   UnlockTheory of string
        |   DuplicateTheory of string * string
        |   SaveThm of string * THM
        |   ListSaveThm of string list * THM
        |   DeleteConst of TERM
        |   DeleteType of string
        |   DeleteAxiom of string
        |   DeleteThm of string
        |   NewAxiom of (string list * TERM)*THM
        |   NewConst of string * TYPE
        |   NewType of string * int
        |   SimpleNewDefn of (string list * string * TERM) * THM
        |   NewTypeDefn of (string list * string * (string list) * THM) * THM
        |   NewSpec of (string list * int * THM) * THM
        |   SetUserDatum of string * USER_DATUM;
```

Description This is an encoding of the arguments of the functions of signature *KernelInterface* which change the state of the theory database. When used to notify the system of a change that has been made certain additional information is also included. If used to notify the system before a change is made the slots will be given "null" default values ("", [], *asm_rule mk_t*).

Operation	Value	Description
open_theory	(*thy*, (*inthys*, *outthys*))	*thy* names the theory which has been opened. *inthys* names the theories which have come into scope. *outthys* names the theories which have gone out of scope.
new_parent	(*thy*, *inthys*)	*thy* names the new parent theory. *inthys* names the theories which have come into scope.
SimpleNewDefn *NewTypeDefn* *NewSpec* *NewAxiom*	(*arg*, *thm*)	*arg* gives the argument to the operation. *thm* is the new defining theorem.

SEE ALSO *on_kernel_state_change*, *before_kernel_state_change*

SML
```
type CHECKPOINT;
val checkpoint : string -> CHECKPOINT;
val rollback : CHECKPOINT -> unit;
```

Description This opaque type and its associated functions implement a system for checkpointing and restoring the state of the theory hierarchy. It is intended primarily for programmatic use in applications that may need to undo multiple extensions to the logical contents of the theory and changes to user data. The check-pointing scheme is unable to keep track of theories, theorems, definitions etc. that have been deleted. Applications that may delete such objects must make their own arrangements for restoring the deleted objects.

The parameter to *checkpoint* is a theory name. The checkpoint returned contains the information required by *rollback* to roll the indicated theory and all its descendants back to the state it had when the checkpoint was taken. The theory becomes the current theory after the rollback.

Rolling back is done using *delete_const* etc. and so rolling back the state of definitions and axioms is restricted to changes made in theories which did not have children when the checkpoint was taken. For uniformity, *rollback* does not attempt to restore the state of the theorems and the user data in theories which had children when the checkpoint was taken. A theory that has been introduced and has become a parent of a theory that existed when the checkpoint was taken will not be deleted (otherwise the child theory would also have to be deleted).

Messages 12015 to 12017 are reported by *rollback* as comments. In general, *rollback* will just report on the problem and continue trying to restore other theories. For example, if *rollback* is unable to delete a theory, it continues to attempt to restore the state of the definitions, etc. in the theories that are to be retained. This is an unlikely situation, since *rollback* unlocks a theory if necessary before trying to delete it, so it will only happen if the application using *rollback* has created a new theory hierarchy and a theory to be deleted has obtained ancestor status. Message 12020 is reported by *rollback* as a failure.

Errors
```
12015 it was not possible to delete theory ?0
12016 the theory ?0 has been deleted since the checkpoint was taken; this change cannot
      be rolled back
12017 a failure was reported while trying to restore theory ?0 (?1)
12020 the theory ?0 has been deleted since this checkpoint was taken and a new
      theory of the same name has been created. Rolling back to this checkpoint
      is not possible.
```

SML
```
val =|- : THM * THM -> bool;
val ~=|- : THM * THM -> bool;
val =# : SEQ * SEQ -> bool;
val ~=# : SEQ * SEQ -> bool;
```

Description =|− provides a strict equality test on the conclusion and assumptions of theorems, ∼=|− provides an equality test on the conclusion and assumptions of theorems up to α-convertibility and order of assumptions. =# provides a strict equality test on sequents, ∼=# provides an equality test on the sequents up to α-convertibility and order of assumptions.

SML
```
val asms : THM -> TERM list;
```

Description This returns the assumptions(hypotheses) of a theorem.

See Also *dest_thm*

SML
```
val before_kernel_state_change : (KERNEL_STATE_CHANGE -> unit) -> unit
```

Description *before_kernel_state_change f* nominates *f* to be called before the theory database is to be modified by functions from the signature *KernelInterface*. The argument to *f* encodes the operation which caused the modification together with its arguments and certain other additional information (usually sets to null defaults for this function). A list of such functions is maintained, and the new function is put at the end of the list, which means it may, if desired undo or overwrite the effects of a function nominated by an earlier call of *before_kernel_state_change*.

Functions handled by *before_kernel_state_change* might be used to raise errors to prevent the state change occurring. This will prevent further checks or actions being made. Thus a careful choice between *before_* or *on_* is called for.

See Also *KERNEL_STATE_CHANGE*, *on_kernel_state_change*

SML
```
val compact_type : TYPE -> TYPE;
val compact_term : TERM -> TERM;
val compact_thm : THM -> THM;
```

Description These functions compactify type, term and theorem values, currently by commoning up type information so that only one ML instance of any type is used in the compactified value. Depending on the value of the integer control variable *compactification_mask*, q.v., these interfaces are invoked automatically as values are stored in the theory database.

The *compactify_$\mathcal{XXX}$* interfaces act as identify functions: *compactify_$\mathcal{XXX}$ x* returns a value which is equal to *x* (in the sense of =:, = $ or = |− as appropriate), but which usually occupies significantly less space than *x*.

SML
```
val concl : THM -> TERM;
```

Description This returns the conclusion of a theorem.

See Also *dest_thm*

SML

```
val delete_axiom : string -> unit
```

Description *delete_axiom key* deletes the axiom stored under key *key* and any other object which depends on it from the current theory. If any objects do depend on the axiom, the interactive user will be notified and asked whether to proceed with the deletion.

After the deletion any theorems which have been proven since the introduction of the axiom will no longer be usable for further proof.

Note that the deletion will attempt to delete all necessary theorems before deleting constants, types, and axioms in single steps, and thus may fail with a partially modified theory. This is because checks in the interface may not be as definitive as those of the kernel. The "on kernel state change" functions will be notified as if all necessary single step deletions, of theorems, constants, types and axioms to achieve the goal had been done, but after all the actual changes have been made. The "before kernel state change" functions will be notified of all the changes, as if single steps, before any are made.

Errors

```
6037   Theory ?0 is locked
6071   Theory ?0 is a read-only ancestor
6076   Theory ?0 has child theories
12003  Theory ?0 does not contain an axiom under key ?1
12012  Deletion of ?0 would require the deletion of ?1
```

SML

```
val delete_const : TERM -> unit
```

Description *delete_const c* deletes the constant *c* (or the constant with the same type, up to renaming of type variables) and any other object which depends on *c* from the current theory. If *c* is the application of a constant to some arguments then that constant is the one deleted. If any saved objects other than *c* and its defining theorem do depend on *c*, the interactive user will be notified and asked whether to proceed with the deletion.

After the deletion any theorems which have been proven since the definition of *c* will no longer be usable for further proof.

Note that the deletion will attempt to delete all necessary theorems before deleting constants, types, and axioms in single steps, and thus may fail with a partially modified theory. This is because checks in the interface may not be as definitive as those of the kernel. The "on kernel state change" functions will be notified as if all necessary single step deletions, of theorems, constants, types and axioms to achieve the goal had been done, but after all the actual changes have been made. The "before kernel state change" functions will be notified of all the changes, as if single steps, before any are made.

Errors

```
6037   Theory ?0 is locked
6071   Theory ?0 is a read-only ancestor
6076   Theory ?0 has child theories
12001  Theory ?0 does not contain the constant ?1 with the supplied type
12012  Deletion of ?0 would require the deletion of ?1
12014  ?0 is not a constant or a constant applied to some arguments
```

SML
```
val delete_theory : string -> unit;
```

Description *delete_theory thy* removes the theory *thy* from the theory database. This means, for instance, that all theorems that were proven with the deleted theory as the current theory, and all constants and types declared within the theory, will become out of scope.

Errors
```
12035  Theory ?0 is not present in the current hierarchy
6037   Theory ?0 is locked
6069   Theory ?0 is in scope
6071   Theory ?0 is a read-only ancestor
6076   Theory ?0 has child theories
```

SML
```
val delete_thm : string -> THM;
```

Description *delete_thm key* deletes the theorem stored under key *key* from the current theory. It returns the deleted theorem.

Errors
```
6037   Theory ?0 is locked
6046   Key ?0 is not used for a theorem in theory ?1
6071   Theory ?0 is a read-only ancestor
```

SML
```
val delete_to_level :
        {do_warn : bool,
         caller : string,
         target : string,
         level : int} -> (string * int) list * (string * TYPE) list;
val thm_level : THM -> int;
```

Description *delete_to_level* deletes constants, types and axioms (and any theorems that may depend on them) down to a specified level number. *do_warn* specifies whether or not the user should be warned before doing this. *caller* is the name of the calling function for use in error messages. *target* is the name of the target being deleted for use in the warning message. *level* is the level of the constant, type or axiom which is the target to be deleted. The returned value comprises the lists of types and constants that have been deleted (with their arities and types).

The level numbers for constants and types may be retrieved using the data structure returned by *get_theory*. *thm_level* returns the level number associated with a theorem or axiom.

SML
```
val delete_type : string -> unit
```

Description *delete_type t* deletes the type constructor *t* and any other object which depends on *t* from the current theory. If any objects other than *t* and its defining theorem do depend on *t*, the interactive user will be notified and asked whether to proceed with the deletion.

After the deletion ny theorems which have been proven since the definition of *ty* will no longer be usable for further proof.

Note that the deletion will attempt to delete all necessary theorems before deleting constants, types, and axioms in single steps, and thus may fail with a partially modified theory. This is because checks in the interface may not be as definitive as those of the kernel. The "on kernel state change" functions will be notified as if all necessary single step deletions, of theorems, constants, types and axioms to achieve the goal had been done, but after all the actual changes have been made. The "before kernel state change" functions will be notified of all the changes, as if single steps, before any are made.

Errors
```
6037  Theory ?0 is locked
6071  Theory ?0 is a read-only ancestor
6076  Theory ?0 has child theories
12002 Theory ?0 does not contain the type constructor ?1
12012 Deletion of ?0 would require the deletion of ?1
```

SML
```
val dest_thm : THM -> SEQ;
```

Description This returns the representation of a theorem as a sequent, i.e. as a list of assumptions and a conclusion.

See Also *asms*, *concl*

SML
```
val do_in_theory : string -> ('a -> 'b) -> 'a -> 'b;
```

Description *do_in_theory thy f a* will change to the named theory *thy*, apply *f* to *a*, and return to the theory in which it was called. It will not notify the kernel state change functions (e.g. *on_kernel_state_change*) when it changes to the named theory, nor will it notify them on its return. Thus for instance the symbol table mechanism, and so term parsing, will behave as if no theory change had taken place before the application of *f* to *a*. This refusal to notify causes this function to be faster than the appropriate two uses of *open_theory*.

The function prevents the application of *f* from once more changing the current theory to another, or functions that may delete the original theory. The block will provoke error 12011. These functions are:

```
open_theory    new_theory    delete_theory
```

It will also discard any changes made by *before_kernel_state_change* during the application of *f* at its end.

The function will intercept any exceptions (including keyboard interrupts), and will attempt to remove the block on changing the current theory, and then return to the original theory. However, in certain circumstances (such as multiple keyboard interrupts, or use of *pp'* functions) the exception handler itself may be interrupted or be otherwise unable to complete its work. In these cases *open_theory* must be used by hand to notify the proof system of the correct theory and its context. If this raises the error *12011* then repeat the use of *open_theory*, as each raising of the error involves the removal of one block put in place by *do_in_theory* before the message is generated.

Errors

12011	*Blocked from changing the current theory. This particular block has now been removed. Exceptionally, further blocks, giving the same error message, may still be in place. These blocks should be cleared now by repeatedly trying open_theory until this error message is not provoked*
12013	*An internal error has corrupted the current theory data. Immediately make a call of open_theory to clear this internal error*
12203	*The kernel interface tables were in an inconsistent state. The tables are now being rebuilt.*

SML
```
val duplicate_theory : (string * string) -> unit;
```

Description *duplicate_theory oldthy newthy* creates a new theory, called *newthy* with the same contents and parents as *oldthy*, but without any children. The current theory remains unchanged.

Uses To allow the user to modify and experiment with a theory that has child theories that are not involved in the experiment, and would perhaps clash with the experiment.

Errors

6026	*Theory ?0 may not be duplicated (it must always be in the scope of any opened theory)*
6042	*Theory ?0 may not be duplicated (the duplicate would not be a descendant of ?1)*
12035	*Theory ?0 is not present in the current hierarchy*
6040	*Theory ?0 is already present in current theory hierarchy*

To ensure that the duplicate theory can be opened by *open_theory* (q.v.) the system will prevent the duplication of theories which would give rise to error 6017 of *open_theory* if opened, and attempts to create such duplicates will give rise to error 6026 or 6042.

SML
```
val get_ancestors : string -> string list;
```

Description This returns all the ancestors of the named theory, including the theory itself. The named theory is the last name in the list returned. The name of the parent first added to the named theory is next to last, preceded by its ancestors. All these are preceded by the second parent theory and its ancestors, apart from those already added. These are preceded by any unnoted ancestors of the third, fourth, etc parents of the named theory. The order in the list of the ancestors of the parent theories is determined recursively by this ordering.

Errors

12035	*Theory ?0 is not present in the current hierarchy*

SML
```
val get_axioms : string -> (string list * THM) list;
val get_axiom_dict : string -> THM OE_DICT;
```

Description *get_axioms* returns all the axioms stored in the indicated theory together with the keys under which they are stored.

get_axiom_dict returns the mapping of keys to axioms represented as an order-preserving efficient dictionary.

Errors

12035	*Theory ?0 is not present in the current hierarchy*

SML
```
val get_axiom : string -> string -> THM;
```

Description *get_axiom theory key* returns the axiom with key *key*, found in theory *theory*.

To improve performance, this function uses a cache containing the values of previous calls. This cache is rebuilt when *open_theory* is called, by removing entries that have gone out of scope. Opening a theory such as *basic_hol* that is low down in the theory hierarchy will reclaim the memory occupied by the cache.

Errors

12035	*Theory ?0 is not present in the current hierarchy*
12005	*Theory ?0 does not have an axiom with key ?1*
12010	*Theory ?0 is not in scope*

SML
```
val get_children : string -> string list;
```

Description This returns the immediate children of the named theory, (not including the theory itself).

Errors
12035 Theory ?0 is not present in the current hierarchy

SML
```
val get_consts : string -> TERM list;
```

Description This returns (most general instances of) all the constants stored in a theory.

Errors
12035 Theory ?0 is not present in the current hierarchy

SML
```
val get_const_keys : string -> E_KEY list;
```

Description This returns the efficient dictionary keys that represent the names of the constants stored in a theory.

Errors
12035 Theory ?0 is not present in the current hierarchy

SML
```
val get_const_theory : string -> string;
```

Description *get_const_theory c* returns the name of the theory in which the constant *c* is defined.

Errors
12201 There is no constant called ?0 in the current context

SML
```
val get_const_type : string -> TYPE OPT;
```

Description If a constant with the given name is in scope, then its type is returned, otherwise *Nil*.

Uses This is likely to be often used just as a rapid test for a constant being in scope.

See Also *get_const_info*

SML
```
val get_current_theory_name : unit -> string;
```

Description Returns the name of the current theory.

SML
```
val get_current_theory_status : unit -> THEORY_STATUS;
```

Description This returns the current theory's status.

SML

```
val get_defns : string -> (string list * THM) list;
val get_defn_dict : string -> THM OE_DICT;
```

Description *get_defns* returns all the defining theorems stored in the indicated theory together with the keys under which they are stored.

get_defn_dict returns the mapping of keys to defining theorems represented as an order-preserving efficient dictionary.

Errors

12035 Theory ?0 is not present in the current hierarchy

SML

```
val get_defn : string -> string -> THM;
```

Description *get_defn theory key* returns the definition with key *key*, found in theory *theory*.

To improve performance, this function uses a cache containing the values of previous calls. This cache is rebuilt when *open_theory* is called, by removing entries that have gone out of scope. Opening a theory such as *basic_hol* that is low down in the theory hierarchy will reclaim the memory occupied by the cache.

Errors

12035 Theory ?0 is not present in the current hierarchy
12004 Theory ?0 does not have a definition with key ?1
12010 Theory ?0 is not in scope

SML

```
val get_descendants : string -> string list;
```

Description This returns all the descendants of the named theory, including itself.

Errors

12035 Theory ?0 is not present in the current hierarchy

SML

```
val get_parents : string -> string list;
```

Description This returns the immediate parents of the named theory, (not including the theory itself).

Errors

12035 Theory ?0 is not present in the current hierarchy

SML

```
val get_theory_names : unit -> string list;
val theory_names : unit -> string list;
```

Description These return the list of undeleted theories in the current hierarchy, whether in scope or not. *theory_names* is an alias for *get_theory_names*.

SML

```
val get_theory_status : string -> THEORY_STATUS;
```

Description This returns the status of the indicated theory.

Errors

12035 Theory ?0 is not present in the current hierarchy

SML
```
val get_theory : string -> THEORY;
val get_theory_info : string -> THEORY_INFO;
```

Description These functions return the data structures associated with a theory in the logical kernel.

Errors
12035 Theory ?0 is not present in the current hierarchy

SML
```
val get_thms : string -> (string list * THM) list;
val get_thm_dict : string -> THM OE_DICT;
```

Description *get_thms* returns all the theorems stored in the indicated theory together with the keys under which they are stored.

get_thm_dict returns the mapping of keys to theorems represented as an order-preserving efficient dictionary.

Errors
12035 Theory ?0 is not present in the current hierarchy

SML
```
val get_thm : string -> string -> THM;
```

Description *get_thm theory key* returns the theorem with key *key*, found in theory *theory*.

To improve performance, this function uses a cache containing the values of previous calls. This cache is rebuilt when *open_theory* is called, by removing entries that have gone out of scope. Opening a theory such as *basic_hol* that is low down in the theory hierarchy will reclaim the memory occupied by the cache.

Errors
12035 Theory ?0 is not present in the current hierarchy
12006 Theory ?0 does not have a theorem with key ?1
12010 Theory ?0 is not in scope

SML
```
val get_types : string -> TYPE list;
```

Description This returns (canonical applications of) all the type constructors stored on a theory.

Errors
12035 Theory ?0 is not present in the current hierarchy

SML
```
val get_type_arity : string -> int OPT;
```

Description If a type with the given name is in scope, then its arity is returned, otherwise *Nil*.

Uses This is likely to be often used just as a rapid test for a type being in scope.

See Also *get_type_info*

SML
```
val get_type_keys : string -> E_KEY list;
```

Description This returns the efficient dictionary keys that represent the names of the type constructors stored in a theory.

Errors
12035 Theory ?0 is not present in the current hierarchy

SML
```
val get_type_theory : string -> string;
```

Description *get_type_theory ty* returns the name of the theory in which the type constructor *ty* is defined.

Errors

12202 There is no type constructor called ?0 in the current context

SML
```
val get_user_datum : string -> string -> USER_DATUM;
```

Description *get_user_datum thy key* returns the value stored in the user data slot allocated to *key* in the theory *thy*, if any.

Errors

12035 Theory ?0 is not present in the current hierarchy
12009 No user data stored under key ?0 in theory ?1

SML
```
val is_theory_ancestor : string -> string -> bool;
```

Description *is_theory_ancestor thy1 thy2* returns true if *thy1* is an ancestor of *thy2* within the current hierarchy.

Errors

12035 Theory ?0 is not present in the current hierarchy

This failure arises if either theory name is not present in the current hierarchy.

SML
```
val kernel_interface_diagnostics : bool -> {
        clean_flag : bool,
        const_thys : int list E_DICT list,
        type_thys: int list E_DICT list,
        int_thy_names : int E_DICT,
        in_scope : int list};
```

Description This function can be used to examine and optionally reset internal state used by the kernel interface module. It is intended for diagnostic purposes. If the argument is *false*, it just returns a representation of the state; if *true*, it also sets the internal state so that the next call on any operation such as *get_const_theory* will cause the state to be recalculated.

SML
```
val list_save_thm : (string list * THM) -> THM
```

Description *list_save_thm(keys, thm)* causes *thm* to be save under the keys *keys* in the current theory. The saved theorem is returned as the function's result. If there is a conjecture stored under any of the keys in the current theory, the theorem must prove each such conjecture, i.e., its conclusion must be the same as the conjecture and it must have an empty assumption list.

See Also *new_conjecture*, *is_proved_conjecture*

Errors

6031 Key list may not be empty
6037 Theory ?0 is locked
6039 Key ?0 has already been used for a theorem in theory ?1
6071 Theory ?0 is a read−only ancestor
103101 This theorem does not prove the conjecture stored under key ?0

SML
```
val lock_theory : string -> unit;
```

Description *lock_theory thy* causes *thy* to be *locked*. The contents of a locked theory are protected from further changes. A locked theory may be unlocked using *unlock_theory*(q.v.).

Errors

12035 Theory ?0 is not present in the current hierarchy
6037 Theory ?0 is locked
6071 Theory ?0 is a read−only ancestor

SML
```
val new_axiom : (string list * TERM) -> THM
```

Description *new_axiom*(*keys*, *tm*) stores the boolean term *tm* an axiom in the current theory as an axiom under keys *keys*.

Errors

3031 ?0 is not of type ⌜:BOOL⌝
6031 Key list may not be empty
6037 Theory ?0 is locked
6047 Key ?0 has already been used for an axiom in theory ?1
6071 Theory ?0 is a read−only ancestor

SML
```
val new_const : (string * TYPE) -> TERM;
```

Description *new_const* (*name*, *type*) introduces a new constant (with no defining theorem) called *name*, with most general type *type*, into the current theory.

Errors

6037 Theory ?0 is locked
6049 There is a constant called ?0 already in scope
6063 There is a constant called ?0 in the descendants of the current theory
6071 Theory ?0 is a read−only ancestor

SML
```
val new_parent : string -> unit;
```

Description Adds the given parent theory to the list of parents of the current theory, considered as a set. It will fail if the parent theory does not exist; is already a parent of the current theory; or if making it a parent would cause a clash by bringing a new theory into scope (perhaps the new parent itself) that declares a new type or constant that is already in scope, or is declared in the descendants of the current theory.

Errors

12035 Theory ?0 is not present in the current hierarchy
6037 Theory ?0 is locked
6067 Making ?0 a parent would cause a clash
6071 Theory ?0 is a read−only ancestor
6082 Theory ?0 is already a parent
6084 Suggested parent ?0 is a child of the current theory

SML

val **new_spec** : (*string list* ∗ *int* ∗ *THM*) −> *THM*;

Description *new_spec* (*keylist*, *ndef*, '$\vdash\exists x\_1, \ldots, x\_n \bullet p[x\_1, \ldots, x\_n]$') will introduce *ndef* new constants named and typed from the x_i. It will also save a defining theorem under each of the keys in *keylist* in the current theory of the form '$\vdash p[c\_1, \ldots, c\_n]$' where c_i is the constant with the name and type of x_i. If either the constant or theorem introduction fails then the function will not change the current theory.

Errors

6016	*Existentially bound variable ?0 is repeated in theorem ?1*
6031	*Key list may not be empty*
6037	*Theory ?0 is locked*
6044	*Must define at least one constant*
6049	*There is a constant called ?0 already in scope*
6051	*Key ?0 has already been used for a definition in theory ?1*
6053	*?0 must not have assumptions*
6056	*?0 is a free variable in ?1*
6062	*?0 are free variables in ?1*
6060	*?0 is not of the form: '⊢ ∃ x1 ... xn • p[x1,...,xn]' where the ⌜xi⌝ are variables, and n(= ?1) is the number of constants to be defined*
6061	*the body of ?0 contains type variables not found in type of constants to be defined, the variables being: ?1*
6063	*There is a constant called ?0 in the descendants of the current theory*
6071	*Theory ?0 is a read−only ancestor*
6081	*Sets of type variables in ?0 and ?1 differ*

SML

val **new_theory** : *string* −> *unit*;

Description *new_theory thy* adds a new, empty, theory called *thy* to the theory database. The empty theory has no declarations within it, but does have the current theory as its sole parent. The new theory then becomes the current theory.

Errors

6040	*Theory ?0 is already present in current theory hierarchy*

SML

```
val new_type_defn :
        (string list * string * string list * THM) -> THM;
```

Description *new_type_defn* (*keys*, *name*, *typars*, *defthm*) declares a new type with name *name*, and arity the length of *typars*. It creates a defining theorem for the type, saves it in the current theory under the keys *keys*. It returns the defining theorem. *defthm* must be a valid well-formed theorem of the form:

$\vdash \exists\ x : type \bullet p\ x$

with no assumptions. The defining theorem will then be of the form:

$\vdash \exists\ f : typars\ name \rightarrow type \bullet$
$\quad TypeDefn\ (p: type \rightarrow BOOL)\ f$

where *TypeDefn* asserts that its predicate argument *p* is non-empty, and its function argument *f* is a bijection between the new type and the subset of *type* delineated by *p*.

Errors

6031	*Key list may not be empty*
6034	*There is a type called ?0 in the descendants of the current theory*
6037	*Theory ?0 is locked*
6045	*There is a type called ?0 already in scope*
6052	*Key ?0 has already been used for an type definition theorem in theory ?1*
6053	*?0 must not have assumptions*
6054	*?0 is not of the form: '⊢ ∃ x • px'*
6055	*?0 is not of the form: '⊢ ∃ x • p y' where ⌜x⌝ is a variable*
6056	*?0 is a free variable in ?1*
6062	*?0 are free variables in ?1*
6057	*?0 contains type variables not found in type variable parameter list, type variables being: ?1*
6071	*Theory ?0 is a read−only ancestor*
6079	*?0 repeated in type parameter list*
6080	*?0 is not of the form: '⊢ ∃ x • p y' where ⌜x⌝ equals ⌜y⌝*

SML

```
val new_type : (string * int) -> TYPE;
```

Description *new_type* (*name*, *arity*) introduces a new type constructor (with no defining theorem) called *name* with arity *arity* into the current theory. The function returns the new type with sufficient arguments $'1,'2,\ldots$ to provide a well-formed type.

Errors

6034	*There is a type called ?0 in the descendants of the current theory*
6037	*Theory ?0 is locked*
6045	*There is a type called ?0 already in scope*
6071	*Theory ?0 is a read−only ancestor*
6088	*The arity of a type must be ≥ 0*

SML
```
val on_kernel_state_change : (KERNEL_STATE_CHANGE -> unit) -> unit
```

Description *on_kernel_state_change f* nominates *f* to be called whenever the theory database is modified by a function from the signature *KernelInterface*. The argument to *f* encodes the operation which caused the modification together with its arguments and certain other additional information. A list of such functions is maintained, and the new function is put at the end of the list, which means it may, if desired undo or overwrite the effects of a function nominated by an earlier call of *on_kernel_state_change*.

Functions handled by *on_kernel_state_change* should not be coded to raise errors that are not handled by themselves, as the handler will not catch such errors either. If the function is to prevent a change from happening *before_kernel_state_change* should be used instead.

See Also *KERNEL_STATE_CHANGE*, *before_kernel_state_change*

SML
```
val open_theory : string -> unit;
```

Description All specification and proof work is carried out in the context of some theory, referred to as the current theory. *open_theory thy* makes an existing theory *thy* the current theory.

Errors
```
6017   Theory ?0 may not be opened (it is not a descendant of ?1 which must be in scope)
12035  Theory ?0 is not present in the current hierarchy
```

Certain theories created when the system is constructed may not be subsequently opened, and attempts to open them give rise to error 6017.

SML
```
val pending_reset_kernel_interface : unit -> unit -> unit;
```

Description This function, applied to () takes a “snapshot” of the current state of the kernel interface module (comprising the“On Kernel State Change”, “Before Kernel State Change” and “On Kernel Inference” functions). The resulting snapshot, when applied to () will restore these functions to their state at the time of making the snap shot.

Uses To assist in saving the overall system state.

SML
```
val save_thm : (string * THM) -> THM
```

Description *save_thm(key, thm)* causes *thm* to be save under the key *key* in the current theory. The saved theorem is returned as the function’s result. If there is a conjecture stored under the same key in the current theory, the theorem must prove the conjecture, i.e., its conclusion must be the same as the conjecture and it must have an empty assumption list.

See Also *new_conjecture*, *is_proved_conjecture*

Errors
```
6037   Theory ?0 is locked
6039   Key ?0 has already been used for a theorem in theory ?1
6071   Theory ?0 is a read-only ancestor
103101 This theorem does not prove the conjecture stored under key ?0
```

SML

```
val set_user_datum : (string * USER_DATUM) -> unit;
```

Description *set_user_datum(key, ud)* assigns the new value *ud* to the user data slot allocated to *key* in the current theory. If an old value was present it will be overwritten.

Errors

6037	Theory ?0 is locked
6071	Theory ?0 is a read−only ancestor

SML

```
val simple_new_defn : (string list * string * TERM) -> THM;
```

Description *simple_new_defn (keys, name, value)* declares a new constant with name *name*, and with most general type being the type of *value* in the current theory. It creates an equational theorem (i.e. of the form '⊢*name = value*'), and saves it as a definition under keys *keys* in the current theory, provided the theorem is well-formed. If either the constant or theorem introduction fails then the function does not change the current theory. The body of *value* may not contain type variables that are not in the type of *value* itself.

Errors

6031	Key list may not be empty
6037	Theory ?0 is locked
6049	There is a constant called ?0 already in scope
6051	Key ?0 has already been used for a definition in theory ?1
6058	the body of ?0 contains type variables not found in type of term itself, the variables being: ?1
6059	?0 contains the following free variables: ?1
6063	There is a constant called ?0 in the descendants of the current theory
6071	Theory ?0 is a read−only ancestor

SML

```
val string_of_thm : THM -> string;
```

Description This returns a display of a theorem in the form of a string, with no inserted new lines, suitable for use with *diag_string* and *fail*.

See Also *format_thm*, a formatted string display of a theorem.

SML

```
val thm_fail : string -> int -> THM list -> 'a;
```

Description *thm_fail area msg thml* first creates a list of functions from *unit* to *string*, providing displays of the list of theorems. It then calls *fail* with the *area*, *msg* and this list of functions. This allows theorems to be presented in error messages.

SML

```
val thm_theory : THM -> string;
```

Description *thm_theory thm* returns the name of the theory which was current when *thm* was proven. This will succeed even if the theory is out of scope, but not if the theory has been deleted.

Errors

12007	?0 proven in theory with internal name ?1, which is not present in current hierarchy

SML

```
val unlock_theory : string -> unit;
```

Description *unlock_theory thy* causes the locked theory *thy* to be unlocked, so that the contents of *thy* may be changed.

Errors

```
12035   Theory ?0 is not present in the current hierarchy
6068    Theory ?0 has not been locked
```

SML

```
val valid_thm : THM -> bool;
```

Description This function uses the check for the validity of theorems: returning true if valid and false otherwise: it cannot raise exceptions.

Uses To preempt errors caused by the primitive inference rules, which raise uncatchable errors when given invalid theorems, and so return more helpful error messages.

6.4 Conjectures Database

SML

```
val is_proved_conjecture: string -> string -> bool;
val get_proved_conjectures: string -> string list;
val get_unproved_conjectures: string -> string list;
```

Description *is_proved_conjecture thy key* returns true if the conjecture with key *key* in theory *thy* has been proved (i.e., there is a theorem stored under the same key in the theory which has the conjecture as its conclusion and has no assumptions).

get_proved_conjectures thy (resp. *get_unproved_conjectures thy*) returns the list of conjectures in theory *thy* which have (resp. have not) been proved in the sense described above.

See Also *save_thm*, *list_save_thm*, *new_conjecture*

Errors

```
20601  There is no theory called ?0
103101 This theorem does not prove the conjecture stored under key ?0
103102 The theorem with key ?0 does not prove this conjecture
103103 Theory ?0 is not in scope
103802 There is no conjecture called ?0 in theory ?1
103803 The conjectures database in theory ?0 is corrupt
       (use delete_all_conjectures to clear).
```

SML

```
val new_conjecture : (string list * TERM) -> unit;
val get_conjecture: string -> string -> TERM;
val get_conjectures: string -> (string list * (int * TERM)) list;
val delete_conjecture: string -> TERM;
val delete_all_conjectures: unit -> unit;
```

Description *new_conjecture*(*keys*, *tm*) stores the boolean term *tm* as a conjecture in the current theory under keys *keys*. If any of the keys is also the key of a theorem saved in the current theory, then each such theorem must prove the conjecture, i.e., its conclusion must be the same as *tm* and it must have an empty assumption list.

delete_conjecture key deletes the conjecture stored in the current theory under key *key*. It returns the deleted conjecture.

delete_all_conjectures() deletes all the conjectures stored in the current theory. This may be used if, for some reason, the data structure used to store the conjectures becomes corrupted.

Note, when a constants or a type is deleted from a theory, conjectures that contain the deleted constant or type are automatically deleted from the current theory. Message 103804 is used as a comment to inform the user when this happens.

See Also *save_thm*, *list_save_thm*, *is_proved_conjecture*

Errors

3031	*?0 is not of type ⌜:BOOL⌝*
6031	*Key list may not be empty*
20601	*There is no theory called ?0*
103101	*The theorem ?0 does not prove the conjecture with key ?1*
103801	*Key ?0 has already been used for a conjecture in the current theory*
103802	*There is no conjecture called ?0 in theory ?1*
103803	*The conjectures database in theory ?0 is corrupt (use delete_all_conjectures to clear).*
103804	*Deletion of ?0 has caused deletion of conjecture?1: ?2*

6.5 Theorem Finder

SML
```
datatype 'a TEST =
        TFun of 'a -> bool
|       TAll of 'a TEST list
|       TAny of 'a TEST list
|       TNone of 'a TEST list;

type THM_INFO_TEST = THM_INFO TEST;
```

Description The type *THM_INFO_TEST* is used for the parameters of general theorem finder functions, *gen_find_thm* and *gen_find_thm_in_theories* that represent search criteria. The constructor *TFun* is used to represent a basic criterion. *TAll*, *TAny* and *TNone* construct new criteria from old by conjunction, disjunction and negated disjunction respectively.

See Also *any_substring_tt* etc. (for ways of constructing basic criteria).

SML
```
datatype THM_TYPE = TTAxiom | TTDefn | TTSaved;

type THM_INFO = {
        theory      : string,
        names       : string list,
        thm_type    : THM_TYPE,
        thm         : THM};
```

Description The types *THM_TYPE* and *THM_INFO* are used by the theorem finder functions, *find_thm* etc., to represent information about a theorem stored in a theory. The representation gives: the name of the theory; the name or names under which the theorem is stored; an indicator of whether the theorem is an axiom, a definition or a theorem that has been proved and saved; and the actual theorem.

SML
```
val find_thm : TERM list -> THM_INFO list;
```

Description This is a simple interface for finding theorems. *find_thm pats* searches for any theorems in the current theory and its ancestors that contains subterms matching each of the pattern terms *tms*.

The return value is a list of records containing the conclusion of the theorem and other useful information, see the description of the type *THM_INFO* for more details.

For example, if the theory ℝ of real numbers is in scope, the following will find all theorems containing both real number addition and real number multiplication.

```
find_thm [⌜x + y : ℝ⌝, ⌜x * y : ℝ⌝];
```

See Also *gen_find_thm*

SML

```
val gen_find_thm_in_theories : THM_INFO_TEST -> string list -> THM_INFO list;
val gen_find_thm : THM_INFO_TEST -> THM_INFO list;

val any_substring_tt : string list -> THM_INFO TEST;
val all_substring_tt : string list -> THM_INFO TEST;
val no_substring_tt : string list -> THM_INFO TEST;
val any_subterm_tt : TERM list -> THM_INFO TEST;
val all_subterm_tt : TERM list -> THM_INFO TEST;
val no_subterm_tt : TERM list -> THM_INFO TEST;
val any_submatch_tt : TERM list -> THM_INFO TEST;
val all_submatch_tt : TERM list -> THM_INFO TEST
val no_submatch_tt : TERM list -> THM_INFO TEST
```

Description *gen_find_thm_in_theories* is the general theorem finder function. Its first parameter specifies the search criteria and its second parameter specifies the names of the theories to be searched. It returns a list representing the theorems satisfying the criteria. See the definitions of the parameter and return data types for more details.

gen_find_thm calls *gen_find_thm_in_theories* with the specified search criteria and the list of all ancestors of the current theory as the list of theories to search (this include the current theory). Thus it finds all the theorems that are currently in scope that match the specified criteria.

The remaining functions give convenient ways of specifying typical search criteria. These functions support three kinds of basic criterion: substring search criteria test for a specified string appearing as a substring of the name of the theorem; subterm search criteria test for the presence (up to α-equivalence) in the conclusion of the theorem of a specified subterm; submatch search criteria test for the presence in the conclusition of the theorem of a subterm that is an instance of a specified pattern term. Given a list of strings or terms giving basic criteria, the functions test for theorems satisfying all of the criteria (*all_...*), at least one of the criteria (*any_...*) or none of the criteria (*no_...*). The constructors of the data type *THM_INFO_TEST*, q.v., allow more complex logical combinations of criteria to be built up from these.

For example, the following will find all theorems in scope that have names containing "plus" or "minus" as a substring and that have a conclusion that does not contain any natural number additions.

```
gen_find_thm(TAll[any_substring_tt["plus", "minus"], no_subterm_tt[⌜$+:ℕ → ℕ → ℕ⌝]]);
```

See Also *find_thm*

PROOF IN HOL

7.1 General Inference Rules

SML
|*signature* **DerivedRules1** = *sig*

Description This provides the derived rules of inference in Release 001 of ICL HOL. Though other rules of inference may be introduced, this document's signature should provide a core set, at least covering the common rules of natural deduction. It subsumes the inference rules of the abstract data type *THM*.

SML
|*signature* **DerivedRules2** = *sig*

Description This provides the further derived rules of inference for ICL HOL. They are primarily concerned with handling paired abstractions.

SML
|*signature* **Rewriting** = *sig*

Description This provides the derived rewriting rule, conversions and tactics for ICL HOL.

SML
|(* **"illformed_rewrite_warning"** *)

Description This flag modifies the behaviour of *REWRITE_MAP_C* and *ONCE_MAP_WARN_C*. When false (its default) it will not warn of illformed rewriting in subterms, with message 26002, though if no other rewriting occurs then error message 26003 will still be used. If true, then the warning will be given if some rewriting is successful, but elsewhere it is illformed.

SML
|*type* **CANON** (* = *THM* −> (*THM list*) *);

Description This is the type abbreviation for a canonicalisation function; such functions are typically used to derive consequences of a theorem meeting some desired criteria. An example is the rewriting canonicalisations which are used to transform theorems into lists of equational theorems for use in the rewriting conversions, rules and tactics.

Combinators are available to assist in the construction of new canonicalisation functions from old.

See Also *THEN_CAN*, *ORELSE_CAN*, *REPEAT_CAN*, *FIRST_CAN*, *EVERY_CAN* as combinators, *fail_can* and *id_can* as building blocks for the combinators.

SML
|*val* **ALL_SIMPLE_∀_C** : *CONV* −> *CONV*;

Description This conversional applies its conversion argument to the body of a repeated simple universal quantification.

Errors As the failure of the conversion argument.

SML

val **all_simple_∀_elim** : THM −> THM;

Description Specialises all the simple universally quantified variables in a theorem:

Rule

$$\frac{\Gamma \vdash \forall\ x1\ ...\ xn \bullet t[x1,\ ...,\ xn]}{\Gamma \vdash t[x1',\ ...,\ xn']} \quad all\_simple\_\forall\_elim$$

where $x1', \ldots, xn'$ are renamed from $x1, \ldots, xn$ as necessary to avoid clashes with free variables in the assumption list, or duplicated names in the list of specialisations.

SML

val **ALL_SIMPLE_∃_C** : $CONV$ −> $CONV$;

Description This conversional applies its conversion argument to the body of a repeated simple existential quantification.

Errors As the failure of the conversion argument.

SML

val **all_simple_β_conv** : $CONV$;

Description A conversion to eliminate all instances of simple β redexes in a term, regardless of nesting, or even that the β redex was created as the result of an earlier reduction in the conversion's evaluation.

Rule

$$\frac{}{\vdash t = t'} \quad \begin{array}{l} all\_simple\_\beta\_conv \\ \ulcorner t \urcorner \end{array}$$

t' is t with all simple β redexes reduced.

Uses This uses an optimised term traversal algorithm, superior in speed to the general term traversal algorithms used with conversions, and should be used in preference to them and β_conv.

Errors

7020 ?0 contains no β−redexes

SML

val **all_simple_β_rule** : THM −> THM;

Description Eliminate all instances of simple β redexes in a theorem, regardless of nesting, or even that the β redex was created as the result of an earlier reduction in the rule's evaluation.

Rule

$$\frac{\Gamma \vdash t}{\Gamma \vdash t'} \quad all\_simple\_\beta\_rule$$

t' is t with all β-redexes reduced.

Errors

7020 ?0 contains no β−redexes

SML

val **ALL_∧_C** : $CONV$ −> $CONV$;
val **ALL_∨_C** : $CONV$ −> $CONV$;

Description These respectively apply their conversion argument to:

- All the conjuncts of a structure of conjuncts (including a term that is not a conjunct at all) failing only if the conversion fails for all the conjuncts.
- All the disjuncts of a structure of disjuncts (including a term that is not a disjunct at all) failing only if the conversion fails for all the disjuncts.

The result is simplified at any conjunct or disjunct where at least one branch had a successful application of the conversion and matches the appropriate theorems of:

$\vdash \forall\ t\bullet\ (T \land t \Leftrightarrow t) \land (t \land T \Leftrightarrow t) \land \neg\ (F \land t) \land \neg\ (t \land F) \land (t \land t \Leftrightarrow t)$

$\vdash \forall\ t\bullet\ (T \lor t) \land (t \lor T) \land (F \lor t \Leftrightarrow t) \land (t \lor F \Leftrightarrow t) \land (t \lor t \Leftrightarrow t)$

Errors As the failure of the conversion argument.

SML

val **all_⇒_intro** :THM −> THM;

Description Discharge all members of assumption list using *⇒_intro*.

Rule

$$\frac{\{t1,\ ...,\ tn\} \vdash t}{\vdash t1 \Rightarrow ... \Rightarrow tn \Rightarrow t} \quad all\_\Rightarrow\_intro$$

SML

val **all_∀_arb_elim** : THM −> THM;

Description Specialise all the quantifiers of a possibly universally quantified theorem with a machine generated variables or variable structures.

Rule

$$\frac{\Gamma \vdash \forall\ vs1[x1,y1,...]\ vs2[x2,y2,...]\ ...\bullet\ p[x1,y1,....x2,y2,...]}{\Gamma \vdash p[x1',y1',...,x2',y2',...]} \quad \forall\_arb\_elim$$

where x_i', y_i', etc, are not variables (free or bound) in p or Γ, created by *gen_vars*(q.v).

See Also *all_∀_elim*

SML

val **all_∀_elim** : THM −> THM;

Description Specialises all the outer universal quantifications in a theorem:

Rule

$$\frac{\Gamma \vdash \forall\ x1\ ...\ xn\bullet\ t[x1,\ ...,\ xn]}{\Gamma \vdash t[x1',\ ...,\ xn']} \quad all\_\forall\_elim$$

where $x1', \ldots, xn'$ are renamed from $x1, \ldots, xn$ as necessary to avoid name clashes with free variables in the assumption list.

See Also *all_∀_arb_elim* which is faster, though the results are slightly opaque. *list_∀_elim*.

SML

val **all_∀_intro** : *THM* −> *THM*;

Description Generalises all the free variables (other than those in the assumption list) in a theorem:

Rule

$$\frac{\Gamma \vdash t}{\Gamma \vdash \forall\ x1\ \ldots\ xn \bullet t} \quad all\_\forall\_intro$$

where $x1, \ldots, xn$ are all the free variables of t. The function introduces variables in their order of occurrence, so:

Example

all_∀_intro (⊢ *a* ∨ *b*) = ⊢ ∀ *a b* • *a* ∨ *b*

SML

val **all_∀_uncurry_conv** : *CONV*;

Description Apply *∀_uncurry_conv* (q.v) to the outer universal quantifications of a term, flattening those binders.

Conversion

$$\frac{}{\Gamma \vdash (\forall\ vs1[x,y,...]\ vs2[x,y,...]\ ...\bullet f[x1,y1,...,x2,y2,...]) = (\forall\ x1\ y1\ ...\ x2\ y2\ ...\bullet f[x1,y1,...,x2,y2,...])} \quad all\_\forall\_uncurry\_conv\ \ulcorner\forall\ vs1[x1,y1,...]\ vs2[x2,y2,...]\ ...\bullet f[x1,y1,...,x2,y2,...]\urcorner$$

where the $vs\_i[x\_i, y\_i, ...]$ are variable structures at least one of which must not be a simple variable, built from variables $x\_i, y\_i, ...,$

Errors

27041 ?0 is not of the form: ⌜∀ ... (*x*,*y*) ...• *f*⌝

SML

val **all_∃_uncurry_conv** : *CONV*;

Description Apply *∃_uncurry_conv* (q.v) to the outer existential quantifications of a term, flattening those binders.

Conversion

$$\frac{}{\Gamma \vdash (\exists\ vs1[x,y,...]\ vs2[x,y,...]\bullet f[x1,y1,...,x2,y2,...]) = (\exists\ x1\ y1\ ...\ x2\ y2\ ...\bullet f[x1,y1,...,x2,y2,...])} \quad all\_\exists\_uncurry\_conv\ \ulcorner\exists\ vs1[x1,y1,...]\ vs2[x2,y2,...]\ ...\bullet f[x1,y1,...,x2,y2,...]\urcorner$$

where the $vs[x, y, ...]$ are variable structures with variables $x, y, ...,$ at least one of which must not be a simple variable.

See Also *all_∀_uncurry_conv*

Errors

27048 ?0 is not of the form: ⌜∃ ... (*x*,*y*) ...• *f*⌝

SML

```
val all_β_conv : CONV;
```

Description A conversion to eliminate all instances of β redexes, including paired abstraction redexes, in a term, regardless of nesting, or even that the β redex was created as the result of an earlier reduction in the conversion's evaluation.

Rule

$$\frac{}{\vdash t = t'} \quad all\_\beta\_conv \ \ulcorner t \urcorner$$

t' is t with all β redexes reduced.

Uses This uses an optimised term traversal algorithm, superior in speed to the general term traversal algorithms used with conversions, and should be used in preference to them and *β_conv*.

See Also *all_simple_β_conv* which only handles simple β-redexes, but does a faster traversal if that is all that is required. *all_β_rule*.

Errors

```
27049 ?0 contains no β−redexes
```

SML

```
val all_β_rule : THM −> THM;
```

Description Eliminate all instances of β redexes, including paired abstraction redexes, in the conclusion of a theorem, regardless of nesting, or even that the β redex was created as the result of an earlier reduction in the rule's evaluation.

Rule

$$\frac{\Gamma \vdash t}{\Gamma \vdash t'} \quad all\_\beta\_rule$$

t' is t with all β-redexes reduced.

See Also *all_β_conv* for the conversion. *all_simple_β_rule* which only handles simple β-redexes, but does a faster traversal if that is all that is required.

Errors

```
27049 ?0 contains no β−redexes
```

SML

```
val AND_OR_C : (CONV * CONV) −> CONV;
```

Description *c1 AND_OR_C c2* will succeed if it can apply one or both of *c1* or *c2*. If it cannot compose the results of applying both conversions successfully (indicating an ill-formed conversion result) it will return the result of the first conversion application.

See Also *THEN_TRY_C*, *ORELSE_C*, *THEN_C*

Errors As the failure message of the second conversion (implying that neither conversion was successfully applied).

SML

```
val app_arg_rule : TERM -> THM -> THM;
```

Description Apply both sides of an equational theorem to an argument.

Rule

$$\frac{\Gamma \vdash f = g}{\Gamma \vdash f\ x = g\ x} \quad \begin{array}{l} app\_arg\_rule \\ \ulcorner x \urcorner \end{array}$$

Errors

6020	*?0 is not of the form: ‘Γ ⊢ t1 = t2‘*
7025	*Sides of equation may not be applied to term*

SML

```
val APP_C : (CONV * CONV) -> CONV;
```

Description Apply one conversion to the operator of a combination, and a second to the operand.

Rule

$$\frac{}{\vdash f\ a = f'\ a'} \quad \begin{array}{l} APP\_C \\ (c1 : CONV, \\ c2 : CONV) \\ \ulcorner f\ a \urcorner \end{array}$$

where *c1 f* gives ‘$\vdash f = f'$‘, and *c2 f* gives ‘$\vdash a = a'$‘.

Errors

3010	*?0 is not of form: ⌜t1 t2⌝*
7110	*Results of conversions, ?0 and ?1, ill−formed or cannot be combined*

Also as the failure of the conversions.

SML

```
val app_fun_rule : TERM -> THM -> THM ;
```

Description Apply a function to both sides of an equational theorem.

Rule

$$\frac{\Gamma \vdash a = b}{\Gamma \vdash f\ a = f\ b} \quad \begin{array}{l} app\_fun\_rule \\ \ulcorner f \urcorner \end{array}$$

Errors

6020	*?0 is not of the form: ‘Γ ⊢ t1 = t2‘*
7024	*?0 may not be applied to each side of equation*

SML

```
val app_if_conv : CONV;
```

Description Move a function application into a conditional.

Conversion

$$\frac{}{\begin{array}{l}\vdash f(if\ a\ then\ b\ else\ c) = \\ \quad (if\ a\ then\ f\ b\ else\ f\ c)\end{array}} \quad \begin{array}{l} app\_if\_conv \\ \ulcorner f(if\ a\ then\ b\ else\ c)\urcorner \end{array}$$

Errors

7098	*?0 is not of the form: ⌜f(if a then b else c)⌝*

SML
```
val asm_elim : TERM -> THM -> THM -> THM;
```

Description Eliminate an assumption with reference to contradictory assumption lists.

Rule

$$\frac{\Gamma 1,\ a' \vdash t;\ \Gamma 2,\ \neg a''\ \vdash t'}{\Gamma 1 \cup \Gamma 2 \vdash t} \qquad \begin{array}{l} asm\_elim \\ \ulcorner a \urcorner \end{array}$$

where a, a' and a'', as well as t and t' are α-convertible. Actually, the assumptions don't have to be present for the function to succeed.

Errors

3031 ?0 is not of type ⌜:BOOL⌝
7029 ?0 and ?1 are not of the form:'Γ1, aa ⊢ t' and 'Γ2, ¬aaa ⊢ ta'
where ⌜t⌝ and ⌜ta⌝ are α−convertible

SML
```
val asm_inst_term_rule : (TERM * TERM) list -> THM -> THM;
```

Description Parallel instantiation of term variables within a theorem's conclusion and assumptions to some other values.

Rule

$$\frac{\Gamma \vdash t[x1,\ ...,\ xn]}{\Gamma' \vdash t[t1,\ ...,\ tn]} \qquad \begin{array}{l} asm\_inst\_term\_rule \\ [...,\ (\ulcorner ti \urcorner,\ \ulcorner xi \urcorner),\ ...] \end{array}$$

See Also *inst_term_rule*

Errors

3007 ?0 is not a term variable
6027 Types of element (?0, ?1) in term association list differ

SML
```
val asm_inst_type_rule : (TYPE * TYPE) list -> THM -> THM;
```

Description Parallel instantiation of some of the type variables of both the conclusion and assumptions of a theorem.

Rule

$$\frac{\Gamma \vdash t[tv1,...tvn]}{\Gamma' \vdash t[\sigma 1,...\sigma n]} \qquad \begin{array}{l} asm\_inst\_type\_rule \\ [(\sigma 1,\ tv1),\ ...,\ (\sigma n,tvn)] \end{array}$$

asm_inst_type_rule talist thm will instantiate each type variable in *talist* with its associated type. It will decorate free variables that would become identified with other variables by their types becoming the same and the names originally being the same. α-convertible duplicate assumptions will be eliminated.

See Also *inst_type_rule*

Errors

3019 ?0 is not a type variable

SML
```
val asm_intro : TERM -> THM -> THM;
```

Description Introduce a new assumption to an existing theorem.

Rule

$$\frac{\Gamma \vdash t2}{\Gamma \cup \{t1\} \vdash t2} \qquad \begin{array}{l} asm\_intro \\ \ulcorner t1 \urcorner \end{array}$$

Errors

3031 ?0 is not of type ⌜:BOOL⌝

SML

val **asm_rule** : *TERM* −> *THM*;

Description "A term is true on the assumption that it is true."

Rule

$$\frac{}{t \vdash t} \qquad \begin{array}{l} asm\_rule \\ \ulcorner t \urcorner \end{array}$$

A primitive inference rule.

Errors

3031 ?0 is not of type ⌜:BOOL⌝

SML

val **BINDER_C** : *CONV* −> *CONV*;

Description Apply a conversion to the body of a binder term:

Rule

$$\frac{}{\vdash (B\ x\bullet\ p[x]) = (B\ x\bullet\ pa[x])} \qquad \begin{array}{l} BINDER\_C \\ (c : CONV) \\ \ulcorner B\ x\bullet\ p \urcorner \end{array}$$

where $c\ p[x]$ gives '$\vdash p[x] = pa[x]$', and B is a binder.

Errors

27035 ?0 is not of the form: ⌜B x• p[x]⌝ where ⌜B⌝ is a binder and ⌜x⌝ a varstruct
7104 Result of conversion, ?0, ill−formed

Also as the failure of the conversion.

SML

val **CHANGED_C** : *CONV* −> *CONV*;

Description Applies a conversion, and fails if either the conversion fails, has ill-formed results in certain ways, or it causes no change. Even α-convertible changes count as a change for this purpose.

Errors

7032 Conversion failed to cause a change
7104 Result of conversion, ?0, ill−formed

It may also fail with the error message of the conversion argument.

SML

val **char_conv** : *CONV*;

Description This function defines the character literal constants, by giving a relationship between character literal constants and their ASCII code (derived by the Standard ML function *ord*). A character literal is indicated by the constant's name starting with single backquote (‘), being a single other character, as well as being of type *CHAR*.

Rule

$$\frac{}{\vdash {}_{ML}\ulcorner(mk\_char("c"))\urcorner = AbsChar\ {}_{ML}\ulcorner ord\ "c"\urcorner} \qquad \begin{array}{l} char\_conv \\ (mk\_char("c")) \end{array}$$

A primitive inference rule(axiom schemata).

See Also *mk_char*

Errors

3024 ?0 is not a character literal

SML
```
val COND_C : (TERM -> bool) -> CONV -> CONV -> CONV;
```

Description *COND_C pred cnv1 cnv2 tm* will be, if the term predicate *pred* applied to *tm* is true, then *cnv1 tm* and otherwise the *cnv2 tm*.

Errors As the failure of the predicate or either conversion.

SML
```
val cond_thm : THM;
```

Description A convenient variant of the definition of the conditional.

Theorem

$$\frac{}{\vdash \forall\ a\ t1\ t2 \bullet (if\ a\ then\ t1\ else\ t2) = (\epsilon\ x \bullet ((a \Leftrightarrow T) \Rightarrow x = t1) \land ((a \Leftrightarrow F) \Rightarrow x = t2))} \quad cond\_thm$$

SML
```
val contr_rule : TERM -> THM -> THM;
```

Description Intuitionistic contradiction rule:

Rule

$$\frac{\Gamma \vdash F}{\Gamma \vdash t} \quad \begin{array}{l} contr\_rule \\ \ulcorner t \urcorner \end{array}$$

Errors
```
7001   ?0 is not of form: ‘Γ ⊢ F‘
3031   ?0 is not of type ⌜:BOOL⌝
```

SML
```
val conv_rule : CONV -> THM -> THM;
```

Description Apply a conversion to the conclusion of a theorem, and do ⇔ modus ponens between the original theorem and the result of the conversion

Rule

$$\frac{\Gamma 1 \vdash t}{\Gamma 1 \cup \Gamma 2 \vdash t'} \quad \begin{array}{l} conv\_rule \\ (c : CONV) \end{array}$$

where $c\ t$ gives $\Gamma 2 \vdash t \Leftrightarrow t'$.

Errors
```
7104   Result of conversion, ?0, ill-formed
```

Also as the failure of the conversion upon the conclusion of the theorem.

SML
```
val cthm_eqn_cxt : CANON -> THM -> EQN_CXT;
```

Description This function applies a canonicalisation (see *CANON*) to a theorem, and then attempts to convert each of the list of resulting theorems into an equational context entry using *thm_eqn_cxt* (q.v.). The results are composed into an equational context (which is only a Standard ML list of equational context entries). Canoncalised theorems that cannot be converted by *thm_eqn_cxt* will be discarded.

SML

val **c_contr_rule** : $TERM \; -> \; THM \; -> \; THM$;

Description Classical contradiction rule:

Rule

$$\frac{\Gamma, \neg t' \vdash F}{\Gamma \vdash t} \quad \begin{array}{l} c\_contr\_rule \\ \ulcorner t \urcorner \end{array}$$

Note that the argument is the unnegated form of what must be present in the assumption list for success. Works up to α-conversion.

Errors

7001	*?0 is not of form: '$\Gamma \vdash F$'*
3031	*?0 is not of type ⌜:BOOL⌝*
7003	*Negation of ?0 is not in assumption list*

SML

val **disch_rule** : $TERM \; -> \; THM \; -> \; THM$;

Description Prove an implicative theorem, removing, if α-convertibly present, the antecedent of the implication from the assumption list, and failing if it is not present.

Rule

$$\frac{\Gamma, t1' \vdash t2}{\Gamma \vdash t1 \Rightarrow t2} \quad \begin{array}{l} disch\_rule \\ \ulcorner t1 \urcorner \end{array}$$

See Also *⇒_intro* (which does not fail if term not in assumption list)

Errors

7031	*?0 not $\alpha-$convertibly present in assumption list*

SML

val **eq_match_conv1** : $THM \; -> \; CONV$;

Description This matches the LHS of an universally quantified (simple or by varstruct) equational theorem to a term, instantiating the RHS accordingly. The conversion will only instantiate its universal quantifications, and type variables not found within the assumptions, not its free term variables.

Conversion

$$\frac{}{\Gamma \vdash t = v[t1,...,tn]} \quad \begin{array}{l} eq\_match\_conv1 \\ (\Gamma \vdash \forall \; x1 \; ... \; xn \bullet \; u[x1,...,xn] = \\ \quad v[x1,...,xn]) \\ \ulcorner t \urcorner \end{array}$$

where ⌜u[t1,...,tn]⌝ is α-convertible to ⌜t⌝. If there are free variables on the RHS of the supplied equational theorem (when stripped of all universal quantification) they will be renamed as necessary to avoid identification with any variables in t.

This conversion may be partially evaluated with only its theorem argument.

Uses In producing a limited rewriting facility, that only instantiates explicitly identified variables.

Errors

27003	*?0 is not of the form '$\Gamma \vdash \forall \; x1 \; ... \; xn \bullet \; u = v$'* *where ⌜xi⌝ are varstructs*
7076	*Could not match term ?0 to LHS of theorem ?1*

SML

```
val eq_match_conv : THM -> CONV ;
```

Description This matches the LHS of an equational theorem to a term, instantiating the RHS accordingly. The equational theorem may be partially or fully universally quantified (simple or by varstruct), without affecting the result of the conversion.

Conversion

$$\frac{}{\Gamma \vdash t = v'} \quad \begin{array}{l} eq\_match\_conv \\ (\Gamma \vdash \forall \ldots \bullet\ u = v) \\ \ulcorner t \urcorner \end{array}$$

where v' is the result of applying to v the instantiation rules required to match u to t (including both term and type instantiation). If there are free variables on the RHS of the supplied equational theorem (when stripped of all universal quantification) they will be renamed as necessary to avoid identification with any variables in t.

This conversion may be partially evaluated with only its theorem argument.

See Also *eq_match_conv1*

Errors

7044	*Cannot match ?0 and ?1*

SML

val **eq_rewrite_thm** : *THM*
val **⇔_rewrite_thm** : *THM*
val **¬_rewrite_thm** : *THM*
val **∧_rewrite_thm** : *THM*
val **∨_rewrite_thm** : *THM*
val **⇒_rewrite_thm** : *THM*
val **if_rewrite_thm** : *THM*
val **∀_rewrite_thm** : *THM*
val **∃_rewrite_thm** : *THM*
val **β_rewrite_thm** : *THM*

Description These are some of the default list of theorems used by those rewriting rules, conversions and tactics whose names do not begin with '*pure_*':

eq_rewrite_thm ⊢ ∀ x•(x = x) ⇔ T

⇔_rewrite_thm ⊢ ∀ t•((T ⇔ t) = t) ∧ ((t ⇔ T) = t) ∧
((F ⇔ t) = (¬ t)) ∧ (t ⇔ F) = (¬ t)

¬_rewrite_thm ⊢ ∀ t•(¬¬t) = t ∧ ((¬ T) = F) ∧ (¬ F) = T

∧_rewrite_thm ⊢ ∀ t•((T ∧ t) = t) ∧ ((t ∧ T) = t) ∧
(¬ (F ∧ t)) ∧ (¬ (t ∧ F)) ∧ (t ∧ t) = t

∨_rewrite_thm ⊢ ∀ t•(T ∨ t) ∧ (t ∨ T) ∧ ((F ∨ t) = t) ∧ ((t ∨ F) = t) ∧ (t ∨ t) = t

⇒_rewrite_thm ⊢ ∀ t•((T ⇒ t) = t) ∧ ((F ⇒ t) = T) ∧ ((t ⇒ T) = T) ∧ ((t ⇒ t) = T)
∧ (t ⇒ F) = (¬ t)

if_rewrite_thm ⊢ ∀ t1 t2:'a•((if T then t1 else t2) = t1) ∧ (if F then t1 else t2) = t2

∀_rewrite_thm ⊢ ∀ t•(∀ x•t) = t
∃_rewrite_thm ⊢ ∀ t•(∃ x•t) = t

β_rewrite_thm ⊢ ∀ t1:'a; t2:'b•((λ x•t1)t2) = t1

The theorems are saved in the theory "misc", and given their design in the design for that theory.

See Also *fst_rewrite_thm*, *snd_rewrite_thm*, *fst_snd_rewrite_thm*.

SML

val **eq_sym_conv** : *CONV*;

Description Symmetry of equality:

Rule

⊢ (t1 = t2) ⇔ (t2 = t1)	*eq_sym_conv* ⌜t1 = t2⌝

See Also *eq_sym_rule*

Errors

3014 *?0 is not of form*: ⌜t = u⌝

SML
```
val eq_sym_rule : THM -> THM;
```

Description Symmetry of equality:

Rule

$$\frac{\Gamma \vdash t1 = t2}{\Gamma \vdash t2 = t1} \qquad eq\_sym\_rule$$

A built-in inference rule.

See Also *eq_sym_conv*

Errors

6020 ?0 is not of the form: '$\Gamma \vdash t1 = t2$'

SML
```
val eq_trans_rule : THM -> THM -> THM;
```

Description Transitivity of equality:

Rule

$$\frac{\Gamma 1 \vdash t1 = t2;\ \Gamma 2 \vdash t2' = t3}{\Gamma 1 \cup \Gamma 2 \vdash t1 = t3} \qquad eq\_trans\_rule$$

where $t2$ and $t2'$ are α convertible. A built-in inference rule.

Errors

6020 ?0 is not of the form: '$\Gamma \vdash t1 = t2$'
6022 ?0 and ?1 are not of the form: '$\Gamma 1 \vdash t1 = t2$' and '$\Gamma 2 \vdash t2a = t3$' where ⌜t2⌝ and ⌜t2a⌝ are α−convertible

SML
```
val EVERY_CAN : CANON list -> CANON
```

Description *EVERY_CAN* is a canonicalisation function combinator which combines the elements of its argument using *THEN_CAN*:

```
EVERY_CAN [can1, can2, ...] = can1 THEN_CAN can2 THEN_CAN ...
```

See Also *CANON*

SML
```
val EVERY_C : CONV list -> CONV;
```

Description Apply each conversion in the list, in the sequence given.

See Also *THEN_C*(which this function iterates)

Errors

7103 List may not be empty

or as the failure of any constituent conversion, or as *THEN_C*.

SML

```
val ext_rule : THM -> THM;
```

Description Extensionality of functions in ICL HOL.

Rule

$$\frac{\Gamma \vdash f = g}{\Gamma \vdash \forall\ x \bullet f\ x = g\ x} \qquad ext\_rule$$

where x is a machine-generated variable of appropriate type, not found free in the equational theorem.

Errors

6020 ?0 is not of the form: ‘Γ ⊢ t1 = t2‘
7026 ?0 is not an equation of functions

SML

```
val fail_canon : CANON
```

Description This is a canonicalisation function which always fails. It is the identity for *ORELSE_CAN*.

See Also *CANON*

Errors

26201 Failed as requested

SML

```
val fail_conv : CONV;
```

Description This conversion always fails.

Errors

7061 Failed as requested

SML

```
val fail_with_canon : string -> int -> (unit -> string) list -> CANON
```

Description This is a canonicalisation function which always fails by passing its arguments to *fail* (q.v.).

See Also *fail_can*

SML

```
val fail_with_conv : string -> CONV;
```

Description This conversion always fails, with the error message being its string argument.

Errors

7075 ?0

SML

```
val FIRST_CAN : CANON list -> CANON
```

Description *FIRST_CAN* is a canonicalisation function combinator which combines the elements of its argument using *ORELSE_CAN*:

```
FIRST_CAN [can1, can2, ...] = can1 ORELSE_CAN can2 ORELSE_CAN ...
```

See Also *CANON*

Errors

26202 the list of canonicalisation functions is empty

SML

```
val FIRST_C : CONV list -> CONV;
```

Description Attempt to apply each conversion in the list, in the sequence given, until one succeeds, or all fail.

See Also *ORELSE_C*(which this function iterates)

Errors

```
7103   List may not be empty
```

or as the failure of the last conversion.

SML

```
val FORWARD_CHAIN_CAN : CANON list -> CANON;
val FC_CAN : CANON list -> CANON;
```

Description *FORWARD_CHAIN_CAN*, which has the alias *FC_CAN*, is a parameterised variant of *fc_canon*. Given a list of canonicalisation functions *cans*, *FC_CAN cans* behaves as *fc_canon* would do if the line

```
⊢ A          →        FIRST_CAN cans A
```

were inserted at the beginning of the table of transformations given in the description of *fc_canon*.

For example, *fc_canon1*, q.v., is the same as:

```
        FC_CAN ((fn (x, y) => [x,y]) o ⇔_elim);
```

Uses In tactic programming, or, occasionally interactively, typically in circumstances where neither *fc_canon* nor *fc_canon1* is able to generate enough implications.

SML

```
val forward_chain_canon : THM -> THM list;
val fc_canon : THM -> THM list;
val forward_chain_canon1 : THM -> THM list;
val fc_canon1 : THM -> THM list;
```

Description *forward_chain_canon* is a canonicalisation function which uses a theorem to generate a list of implications. (*fc_canon* is an alias for *forward_chain_canon*.) It may be used for constructing rules and tactics in conjunction with *forward_chain_rule*. An example of such a tactic is *forward_chain_tac*. *forward_chain_canon1*, which has alias *fc_canon1*, is just like *fc_canon* except for its treatment of bi-implications. The effects of *fc_canon* and *fc_canon1* are shown schematically in the following table (which only shows assumptions relevant to the process):

$\vdash A \land B$	$\rightarrow$	$\vdash A\ ;\ \vdash B$	
$\vdash \forall x\bullet A$	$\rightarrow$	$\vdash A[x'/x]$	
$\vdash A \land B \Rightarrow C$	$\rightarrow$	$map\ (\Rightarrow\_intro\ (st\ulcorner A\urcorner))\ (xf(\ st\ulcorner A\urcorner \vdash B \Rightarrow C\ ))$	
$\vdash A \lor B \Rightarrow C$	$\rightarrow$	$xf(\ \vdash (A \Rightarrow C) \land (B \Rightarrow C)\ )$	
$\vdash (\exists x\bullet A) \Rightarrow C$	$\rightarrow$	$map\ (\forall\_intro\ \ulcorner x'\urcorner)\ (xf(\ \vdash A[x'/x] \Rightarrow B\ ))$	
$A \vdash A \Rightarrow B$	$\rightarrow$	$A \vdash B$	
$\vdash T \Rightarrow B$	$\rightarrow$	$\vdash B$	
$A \vdash \neg A \Rightarrow B$	$\rightarrow$	(∗ *discarded* ∗)	
$\vdash F \Rightarrow B$	$\rightarrow$	(∗ *discarded* ∗)	
$\vdash A \Rightarrow B$	$\rightarrow$	$map\ (\Rightarrow\_intro\ (st\ulcorner A\urcorner))\ (xf(\ st\ulcorner A\urcorner \vdash B\ ))$	
$\vdash A \Leftrightarrow B$	$\rightarrow$	$\vdash A \Rightarrow B$	(∗ *fc_canon* ∗)
$\vdash A \Leftrightarrow B$	$\rightarrow$	$\vdash A \Rightarrow B; \vdash B \Rightarrow A$	(∗ *fc_canon1* ∗)
$\vdash T$	$\rightarrow$	(∗ *discarded* ∗)	
$\vdash A$	$\rightarrow$	$\vdash sc\ulcorner A\urcorner$	
$\vdash A$	$\rightarrow$	$\vdash \neg A \Rightarrow F$	

The intention here is that is that the first applicable transformation is applied repeatedly until no further change is possible. The resulting theorems are then universally quantified over all of the free variables in their conclusions which were not free in the original theorem. In the table, *st* and *sc* stand for attempts to apply the theorem and conclusion stripping conversions in the current proof context (as returned by *current_ad_st_conv* and *current_ad_sc_conv*). If the stripping conversions fail then *st* and *sc* have no effect. x' denotes a variable name derived from x and chosen to avoid variable capture problems. *xf* stands for a nested recursive application of the transformation process.

In the transformations involving $\Rightarrow$*_intro* the implication is only introduced if the antecedent is in the assumptions. So, for example, $A \Rightarrow B \Rightarrow A \Rightarrow C$ is transformed into $B \Rightarrow A \Rightarrow C$. The transformation for $A \Rightarrow B$ is only applied if it changes the theorem, and the last of the transformations is only applied if A is neither an implication nor F.

The asymmetry in the rules is deliberate. E.g., they derive $A \Rightarrow B \Rightarrow C$ from $A \land B \Rightarrow C$, but not $B \Rightarrow A \Rightarrow C$. This is intended to give slightly finer control and to result in less duplication of results in the intended application in *forward_chain_tac*(q.v.).

See Also *forward_chain_rule*, *forward_chain_tac*, *FC_CAN*

SML
```
val forward_chain_rule : THM list -> THM list -> THM list;
val fc_rule : THM list -> THM list -> THM list;
```

Description This is a rule which uses a list of possibly universally quantified implications and a list of other theorems to infer new theorems, using the matching modus ponens rule from the proof context, if present, or *⇒_match_mp_rule2*if *current_ad_mmp_rule*() returns *Nil*. (*fc_rule* is an alias for *forward_chain_rule*.) *fc_rule imps ants* returns the list of all theorems which may be derived by applying the matching modus ponens rule to a theorem from *imps* and one from *ants*. As a special case, if any theorem to be returned is determined to have ⌜*F*⌝ as its conclusion, the first such found wil be returned as a singleton list. In order to work well in conjunction with *fc_canon* and *fc_tac* the theorems returned by the matching modus ponens rule are transformed as follows:

1. Theorems of the form: $\vdash \forall\ x_1\ ...\bullet\ t_1 \Rightarrow t_2 \Rightarrow ... \Rightarrow \neg t_k \Rightarrow F$ have their final implication changed to t_k.
2. Theorems of the form: $\vdash \forall\ x_1\ ...\bullet\ t_1 \Rightarrow t_2 \Rightarrow ... \Rightarrow t_k \Rightarrow F$ have their final implication changed to ⇒\\$\neg t_k$.
3. All theorems are universally quantified over all the variables which appear free in their conclusions but not in their assumptions (using *all_∀_intro*).

Note that when the matching modus ponens rule is either *⇒_match_mp_rule2* or *⇒_match_mp_rule1*, there is some control over the number of results generated, since variables which appear free in *imps* are not considered as candidates for instantiation.

The rule does not check that the theorems in its first argument are (possible universally) quantified implications.

See Also *forward_chain_tac*, *forward_chain_canon*.

SML
```
val FORWARD_CHAIN_⇔_CAN : CANON list -> CANON;
val FC_⇔_CAN : CANON list -> CANON;
```

Description These are just like *FORWARD_CHAIN_CAN*, q.v., except that they do *not* break up bi-implications. Thus, given a list of canonicalisation functions *cans*, *FC_⇔_CAN cans* behaves as *fc_canon* would do if the line

⊢ *A*	→	*FIRST_CAN cans A*

were inserted at the beginning of the table of transformations given in the description of *fc_canon*and all transformations (including those coming from the proof context) that eliminate bi-implications were suppressed.

Uses In tactic programming, or, occasionally interactively, typically in circumstances where *fc_⇔_canon* is not able to generate enough implications.

SML
```
val forward_chain_⇔_canon : THM -> THM list;
val fc_⇔_canon : THM -> THM list;
```

Description *forward_chain_⇔_canon* is a canonicalisation function very similar to *forward_chain_canon*, q.v. The difference is that *forward_chain_⇔_canon* suppresses all transformations which break up bi-implications. It is intended for use in situations where a bi-implication is to be used as a conditional rewrite rule.

For example, the tactic *ALL_ASM_FC_T1 fc_⇔_canon rewrite_tac* [] can instantiate an assumption of the form $\forall x1\ x2\ ...\bullet\ A \Rightarrow B \Rightarrow (C \Leftrightarrow D)$ and use the result to rewrite instances of C.

See Also *FC_T1*, *ALL_FC_T1* etc.

SML
```
val f_thm : THM;
```

Description "Not False" is true.

Theorem

$$\frac{}{\vdash \neg F} \quad f\_thm$$

SML
```
val id_canon : CANON
```

Description This is the identity for the canonicalisation function combinator *THEN_CAN*:

```
id_canon thm = [thm]
```

See Also *CANON*

SML
```
val id_conv : CONV;
```

Description This is an alias for *refl_conv*, reflecting the fact that *refl_conv* is the identity for the conversional *THEN_C*.

Errors
```
7061  Failed as requested
```

SML
```
val if_app_conv : CONV;
```

Description Move a function application out of a conditional.

Conversion

$$\frac{}{\vdash (if\ a\ then\ f\ b\ else\ f'\ c) = f(if\ a\ then\ b\ else\ c)} \quad \begin{array}{l} if\_app\_conv \\ \ulcorner(if\ a\ then\ f\ b\ else\ f\ c)\urcorner \end{array}$$

where f and f' are α-convertible, and f is used on the RHS of the resulting equational theorem

Errors
```
7037  ?0 is not of the form: ⌜if a then (f b) else (g c)⌝
7038  ?0 is not of the form: ⌜if a then (f b) else (fa c)⌝
      where ⌜f⌝ and ⌜fa⌝ are α−convertible
```

SML

```
val if_else_elim : THM -> THM;
```

Description Give the dependence of the *else* branch of a conditional upon the condition.

Rule

$$\frac{\Gamma \vdash if\ \ tc\ \ then\ \ tt\ \ else\ \ te}{\Gamma \vdash \neg\ tc \Rightarrow te} \qquad if\_else\_elim$$

Errors

7012 ?0 is not of the form: ‘Γ ⊢ if tc then tt else te‘

SML

```
val if_intro : TERM -> THM -> THM -> THM;
```

Description Introduce a conditional, based on the assumptions of two theorems.

Rule

$$\frac{\Gamma 1,\ a \vdash tt\ ;\ \ \Gamma 2,\ \neg a' \vdash et}{\Gamma 1 \cup \Gamma 2 \vdash if\ \ a\ \ then\ \ tt\ \ else\ \ et} \qquad \begin{array}{l} if\_intro \\ \ulcorner a \urcorner \end{array}$$

where a and a' are α-convertible. Actually, the assumptions may be missing, and the rule still works.

Example

```
(⊢ x = tt), (⊢ x = te)                (* hypothesis *)
⊢ if a then (x = tt) else (x = te)    (* if_intro ⌜a⌝ *)
⊢ x = if a then tt else te            (* if_fun_rule *)
```

Errors

3031 ?0 is not of type ⌜:BOOL⌝

SML

```
val if_then_elim : THM -> THM;
```

Description Give the dependence of the *then* branch of a conditional upon the condition.

Rule

$$\frac{\Gamma \vdash if\ \ tc\ \ then\ \ tt\ \ else\ \ te}{\Gamma \vdash tc \Rightarrow tt} \qquad if\_then\_elim$$

Errors

7012 ?0 is not of the form: ‘Γ ⊢ if tc then tt else te‘

SML
```
val initial_rw_canon : CANON;
```

Description This is the initial rewrite canonicalisation function, defined as

```
val initial_rw_canon =
        REWRITE_CAN
        (REPEAT_CAN(FIRST_CAN [
        simple_∀_rewrite_canon,
        ∧_rewrite_canon,
        simple_¬_rewrite_canon,
        f_rewrite_canon,
        ⇔_t_rewrite_canon]));
```

This is the repeated application of the first applicable operation in the following list:

1. stripping universal quantifiers;
2. dividing conjunctive theorems into their conjuncts;
3. changing $\vdash\neg(t1\lor t2)$ to $\neg t1\land\neg t2$;
4. changing $\vdash\neg\exists x\bullet t$ to $\forall x\bullet\neg t$;
5. changing $\vdash\neg\neg t$ to t;
6. changing $\vdash\neg t$ to $t\Leftrightarrow F$;
7. changing $\vdash F$ to $\vdash \forall\ x \bullet x$;
8. if none of the above apply, changing $\vdash t$ to $\vdash t = T$.

   Finally, after all this canonicalisation we then universally quantify the resulting theorems in all free variables other than those that were free in the original.

SML
```
val inst_term_rule : (TERM * TERM) list -> THM -> THM;
```

Description Parallel instantiation of term variables within a theorem's conclusion to some other values.

Rule

$$\frac{\Gamma \vdash t[x1,\ ...,\ xn]}{\Gamma \vdash t[t1,\ ...,\ tn]} \quad inst\_term\_rule\ [...,\ (\ulcorner ti\urcorner,\ \ulcorner xi\urcorner),\ ...]$$

A built-in inference rule.

See Also *asm_inst_term_rule*

Errors

3007	?0 is not a term variable
6027	Types of element (?0, ?1) in term association list differ
6028	Instantiation variable ?0 free in assumption list

SML

val **inst_type_rule** : $(TYPE * TYPE)$ *list* $->$ *THM* $->$ *THM*;

Description Parallel instantiation of some of the type variables of the conclusion of a theorem.

Rule

$$\frac{\Gamma \vdash t[tyv1,...tyvn]}{\Gamma \vdash t[\sigma 1,...\sigma n]} \qquad \begin{array}{l} inst\_type\_rule \\ [(\sigma 1,\ tyv1),\ ...,\ (\sigma n,tyvn)] \end{array}$$

inst_type_rule talist thm will instantiate each type variable in *talist* with its associated type. It will decorate free variables that would become identified with other variables (both in conclusion and assumptions) by their types becoming the same and the names originally being the same. To instantiate types in the assumption list, see *asm_inst_type_rule*.

A primitive inference rule.

See Also *asm_inst_type_rule* for something that also works on type variables in the assumption list.

Errors

3019 ?0 is not a type variable
6006 Trying to instantiate type variable ?0, which occurs in assumption list

SML

val **LEFT_C** : *CONV* $->$ *CONV*;

Description Apply a conversion to the first operand of a binary operator:

Rule

$$\frac{}{\vdash f\ a\ b = f\ a'\ b} \qquad \begin{array}{l} LEFT\_C \\ (c : CONV) \\ \ulcorner f\ a\ b \urcorner \end{array}$$

where $c\ a$ gives $\vdash a = a'$. f may itself be a function application.

Errors

3013 ?0 is not of form: $\ulcorner f\ a\ b \urcorner$
7104 Result of conversion, ?0, ill−formed

Also as the failure of the conversion.

SML

val **let_conv** : *CONV*;

Description Eliminate an outermost *let* . . . *and* . . . *in* . . . construct.

Conversion

$$\frac{}{\begin{array}{c}\vdash (let\ vs1[x1,y1,..] = t1 \\ and\ ...\ and\ vsn[xn,yn,..] = tn \\ in\ t[x1,...,xn,...] \\ = t[t1x,...,t1y,...,tnx,tny,...]\end{array}} \qquad \begin{array}{l} let\_conv \\ \ulcorner let\ vs1[x1,y1,..] = \\ \ t1\ and\ ...\ vsn[xn,yn,..] = tn \\ \ in\ t[x1,...,xn,...]\urcorner \end{array}$$

Where the t_ix is the component of t_i matching x_i when t_i matches $vs\_i[x\_i, y\_i, ..]$.

Errors

4009 ?0 is not of form: $\ulcorner let\ ...\ in\ ...\urcorner$

SML

val **list_simple_∀_elim** : *TERM list −> THM −> THM*;

Description Generalised ∀ elimination.

Rule

$$\frac{\Gamma \vdash \forall\ x1\ \ldots\ xn \bullet t[x1,\ \ldots,\ xn]}{t[t1,\ \ldots,\ tn]} \quad \begin{array}{l} list\_simple\_\forall\_elim \\ [\ulcorner t1 \urcorner,\ \ldots,\ \ulcorner tn \urcorner] \end{array}$$

A built-in inference rule. The instantiation is done simultaneously, rather than by iteration of a single instantiation, which may affect renaming.

See Also *∀_elim*

Errors

3012 ?0 and ?1 do not have the same types
6018 ?0 is not of the form: ‘Γ ⊢ ∀ ...xi... • t‘ where the ⌜xi⌝ are ?1 variables

SML

val **list_simple_∀_intro** : *TERM list −> THM −> THM*;

Description Generalised simple ∀ introduction.

Rule

$$\frac{\Gamma \vdash t[x1,\ \ldots,\ xn]}{\Gamma \vdash \forall\ x1\ \ldots\ xn \bullet t[x1,\ \ldots,\ xn]} \quad \begin{array}{l} list\_simple\_\forall\_intro \\ [\ulcorner x1 \urcorner,\ \ldots,\ \ulcorner xn \urcorner] \end{array}$$

See Also *∀_intro*

Errors Same messages as *simple_∀_intro*.

SML

val **list_simple_∃_intro** : *TERM list −> TERM −> THM −> THM* ;

Description Introduce an iterated existential quantifier by providing a list of witnesses and a theorem asserting that the desired property holds of these witnesses.

Rule

$$\frac{\Gamma \vdash t[t1,t2,\ldots]}{\Gamma \vdash \exists\ x1\ x2 \ldots \bullet t[x1,x2,\ldots]} \quad \begin{array}{l} list\_simple\_\exists\_intro \\ [\ulcorner t1 \urcorner,\ \ulcorner t2 \urcorner,\ \ldots] \\ \ulcorner \exists\ x1\ x2\ \ldots \bullet t[x1,x2,\ldots] \urcorner \end{array}$$

Errors

7047 ?0 cannot be matched to conclusion of theorem ?1

SML

val **list_∧_intro** : *THM list −> THM*;

Description Conjoin a list of theorems.

Rule

$$\frac{[\Gamma 1 \vdash t1,\ \ldots,\ \Gamma n \vdash tn]}{\Gamma 1 \cup \ldots\ \Gamma n \vdash t1\ \wedge\ \ldots\ tn} \quad list\_\wedge\_intro$$

Errors

7107 List may not be empty

SML

val **list_∀_elim** : *TERM list -> THM -> THM*;

Description Generalised ∀ elimination. Specialise a universally quantified theorem with given values, instantiating the types of the theorem as necessary.

Rule

$$\frac{\Gamma \vdash \forall\ x1\ \ldots\ xn \bullet\ t[x1,\ \ldots,\ xn]}{t'[t1,\ \ldots,\ tn]} \qquad \begin{array}{l} list\_\forall\_elim \\ [\ulcorner t1 \urcorner,\ \ldots,\ \ulcorner tn \urcorner] \end{array}$$

where t' is renamed from t to prevent bound variable capture and type instantiated as necessary, the x_i are varstructs, instantiable to the structures of t_i. The values will be expanded using *Fst* and *Snd* as necessary to match the structure of $\ulcorner x \urcorner$.

Note that due to the type instantiation this function is somewhat more that a *fold* of *∀_elim*.

See Also *∀_elim*, *all_∀_elim*.

Errors

27014 *?0 is not of the form: ‘Γ ⊢ ∀ vs1 ... vsi • t‘ where i ≥ ?1*

27015 *?0 is not of the form: ‘Γ ⊢ ∀ vs1 ... vsi • t‘ where the types of the vsi are instantiable to the types of ?1*

27016 *?0 is not of the form: ‘Γ ⊢ ∀ vs1 ... vsi • t‘ where the types of the vsi are instantiable to the types of ?1 without instantiating type variables in the assumptions*

SML

val **list_∀_intro** : *TERM list -> THM -> THM*;

Description Generalised ∀ introduction.

Rule

$$\frac{\Gamma \vdash t[x1,\ \ldots,\ xn]}{\Gamma \vdash \forall\ x1\ \ldots\ xn \bullet\ t[x1,\ \ldots,\ xn]} \qquad \begin{array}{l} list\_\forall\_intro \\ [\ulcorner x1 \urcorner,\ \ldots,\ \ulcorner xn \urcorner] \end{array}$$

See Also *∀_intro*, *all_∀_intro*.

Errors Same messages as *∀_intro*.

SML

val **MAP_C** : *CONV -> CONV*;

Description This traverses a term from its leaves to its root node. It will repeat the application of its conversion argument, until failure, on each subterm encountered en route. At each node the conversion is applied to the sub-term that results from the application of the preceding traversal, not the original. It traverses from left to right, though this should only matter for conversions that work by side-effect. It fails if the conversion applies nowhere within the tree.

Errors

7005 *Conversion fails on term and all its subterms*

SML
```
val mk_app_rule : THM -> THM -> THM;
```

Description Given two equational theorems, one being between two functions, apply the two functions to the LHS and RHS of the other equation.

Rule

$$\frac{\Gamma 1 \vdash u1 = u2;\ \Gamma 2 \vdash v1 = v2}{\Gamma 1 \cup \Gamma 2 \vdash u1\ v1 = u2\ v2} \quad mk\_app\_rule$$

The second input theorem or the result may be expressed using ⇔.

A built-in inference rule.

Errors

6020 *?0 is not of the form: ‘Γ ⊢ t1 = t2‘*
6023 *?0 and ?1 are not of the form :‘Γ1 ⊢ u1 = u2‘ and ‘Γ2 ⊢ v1 = v2‘ where ⌜u1⌝ can be functionally applied to ⌜v1⌝*

SML
```
val modus_tollens_rule : THM -> THM -> THM;
```

Description If the consequent of an implicative theorem is false, then so must be the antecedent (modus tollens).

Rule

$$\frac{\Gamma 1 \vdash t1 \Rightarrow t2;\ \Gamma 2 \vdash \neg t2'}{\Gamma 1 \cup \Gamma 2 \vdash \neg t1} \quad modus\_tollens\_rule$$

where *t2* and *t2′* are α-convertible.

Errors

7040 *?0 is not of the form: ‘Γ ⊢ t1 ⇒ t2‘*
7051 *?0 and ?1 are not of the form: ‘Γ1 ⊢ t1 ⇒ t2‘ and ‘Γ2 ⊢ ¬t2a‘ where ⌜t2⌝ and ⌜t2a⌝ are α−convertible*

SML
```
val ONCE_MAP_C : CONV -> CONV;
```

Description This traverses a term from the root node to its leaves, attempting to apply its conversion argument. If it successfully applies the conversion to any subterm then it will not further traverse that subterm, but will still continue on other branches. If it fails to apply its conversion to a leaf, its functionality is equivalent to then applying *refl_conv*. It traverses from left to right, though this should only matter for conversions that work by side-effect. It will fail if the conversion succeeds nowhere in the tree, or if the results of certain conversion applications are ill-formed.

Errors

7005 *Conversion fails on term and all its subterms*

SML

val **ONCE_MAP_WARN_C** : *string* −> *CONV* −> *CONV*;

Description This is an equivalent to *ONCE_MAP_C* (q.v.) except that it warns the user if it failed to recompose the theorems from the term it just traversed.

This traverses a term from the root node to its leaves, attempting to apply its conversion argument. If it successfully applies the conversion to any subterm then it will not further traverse that subterm, but will still continue on other branches. If it fails to apply its conversion to a leaf, its functionality is equivalent to then applying *refl_conv*. It traverses from left to right, though this should only matter for conversions that work by side-effect. It will fail if the conversion succeeds nowhere in the tree, or if the results of certain conversion applications are ill-formed.

Errors

26001 no rewriting occurred
26003 no successful rewriting occurred, rewriting gave ill−formed results on some subterms

It issues the following warning message if at any point it fails to recompose the theorems from the subterm it just traversed, some successful rewriting occurs, and the flag "*illformed_rewrite-_warning*" is true.

Errors

26002 rewriting gave ill−formed results on some subterms

Errors and warnings are from the area indicated by the string argument.

SML

val **ORELSE_CAN** : (*CANON* ∗ *CANON*) −> *CANON*

Description *ORELSE_CAN* is a canonicalisation function combinator written as an infix operator. (*can1 ORELSE_CAN can2*)*thm* is the same *can1 thm* unless evaluation of *can1 thm* fails in which case it is the same as *can2 thm*.

See Also *CANON*

SML

val **ORELSE_C** : (*CONV* ∗ *CONV*) −> *CONV*;

Description Attempt to apply one conversion, and if that fails, try the second one.

Rule

$$\frac{}{\Gamma \vdash t = t'} \quad (c1\text{: } CONV)\ ORELSE\_C\ (c2\text{: } CONV)\ \ulcorner t \urcorner$$

where *c1 t* returns $\Gamma \vdash t = t'$, or *c1* fails, and *c2 t* returns $\Gamma \vdash t = t'$.

See Also *FIRST_C* (the iterated version of this function), *THEN_C*, *AND_OR_C*, and *THEN_TRY_C*

Errors As the failure of second conversion, should both conversions fail.

SML

val **plus_conv** : *CONV*;

Description Provides the value of the addition of two numeric literals.

Rule

$$\frac{}{\vdash \ulcorner {}_{\mathrm{ML}}\lceil mk\_\mathbb{N}\ m\urcorner + {}_{\mathrm{ML}}\lceil mk\_\mathbb{N}\ n\urcorner\urcorner = {}_{\mathrm{ML}}\lceil mk\_\mathbb{N}(m + n)\urcorner} \quad plus\_conv \ \ulcorner {}_{\mathrm{ML}}\lceil mk\_\mathbb{N}\ m\urcorner + {}_{\mathrm{ML}}\lceil mk\_\mathbb{N}\ n\urcorner\urcorner$$

Uses For doing fast arithmetic proofs.

Errors

6085 ?0 is not of the form: $\ulcorner {}_{\mathrm{ML}}\lceil mk\_\mathbb{N}\ m\urcorner + {}_{\mathrm{ML}}\lceil mk\_\mathbb{N}\ n\urcorner\urcorner$

SML

val **prim_rewrite_conv** : *CONV NET -> CANON -> (THM -> TERM * CONV) OPT ->*
(CONV -> CONV) -> EQN_CXT -> THM list -> CONV;

Description The primitive rewrite conversion.

Conversion

$$\frac{}{\Gamma \vdash t = t'}$$

prim_rewrite_conv
(initial_net: CONV NET)
(canon : CANON)
*(eqm_rule : (THM -> TERM * CONV) OPT)*
(traverse : CONV -> CONV)
(with_eqn_cxt : EQN_CXT)
(with_thms : THM list) ⌜*t*⌝

where ⌜t'⌝ is ⌜t⌝, rewritten according to the parameters of the conversion, and Γ are the assumptions required to allow the rewriting. The failure of the conversion constructed by *prim_rewrite_conv* will not be caught by *prim_rewrite_conv*.

The arguments have the following effects:

initial_net This is a pre-calculated conversion net, that will serve as the initial rewriting that may be done.

canon This canonicalisation function will be applied to all of the *with_thms* theorems, to produce a list of theorems to be rewritten with from these inputs. This will generally involve producing canonical or simplified forms of the original theorems.

The resulting theorems are intended to be simply universally quantified equations, and theorems which are not of this form are discarded. Rewriting attempts to instantiate some or all of the universally quantified variables, or any type variables (which do not appear in the assumptions), so as to to match the left-hand side of an equation to the term being rewritten. N.b. free variables are not instantiated. An equation whose left-hand side matches the term being rewritten in such a way that rewriting would not change the term is treated as if it did not match the term.

eqm_rule This equation matcher is mapped over the theorems resulting from the canonicalisation to convert them into an equation context. *thm_eqn_cxt* is used if *Nil* is supplied.

traverse This is a conversional, which defines the traversal of term *t* by the rewriting conversion derived from *prim_rewrite_conv*'s other arguments.

with_eqn_cxt This is additional equational context to be added directly into the rewriting conversion net.

with_thms This is an additional set of theorems to be processed by *canon* and the results used in added directly into the rewriting conversion net.

Uses This is the basis of the primary rewriting tools, by varying the first four parameters.

prim_rewrite_conv preprocesses its arguments in various ways. The preprocessing for an argument takes place as soon as that argument is supplied, so, for example, the overhead of preprocessing *with_eqn_cxt* need not be incurred in calls with the same *with_eqn_cxt* but different *with_thms*.

SML

val **prim_rewrite_rule** : $CONV\ NET\ {-}{>}\ CANON\ {-}{>}\ (THM\ {-}{>}\ TERM * CONV)\ OPT\ {-}{>}$
$(CONV\ {-}{>}\ CONV)\ {-}{>}\ EQN\_CXT\ {-}{>}\ THM\ list\ {-}{>}\ THM\ {-}{>}\ THM;$

Description This is the inference rule based on *prim_rewrite_conv* (q.v.), with the same parameters as that function, except for the last argument:

Rule

$$\frac{\Gamma \vdash t}{\Gamma \cup \Gamma 1 \vdash t'}$$

prim_rewrite_rule
(*initial_net*: *CONV NET*)
(*canon* : *CANON*)
(*epp* : (*THM* -> *TERM* * *CONV*) *OPT*)
(*traverse* : *CONV* -> *CONV*)
(*with_eqn_cxt* : *EQN_CXT*)
(*with_thms* : *THM list*)

where ⌜t'⌝ is the result of rewriting ⌜t⌝ in the manner prescribed by the arguments, and $\Gamma 1$ are the assumptions required to allow this rewriting.

SML

val **prim_suc_conv** : *CONV*;

Description This conversion gives the definition schema for all natural number literals.

Rule

$$\frac{}{\vdash {}_{\mathrm{ML}}\ulcorner(mk\_\mathbb{N}(m{+}1))\urcorner = Suc\ {}_{\mathrm{ML}}\ulcorner mk\_\mathbb{N}\ m\urcorner}$$

prim_suc_conv
(*mk_*ℕ (*m+1*))

Rule

$$\frac{}{\vdash {}_{\mathrm{ML}}\ulcorner(mk\_\mathbb{N}\ 0)\urcorner = Zero}$$

prim_suc_conv
(*mk_*ℕ *0*)

Errors

3026 ?0 is not a numeric literal

See Also *mk_*ℕ, *suc_conv*

SML

val **prove_asm_rule** : *THM* -> *THM* -> *THM*;

Description Eliminate an assumption with reference to a the assumption being a conclusion of a theorem.

Rule

$$\frac{\Gamma 1 \vdash t1;\ \Gamma 2,\ t1 \vdash t2}{\Gamma 1 \cup \Gamma 2 \vdash t2}$$

prove_asm_rule

This will in fact work even if the assumption is not present.

SML
```
val RANDS_C : CONV -> CONV;
```

Description Apply a conversion to each of the arguments of a a function

Rule

$$\frac{}{\vdash f\ a\ ...\ z = f\ a'...z'} \quad \begin{array}{l} RANDS\_C \\ (c : CONV) \\ \ulcorner f\ a\ ...\ z \urcorner \end{array}$$

where $c\ a$ gives $\vdash a = a'$, etc. The function f may have no arguments in which case *refl_conv f* is returned.

Errors

7104 Result of conversion, ?0, ill−formed

Also as the failure of the conversion.

SML
```
val RAND_C : CONV -> CONV;
```

Description Apply a conversion to the operand of a combination:

Rule

$$\frac{}{\vdash f\ a = f\ a'} \quad \begin{array}{l} RAND\_C \\ (c : CONV) \\ \ulcorner f\ a \urcorner \end{array}$$

where $c\ a$ gives $\vdash a = a'$.

Errors

3010 ?0 is not of form: ⌜t1 t2⌝
7104 Result of conversion, ?0, ill−formed

Also as the failure of the conversion.

SML
```
val RATOR_C : CONV -> CONV;
```

Description Apply a conversion to the operator of a combination:

Rule

$$\frac{}{\vdash f\ a = f'\ a} \quad \begin{array}{l} RATOR\_C \\ (c : CONV) \\ \ulcorner f\ a \urcorner \end{array}$$

where $c\ f$ gives '$\vdash f = f'$'.

Errors

3010 ?0 is not of form: ⌜t1 t2⌝
7104 Result of conversion, ?0, ill−formed

Also as the failure of the conversion.

SML
```
val refl_conv : CONV;
```

Description The reflexivity of equality implemented as a conversion.

Rule

$$\frac{}{\vdash t = t} \quad \begin{array}{l} refl\_conv \\ \ulcorner t \urcorner \end{array}$$

A primitive inference rule.

SML
```
val REPEAT_C1 : CONV -> CONV;
```

Description Repeatedly apply a conversion to a term, failing if not successfully applied at least once. To be more precise, the functionality is equivalent that of the following definition:

```
fun REPEAT_C1 (c:CONV) = (c THEN_TRY_C REPEAT_C1 c)
```

Errors As the error of the conversion if it cannot be applied at least once.

SML
```
val REPEAT_CAN : CANON -> CANON
```

Description *REPEAT_CAN* is a canonicalisation function combinator which repeatedly applies its argument until it fails:

```
REPEAT_CAN can thm =
	((can THEN_CAN REPEAT_CAN can) ORELSE_CAN id_can) thm
```

See Also *CANON*

SML
```
val REPEAT_C : CONV -> CONV;
```

Description Repeatedly apply a conversion to a term. To be more precise, the functionality is equivalent that of the following definition:

```
fun REPEAT_C (c:CONV) =
	(c THEN_C (REPEAT_C c)) ORELSE_C refl_conv
```

See Also *REPEAT_C1*

SML
```
val REPEAT_MAP_C : CONV -> CONV;
```

Description This traverses a term from its leaves to its root node. It will attempt the application of its conversion argument on each subterm encountered en route. If the conversion is successfully applied to a given sub-term, then the resulting sub-term from the conversion is re-traversed by the function. It traverses from left to right, though this should only matter for conversions that work by side-effect. It fails if the conversion is not applicable anywhere within the term, or if certain applications of the conversion have ill-formed results.

Errors
```
7005	Conversion fails on term and all its subterms
```

SML
```
val REWRITE_CAN : CANON -> CANON;
```

Description For rewriting, after all other canonicalisation we will usually wish to then universally quantify the resulting theorems in all free variables that are only in in the conclusion, other than those that were free anywhere in the original theorem, before any canonicalisation. A canonicalisation is transformed to work this way by *REWRITE_CAN*.

When evaluating proof contexts (see, e.g., *commit_pc*) the list of rewrite canonicalisations in the argument (see *get_rw_canons*), *arg*, will be converted to a single canonicalisation in the result by:

```
REWRITE_CAN
	(REPEAT_CAN(FIRST_CAN (arg @
		[⇔_t_rewrite_canon])));
```

SML
```
val rewrite_conv : THM list -> CONV;
val pure_rewrite_conv : THM list -> CONV;
val once_rewrite_conv : THM list -> CONV;
val pure_once_rewrite_conv : THM list -> CONV;
```

Description These are the standard rewriting conversions. They use the canonicalisation rule held by the proof context (see, e.g, *push_pc*) preprocess the theorem list. The context is accessed at the point when the rules are given a list of theorems.

If a conversion is "pure" then there is no default rewriting, otherwise the default rewriting conversion net held by the proof context will be used in addition to user supplied material.

If a conversion is "once" then rewriting will proceed from the root of the of the conclusion of the theorem to be rewritten, towards the leaves, and will not descend through any rewritten subterm, using *ONCE_MAP_WARN_C*. If not, rewriting will continue, moving from the root to the leaves, repeating if any rewriting is successful, until there is no rewriting redex anywhere within the rewritten conclusion, using *REWRITE_MAP_C*. This may cause non-terminating looping.

Errors
```
26001 no rewriting occurred
```

Also as error 26003 and warning 26002 of *REWRITE_MAP_C* (q.v.).

SML
```
val REWRITE_MAP_C : string -> CONV -> CONV;
```

Description This conversional is an equivalent to *TOP_MAP_C* (q.v.) except that it warns the user if it failed to recompose the theorems from the term it just traversed.

REWRITE_MAP_C conv tm traverses *tm* from its root node to its leaves. It will repeat the application of *conv*, until failure, on each subterm encountered en route. It then descends through the sub-term that results from the repeated application of *conv*. If the descent causes any change, on "coming back out" to the sub-term the conversional will attempt to reapply *conv*, and if successful will then (recursively) reapply *REWRITE_MAP_C conv* once more. If *conv* cannot be reapplied then the conversional continues to ascend back to the root.

It traverses from left to right, though this should only matter for conversions that work by side-effect. It fails if the conversion is applied nowhere within the term.

Errors
```
26001 no rewriting occurred
26003 no successful rewriting occurred, rewriting gave ill-formed results on some subterms
```

It issues the following warning message if at any point it fails to recompose the theorems from the subterm it just traversed, some successful rewriting occurs, and the flag "*illformed_rewrite_warning*" is true.

Errors
```
26002 rewriting gave ill-formed results on some subterms
```

Errors and warnings are from the area indicated by the string argument.

SML
```
val rewrite_rule : THM list -> THM -> THM;
val pure_rewrite_rule : THM list -> THM -> THM;
val once_rewrite_rule : THM list -> THM -> THM;
val pure_once_rewrite_rule : THM list -> THM -> THM;
val asm_rewrite_rule : THM list -> THM -> THM;
val pure_asm_rewrite_rule : THM list -> THM -> THM;
val once_asm_rewrite_rule : THM list -> THM -> THM;
val pure_once_asm_rewrite_rule : THM list -> THM -> THM;
```

Description These are the standard rewriting rules. They use the canonicalisation rule held by the proof context (see, e.g, *push_pc*) to preprocess the theorem list. The context is accessed at the point when the rules are given a list of theorems.

If a rule is "pure" then there is no default rewriting, otherwise the default rewriting conversion net held by the proof context will be used in addition to user supplied material.

If a rule is "once" then rewriting will proceed from the root of the of the conclusion of the theorem to be rewritten, towards the leaves, and will not descend through any rewritten subterm, using *ONCE_MAP_WARN_C*. If not, rewriting will continue, moving from the root to the leaves, repeating if any rewriting is successful, until there is no rewriting redex anywhere within the rewritten conclusion, using *REWRITE_MAP_C*. This may cause non-terminating looping.

If a rule is "asm" then the theorems rewritten with will include the canonicalised *asm_rule*d assumptions of the theorem being rewritten.

See Also *prim_rewrite_rule*

Errors
```
26001 no rewriting occurred
```

Also as error 26003 and warning 26002 of *REWRITE_MAP_C*(q.v.).

SML
```
val RIGHT_C : CONV -> CONV;
```

Description Apply a conversion to the second operand of a binary operator:

Rule

$$\frac{}{\vdash f\ a\ b = f\ a\ b'} \quad \begin{array}{l} RIGHT\_C \\ (c : CONV) \\ \ulcorner f\ a\ b \urcorner \end{array}$$

where c b gives '$\vdash b = b'$'. f may itself be a function application.

Errors
```
3013 ?0 is not of form: ⌜f a b⌝
7104 Result of conversion, ?0, ill-formed
```

Also as the failure of the conversion.

SML
```
val SIMPLE_BINDER_C : CONV -> CONV;
```

Description Apply a conversion to the body of a simple binder term:

Rule

$$\frac{}{\vdash (B\ x \bullet p[x]) = (B\ x \bullet p'[x])} \quad \begin{array}{l} SIMPLE\_BINDER\_C \\ (c : CONV) \\ \ulcorner B\ x \bullet p \urcorner \end{array}$$

where $c\ p[x]$ gives '$\vdash p[x] = p'[x]$', and B is a binder.

Errors

7059 *?0 is not of the form:* $\ulcorner B\ x \bullet p[x] \urcorner$ *where* $\ulcorner B \urcorner$ *is a binder and* $\ulcorner x \urcorner$ *a variable*

7104 *Result of conversion, ?0, ill−formed*

Also as the failure of the conversion.

SML
```
val simple_eq_match_conv : THM -> CONV ;
```

Description This matches the LHS of an equational theorem to a term, instantiating the RHS accordingly. In fact the equation may be partially or fully universally quantified (simple quantification only), without affecting the result of the conversion.

Conversion

$$\frac{}{\Gamma' \vdash t = v'} \quad \begin{array}{l} simple\_eq\_match\_conv \\ (\Gamma \vdash \forall \ldots \bullet u = v) \\ \ulcorner t \urcorner \end{array}$$

where v' is the result of applying to v the instantiation rules required to match u to t (including both term and type instantiation). If there are free variables on the RHS of the supplied equational theorem (when stripped of all universal quantification) they will be renamed as necessary to avoid identification with any variables in t.

Errors

7044 *Cannot match ?0 and ?1*

SML

val **simple_eq_match_conv1** : *THM* −> *CONV* ;

Description This matches the LHS of an universally quantified (simple quantifiers only) equational theorem to a term, instantiating the RHS accordingly. The conversion will only instantiate its universal quantifications, and type variables not present in the assumptions, and not its free term variables.

Conversion

$$\frac{}{\Gamma \vdash t = v[t1,...,tn]} \quad \begin{array}{l} simple\_eq\_match\_conv1 \\ (\Gamma \vdash \forall\ x1\ ...\ xn \bullet u[x1,...,xn] = \\ \quad v[x1,...,xn]) \\ \ulcorner t \urcorner \end{array}$$

where ⌜u[t1,...,tn]⌝ is α-convertible to ⌜t⌝. If there are free variables on the RHS of the supplied equational theorem(when stripped of all universal quantification) they will be renamed as necessary to avoid identification with any variables in t.

This conversion may be partially evaluated with only its theorem argument.

Uses In producing a limited rewriting facility, that only instantiates explicitly identified variables.

Errors

7095 ?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u = v‘ where ⌜xi⌝ are variables
7076 Could not match term ?0 to LHS of theorem ?1

SML

val **simple_ho_eq_match_conv** : *THM* −> *CONV*

Description This conversion is like *simple_eq_match_conv* but uses higher-order matching. It uses *ho_match* (q.v.) to match the LHS of an equational theorem to a term t. It then instantiates the theorem (including both term and type instantiation) and carries out any $\beta\eta$-reductions required to give a theorem of the form $t = v'$. The equation may be partially or fully universally quantified (simple quantification only, not quantification over pairs).

Conversion

$$\frac{}{\Gamma' \vdash t = v'} \quad \begin{array}{l} simple\_ho\_eq\_match\_conv \\ (\Gamma \vdash \forall\ ... \bullet u = v) \\ \ulcorner t \urcorner \end{array}$$

where v' is the result of applying to v the instantiations required to match u to t (including term and type instantiation). If there are free variables on the RHS of the supplied equational theorem (when stripped of all universal quantification) they will be renamed as necessary to avoid identification with any variables in t.

Errors

7095 ?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u = v‘ where ⌜xi⌝ are variables
7076 Could not match term ?0 to LHS of theorem ?1

SML

|*val* **simple_ho_eq_match_conv1** : *THM −> CONV*

Description This conversion is like *simple_eq_match_conv1* but uses higher-order matching. It uses *ho_match* (q.v.) to match the LHS of an equational theorem to a term t. The equation may be partially or fully universally quantified (simple quantification only, not quantification over pairs). It instantiates the theorem (including both term and type instantiation) and carries out any $\beta\eta$-reductions required to give a theorem of the form $t = v'$. Only type variables that do not appear in the assumptions of the theorem and universally quantified term variables will be instantiated.

Conversion

$$\frac{}{\Gamma \vdash t = v'} \qquad simple\_ho\_eq\_match\_conv1\ (\Gamma \vdash \forall \ldots \bullet u = v)\ \ulcorner t \urcorner$$

where v' is the result of applying to v the instantiation rules required to match u to t (including term and type instantiation). If there are free variables on the RHS of the supplied equational theorem (when stripped of all universal quantification) they will be renamed as necessary to avoid identification with any variables in t.

Errors

|*7095* *?0 is not of the form ‘$\Gamma \vdash \forall$ x1 ... xn $\bullet$ u = v‘ where $\ulcorner$xi$\urcorner$ are variables*
|*7076* *Could not match term ?0 to LHS of theorem ?1*

SML

|*val* **simple_⇔_match_mp_rule** : *THM −> THM −> THM*;

Description A matching Modus Ponens for ⇔.

Rule

$$\frac{\Gamma 1 \vdash \forall\ x1\ \ldots \bullet t1 \Leftrightarrow t2;\ \Gamma 2 \vdash t1'}{\Gamma 1' \cup \Gamma 2 \vdash t2'} \qquad simple\_\Leftrightarrow\_match\_mp\_rule$$

where *t1′* is an instance of *t1* under type instantiation and substitution for the x_i and the free variables of the first theorem, and where *t2′* is the corresponding instance of *t2*. No type instantiation or substitution will occur in the assumptions of either theorem.

See Also *⇒_elim* (Modus Ponens on ⇒), *simple_⇔_match_mp_rule*

Errors

|*7044* *Cannot match ?0 and ?1*
|*7046* *?0 is not of the form ‘$\Gamma \vdash \forall$ x1 ... xn $\bullet$ u ⇔ v‘*

SML

|*val* **simple_⇔_match_mp_rule1** : *THM −> THM −> THM*;

Description A matching Modus Ponens for ⇔ that doesn't affect assumption lists.

Rule

$$\frac{\Gamma 1 \vdash \forall\ x1\ \ldots \bullet t1 \Leftrightarrow t2;\ \Gamma 2 \vdash t1'}{\Gamma 1 \cup \Gamma 2 \vdash t2'} \qquad simple\_\Leftrightarrow\_match\_mp\_rule1$$

where *t1′* is an instance of *t1* under type instantiation and substitution for the x_i (but not free variables), and where *t2′* is the corresponding instance of *t2*. Types in the assumptions of the theorems will not be instantiated.

See Also *⇒_elim* (Modus Ponens on ⇒), *simple_⇔_match_mp_rule1*

Errors

|*7044* *Cannot match ?0 and ?1*
|*7046* *?0 is not of the form ‘$\Gamma \vdash \forall$ x1 ... xn $\bullet$ u ⇔ v‘*

SML

val **simple_⇒_match_mp_rule** : *THM −> THM −> THM* ;

Description A matching Modus Ponens rule for an implicative theorem.

Rule

$$\frac{\Gamma 1 \vdash \forall\ x1\ ...\ \bullet\ t1\ \Rightarrow\ t2;\ \Gamma 2 \vdash t1'}{\Gamma 1 \cup \Gamma 2 \vdash t2'} \qquad simple\_\Rightarrow\_match\_mp\_rule$$

where *t1′* is an instance of *t1* under type instantiation and substitution for the x_i and the free variables of the first theorem, and where *t2′* is the corresponding instance of *t2*. No type instantiation or substitution will occur in the assumptions of either theorem.

See Also *simple_⇒_match_mp_rule1*, *simple_⇒_match_mp_rule2*

Errors

7044 Cannot match ?0 and ?1
7045 ?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u ⇒ v‘

SML

val **simple_⇒_match_mp_rule1** : *THM −> THM −> THM* ;
val **simple_⇒_match_mp_rule2** : *THM −> THM −> THM* ;

Description Two variants on a matching Modus Ponens rule for an implicative theorem.

Rule

$$\frac{\Gamma 1 \vdash \forall\ x1\ ...\ \bullet\ t1\ \Rightarrow\ t2;\ \Gamma 2 \vdash t1'}{\Gamma 1 \cup \Gamma 2 \vdash t2'} \qquad simple\_\Rightarrow\_match\_mp\_rule1$$

where *t1′* is an instance of *t1* under type instantiation and substitution for the x_i (but not free variables), and where *t2′* is the corresponding instance of *t2*.

simple_⇒_match_mp_rule2 is just like *simple_⇒_match_mp_rule1* except that the instantiations and substitutions returned by *term_match* are extended to replace type variables that do not occur in *t1* or in *Γ1* and *x_i* that do not occur free in *t1* by fresh variables to avoid clashes with each other and with the type variables and free variables of *Γ1* and *Γ2*.

Types in the assumptions of the theorems will not be instantiated.

See Also *simple_⇒_match_mp_rule*

Errors

7044 Cannot match ?0 and ?1
7045 ?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u ⇒ v‘

SML

val **simple_∀_elim** : *TERM −> THM −> THM*;

Description Instantiate a universally quantified variable to a given value.

Rule

$$\frac{\Gamma \vdash \forall\ x\ \bullet\ t2[x]}{\Gamma \vdash t2'[t1]} \qquad \begin{array}{l} simple\_\forall\_elim \\ \ulcorner t1 \urcorner \end{array}$$

where *t2′* is renamed from *t2* to prevent bound variable capture, and *x* is a variable.

Errors

3012 ?0 and ?1 do not have the same types
7039 ?0 is not of the form: ‘Γ ⊢ ∀ x • t‘ where ⌜x⌝ is a variable

SML

val **simple_∀_intro** : *TERM −> THM −> THM*;

Description Introduce a simple universally quantified theorem.

Rule

$$\frac{\Gamma \vdash t}{\Gamma \vdash \forall\ x \bullet t} \qquad \textit{simple\_}\forall\textit{\_intro}\ \ulcorner x \urcorner$$

A built-in inference rule.

See Also *∀_intro*

Errors

3007	*?0 is not a term variable*
6005	*?0 occurs free in assumption list*

SML

val **simple_∀_∃_conv**: *CONV*;

Description Swap the order of a simple ∀ and ∃:

Conversion

$$\frac{}{\vdash (\forall\ x \bullet \exists\ y \bullet P[x,y]) \Leftrightarrow (\exists\ y' \bullet \forall\ x \bullet P[x,\ y'\ x])} \qquad \textit{simple\_}\forall\textit{\_}\exists\textit{\_conv}\ \ulcorner \forall\ x \bullet \exists\ y \bullet P[x,y] \urcorner$$

where y' is renamed to distinguish it from y (for the types differ) and every other term variable in the argument.

Errors

27031	*?0 is not of the form:* $\ulcorner \forall\ x \bullet \exists\ y \bullet P[x,y] \urcorner$

SML

val **simple_∃_elim** : *TERM −> THM −> THM −> THM* ;

Description Eliminate an existential quantifier.

Rule

$$\frac{\Gamma 1 \vdash \exists\ x \bullet t1[x];\ \Gamma 2,\ t1[y] \vdash t2}{\Gamma 1 \cup \Gamma 2 \vdash t2} \qquad \textit{simple\_}\exists\textit{\_elim}\ \ulcorner y \urcorner$$

where y must be variable which is not present elsewhere in the second theorem, nor in the conclusion of the first. $t1[y]$ need not actually be present in the assumptions of the second theorem.

Errors

3007	*?0 is not a term variable*
7014	*?0 has the wrong type*
7109	*?0 is not of the form* '$\Gamma \vdash \exists\ x \bullet t[x]$'
7120	*?0 occurs free in conclusion of ?1*
7121	*?0 occurs free in hypotheses of ?1 other than ?2*

SML
val **simple_∃_intro** : *TERM −> THM −> THM* ;

Description Introduce an existential quantifier by reference to a witness.

Rule

$$\frac{\Gamma \vdash t1[t2]}{\Gamma \vdash \exists\ x \bullet t1[x]} \quad \begin{array}{l} simple\_\exists\_intro \\ \ulcorner \exists\ x \bullet t1[x] \urcorner \end{array}$$

where ⌜x⌝ is a variable.

Errors
3034 ?0 is not of form: ⌜∃ *var* • *body*⌝
7047 ?0 cannot be matched to conclusion of theorem ?1

SML
val **simple_∃_∀_conv**: *CONV*;

Description Swap the order of a simple ∃ and ∀:

Conversion

$$\frac{}{\vdash (\exists\ x\bullet\ \forall\ y\bullet\ P[x,y]) \Leftrightarrow (\forall\ y'\bullet\ \exists\ x\bullet\ P[x,\ y'\ x])} \quad \begin{array}{l} simple\_\exists\_\forall\_conv \\ \ulcorner \exists\ x\bullet\ \forall\ y\bullet\ P[x,y] \urcorner \end{array}$$

where y' is renamed to distinguish it from y (for the types differ) and every other term variable in the argument.

Errors
27032 ?0 is not of the form: ⌜∃ *x*• ∀ *y*• *P*[*x*,*y*]⌝

SML
val **simple_∃_∀_conv1**: *CONV*;

Description Swap the order of a simple ∃ and ∀, where the first variable is always applied to the second:

Conversion

$$\frac{}{(\exists\ f\bullet\ \forall\ x\bullet\ P[f\ x,\ x]) \Leftrightarrow (\forall\ x\bullet\ \exists\ f'\bullet\ P[f',\ x])} \quad \begin{array}{l} simple\_\exists\_\forall\_conv1 \\ \ulcorner \exists\ f\bullet\ \forall\ x\bullet\ P[f\ x,\ x] \urcorner \end{array}$$

where f' is renamed to distinguish it from f (for the types differ) and every other term variable in the argument.

Errors
27033 ?0 is not of the form: ⌜∃ *f*• ∀ *x*• *P*[*f x*,*x*]⌝

SML
val **simple_∃_ϵ_conv** : *CONV*;

Description Give that ϵ of a predicate satisfies the predicate by reference to an ∃ construct.

Rule

$$\frac{}{\Gamma \vdash (\exists\ x \bullet p[x]) \Leftrightarrow p[\epsilon\ x \bullet p\ [x]]} \quad \begin{array}{l} simple\_\exists\_\epsilon\_conv \\ \ulcorner \exists\ x \bullet p[x] \urcorner \end{array}$$

See Also *∃_ϵ_rule*

Errors
3034 ?0 is not of form: ⌜∃ *var* • *body*⌝

SML

val **simple_∃_ϵ_rule** : $THM \; -> \; THM$;

Description Give that ϵ of a predicate satisfies the predicate by reference to an ∃ construct. It can properly handle paired existence.

Rule

$$\frac{\Gamma \vdash \exists\ x \bullet p[x]}{\Gamma \vdash p[\epsilon\ x \bullet p\ x]} \qquad simple\_\exists\_\epsilon\_rule$$

See Also $\exists\_\epsilon\_conv$

Errors

7092 *?0 is not of the form:* ‘$\Gamma \vdash \exists\ x \bullet p[x]$‘

SML

val **simple_∃₁_elim** : $THM \; -> \; THM$;

Description Express a $\exists_1$ in terms of ∃ and a uniqueness property.

Rule

$$\frac{\Gamma \vdash \exists_1\ x \bullet P[x]}{\Gamma \vdash \exists\ x \bullet P[x] \land \forall\ y \bullet P[y] \Rightarrow y = x} \qquad simple\_\exists_1\_elim$$

Errors

7015 *?0 is not of the form:* ‘$\Gamma \vdash \exists_1\ x \bullet P[x]$‘

SML

val **simple_∃₁_intro** : $THM \; -> \; THM \; -> \; THM$;

Description Introduce $\exists_1$ by reference to a witness, and a uniqueness theorem.

Rule

$$\frac{\Gamma 1 \vdash P'[t'] \quad \Gamma 2 \vdash \forall\ x \bullet P[x] \Rightarrow x = t}{\Gamma 1 \cup \Gamma 2 \vdash \exists_1\ x \bullet P[x]} \qquad simple\_\exists_1\_intro$$

Where P' is α-convertible to P, and t' is α-convertible to t. Notice that for the resulting theorem we take the bound variable name, x, and the form of the predicate, P, from the second theorem.

Errors

7066 *?0 not of the form:* ‘$\Gamma \vdash \forall\ x \bullet P[x] \Rightarrow x = t$‘
7067 *?0 and ?1 are not of the form:* ‘$\Gamma 1 \vdash Pa[ta]$‘ *and* ‘$\Gamma 2 \vdash \forall\ x \bullet P[x] \Rightarrow x = t$‘ *where* $\ulcorner Pa \urcorner$ *and* $\ulcorner P \urcorner$, $\ulcorner ta \urcorner$ *and* $\ulcorner t \urcorner$ *are* $\alpha-convertible$

SML

val **simple_α_conv** : $string \; -> \; CONV$;

Description Rename a bound variable name, as a conversion. This only works with simple abstractions.

Rule

$$\frac{}{\vdash (\lambda\ x \bullet t[x]) = (\lambda\ v \bullet t[v])} \qquad \begin{array}{l} simple\_\alpha\_conv \\ (v : string) \\ \ulcorner \lambda\ x \bullet t[x] \urcorner \end{array}$$

Errors

3011 *?0 is not of form:* $\ulcorner \lambda\ var \bullet t \urcorner$
7035 *Cannot rename bound variable ?0 to ?1 as this would cause variable capture*

SML

val **simple_β_conv** : $CONV$;

Description Apply a β-reduction to a simple abstraction.

Conversion

$$\frac{}{\vdash (\lambda\ x \bullet t1[x])\ t2 = t1[t2]} \qquad \begin{array}{l} simple\_\beta\_conv \\ \ulcorner(\lambda\ x \bullet t1[x])\ t2\urcorner \end{array}$$

A primitive inference rule.

See Also β_conv

Errors

6012 *?0 is not of the form:* $\ulcorner(\lambda\ x \bullet t1[x])t2\urcorner$ *where* $\ulcorner x\urcorner$ *is a variable*

SML

val **simple_β_η_conv** : $TERM\ {-}{>}\ CONV$;

Description If t is any term, *simple_β_η_conv* t is a conversion which will prove all theorems of the form $\vdash t = s$ where t and s are simply $\alpha\beta\eta$-equivalent, i.e., can be reduced to α-equivalent normal forms by β- and η-reduction involving only simple (rather than paired) λ-abstractions.

Errors

7131 *?0 and ?1 are not simply* $\alpha\beta\eta{-}equivalent$

SML

val **simple_β_η_norm_conv** : $CONV$;

Description This conversion eliminates all simple β- and η-redexes from a term giving the $\beta\eta$-normal form. It does not eliminate β- and η-redexes involving abstraction over pairs. It fails if the term is already in normal form.

Errors

7130 *?0 contains no simple* $\beta{-}$ *or* $\eta{-}redexes$

SML

val **simple_ϵ_elim_rule** : $TERM\ {-}{>}\ THM\ {-}{>}\ THM\ {-}{>}\ THM$;

Description Given that ϵ of a predicate satisfies that predicate, then in a different theorem we may eliminate an assumption that claims an otherwise unused variable satisfies the predicate.

Rule

$$\frac{\begin{array}{c}\Gamma 1 \vdash t'\ (\$\epsilon\ t'');\\ \Gamma 2,\ t\ x \vdash s\end{array}}{\Gamma 1 \cup \Gamma 2 \vdash s} \qquad \begin{array}{l} simple\_\epsilon\_elim\_rule \\ \ulcorner x\urcorner \end{array}$$

where t, t' and t'' are α-convertible, and x is a free variable whose only free occurrence in the second theorem is the one shown and which does not appear free in the conclusion of the first theorem. In fact, ($\$\epsilon\ t''$) here can be any term, it is not constrained to be an application of the choice function.

Errors

3007 *?0 is not a term variable*
7019 *?0 is not of the form:* '$\Gamma \vdash t1(\$\epsilon\ t1)$'
7054 *?0 is not of same type as choice sub−term of first theorem*
7108 *Arguments not of the form* $\ulcorner ?0\urcorner$, '$\Gamma 1 \vdash t\ (\$\epsilon\ t)$' *and* '$\Gamma 2,\ (t\ ?0) \vdash s$'
7120 *?0 occurs free in conclusion of ?1*
7121 *?0 occurs free in hypotheses of ?1 other than ?2*
7122 *?0 occurs free in operator of the conclusion of ?1*

SML

val **SIMPLE_λ_C** : $\mathit{CONV} \; -{>} \; \mathit{CONV}$;

Description Apply a conversion to the body of a simple abstraction:

Rule

$$\frac{}{\vdash (\lambda\ x \bullet p[x]) = (\lambda\ x \bullet p'[x])} \qquad \begin{array}{l} \mathit{SIMPLE\_\lambda\_C} \\ (c : \mathit{CONV}) \\ \ulcorner \lambda\ x \bullet p \urcorner \end{array}$$

where $c\ p[x]$ gives '$\vdash p[x] = p'[x]$'.

See Also *SIMPLE_BINDER_C*

Errors

3011 *?0 is not of form*: $\ulcorner \lambda\ var \bullet t \urcorner$
7104 *Result of conversion, ?0, ill−formed*

Also as the failure of the conversion.

SML

val **simple_λ_eq_rule** : $\mathit{TERM} \; -{>} \; \mathit{THM} \; -{>} \; \mathit{THM}$;

Description Given an equational theorem, return the equation formed by abstracting the term argument (which must be a variable) from both sides.

Rule

$$\frac{\Gamma \vdash t1[x] = t2[x]}{\Gamma \vdash (\lambda\ x \bullet t1[x]) = (\lambda\ x \bullet t2[x])} \qquad \begin{array}{l} \mathit{simple\_\lambda\_eq\_rule} \\ \ulcorner x \urcorner \end{array}$$

A primitive inference rule.

See Also *λ_eq_rule*

Errors

3007 *?0 is not a term variable*
6005 *?0 occurs free in assumption list*
6020 *?0 is not of the form*: '$\Gamma \vdash t1 = t2$'

SML

val **string_conv** : *CONV*;

Description This function defines the constants with names starting with ", and type *CHAR LIST* (an abbreviation of *CHAR LIST*). A string literal constant is indicated by the constant name starting with a double quote("), as well as being of type *CHAR LIST*. This is equivalent to a list of character literal constants, one for each but the first (") character of the string constant's name. This conversion defines this relationship, by returning the head and unexploded tail of the list of characters. A character literal is indicated by the constant's name starting with single backquote (‘), as well as being of type *CHAR*.

Rule

$$\frac{}{\vdash \ulcorner_{ML}(mk\_string("c...")) \urcorner = Cons\ \ulcorner_{ML}(mk\_char("c"))\urcorner\ \ulcorner_{ML}(mk\_string("..."))\urcorner} \quad string\_conv\ (mk\_string\ ("c..."))$$

Or:

Rule

$$\frac{}{\vdash \ulcorner_{ML}(mk\_string(""))\urcorner = Nil} \quad string\_conv\ (mk\_string\ "")$$

A primitive inference rule(axiom schemata).

See Also *mk_string*

Errors

3025 ?0 is not a string literal

SML

val **strip_∧_rule** : *THM −> THM list*;

Description Break a theorem into conjuncts as far as possible.

Rule

$$\frac{\Gamma \vdash t}{[\Gamma \vdash t1, ..., \Gamma \vdash tn]} \quad strip\_\wedge\_rule$$

where t can be formed from the t_i by *∧_intro* alone, with no duplication, exception or reordering.

Example

```
strip_∧_rule ‘⊢ (a ∧ b) ∧ (a ∧ c ∧ d)‘
 =
[‘⊢ a‘, ‘⊢ b‘, ‘⊢ a‘, ‘⊢ c‘, ‘⊢ d‘]
```

SML

val **strip_⇒_rule** : *THM −> THM*;

Description Repeatedly apply *undisch_rule*:

Rule

$$\frac{\Gamma \vdash t1 \Rightarrow ... \Rightarrow tn \Rightarrow t}{\Gamma \cup \{t1, ..., tn\} \vdash t} \quad strip\_\Rightarrow\_rule$$

SML

```
val subst_conv : (THM * TERM) list -> TERM -> CONV;
```

Description Substitution of equational theorems according to a template.

Conversion

$$\frac{}{\Gamma 1 \cup ... \Gamma n \vdash t[...,ti,...] = t[...,ti',...]} \quad \begin{array}{l} subst\_conv \\ [...,(`\Gamma i \vdash ti=ti'`,\ulcorner xi \urcorner), ...] \\ \ulcorner t[...,xi,...]\urcorner \\ \ulcorner t[...,ti,...]\urcorner \end{array}$$

subst_conv $[(thm\_1, x\_1), \ldots, (thm\_n, x\_n)]$ *template term* returns a theorem in which *template* determines where in *term* the *thm_i* are substituted, when forming the RHS of the equation. The *x_i* must be variables. The template is of the form $t[x\_1, \ldots, x\_n]$, and wherever the *x_i* are free in *template* their associated equational theorem, *thm_i*, is substituted into *thm*. The rule will rename as necessary to avoid bound variable capture. The assumption list of the resulting theorem will be the union of all substitution theorems, regardless of use.

The RHS of the resulting theorem will take its bound variable names from *template*, not *term*, as shown in the following example. This provides an α-conversion facility.

This function may be partially evaluated with only one argument.

Example

```
subst_conv [(`⊢ p = q`, ⌜x1⌝), (`⊢ r = s`, ⌜x2⌝)]
        (⌜∀ y • f x1 r y + g x2 p = h y⌝)
        (⌜∀ x • f p r x + g r p = h x⌝)
        =
        `⊢ (∀ x • f p r x + g r p = h x) ⇔
            ∀ y • f q r y + g s p = h y`
```

See Also *subst_rule*

Errors

3007	?0 is not a term variable
3012	?0 and ?1 do not have the same types
6001	?0 does not substitute to conclusion of theorem ?1
6002	Substitution theorem ?0 is not of the form: `Γ ⊢ t1 = t2`
6029	Substitution list contains entry (?0,?1) where the type of the variable differs from the type of the LHS of the theorem

SML

```
val subst_rule : (THM * TERM) list -> TERM -> THM -> THM;
```

Description Substitution of equational theorems according to a template.

Rule

$$\frac{[\Gamma 1 \vdash t1{=}t1', \ldots, \Gamma n \vdash tn{=}tn'] \quad \Gamma \vdash t[t1,\ldots tn]}{\Gamma 1 \cup \ldots \Gamma n \cup \Gamma \vdash t[t1',\ldots tn']} \quad subst\_rule$$

subst_rule [(*thm_1*, *x_1*), ..., (*thm_n*, *x_n*)] *template thm* returns a theorem in which *template* determines where in *thm* the *thm_i* are substituted. The *x_i* must be variables. The template is of the form *t*[*x_1*, ..., *x_n*], and wherever the *x_i* are free in *template* their associated equational theorem, *thm_i*, is substituted into *thm*. The rule will rename as necessary to avoid bound variable capture. The assumption list of the resulting theorem will be the union of all substitution theorems, regardless of use.

The conclusion of the resulting theorem will take its bound variable names from *template*, not *thm*, as shown in the following example. This provides an α-conversion facility.

The function may be usefully partially evaluated with one or two arguments.

A primitive inference rule.

Example

```
subst_rule [(‘⊢ p = q‘, ⌜x1⌝), (‘⊢ r = s‘, ⌜x2⌝)]
        (⌜∀ y • f x1 r y + g x2 p = h y⌝)
        (‘⊢ ∀ x • f p r x + g r p = h x‘)
        =
        ‘⊢ ∀ y • f q r y + g s p = h y‘
```

See Also *subst_conv*

Errors

3007	*?0 is not a term variable*
6001	*?0 does not substitute to conclusion of theorem ?1*
6002	*Substitution theorem ?0 is not of the form: ‘Γ ⊢ t1 = t2‘*
6029	*Substitution list contains entry (?0,?1) where the type of the variable differs from the type of the LHS of the theorem*

SML

val **SUB_C1** : *CONV* −> *CONV*;

Description Apply a conversion to each of the constituents of a term, failing if the term cannot be broken up, or the conversion fails on all constituents (if only one of the two constituents of a *mk_app* have failures, then the offending term will be *refl_conv*ed instead). Thus:

$SUB\_C1\ cnv\ var = fail\_conv\ var$

$SUB\_C1\ cnv\ const = fail\_conv\ const$

$SUB\_C1\ cnv\ (f\ x) = \Gamma \vdash f\ x = f'\ x'$
$\quad where\ cnv\ f = \Gamma 1 \vdash f = f'$
$\quad and\ \ cnv\ x = \Gamma 2 \vdash x = x'$
$\quad and\ \Gamma = \Gamma 1 \cup \Gamma 2$

$SUB\_C1\ cnv\ (\lambda\ x \bullet t) = \Gamma \vdash (\lambda\ x \bullet t) = (\lambda\ x \bullet t')$
$\quad where\ cnv\ t = \Gamma \vdash t = t'$

Errors

7104 Result of conversion, ?0, ill−formed
7105 ?0 has no constituents

There may be failure messages from the conversions.

SML

val **SUB_C** : *CONV* −> *CONV*;

Description Apply a conversion to each of the constituents of a term, however that term might be constructed, and recombine the results. Thus:

$SUB\_C\ cnv\ var = refl\_conv\ var$

$SUB\_C\ cnv\ const = refl\_conv\ const$

$SUB\_C\ cnv\ (f\ x) = \Gamma \vdash f\ x = f'\ x'$
$\quad where\ cnv\ f = \Gamma 1 \vdash f = f'$
$\quad and\ \ cnv\ x = \Gamma 2 \vdash x = x'$
$\quad and\ \Gamma = \Gamma 1 \cup \Gamma 2$

$SUB\_C\ cnv\ (\lambda\ x \bullet t) = \Gamma \vdash (\lambda\ x \bullet t) = (\lambda\ x \bullet t')$
$\quad where\ cnv\ t = \Gamma \vdash t = t'$

See Also *SUB_C1*

SML

val **suc_conv** : *CONV*;

Description This conversion gives the definition schema for non-zero natural number literals.

Rule

$$\frac{}{\vdash {}_{\mathrm{ML}}\ulcorner(mk\_\mathbb{N}(m{+}1))\urcorner = Suc\ {}_{\mathrm{ML}}\ulcorner mk\_\mathbb{N}\ m\urcorner} \quad \begin{array}{l} suc\_conv \\ (mk\_\mathbb{N}\ (m{+}1)) \end{array}$$

The conversion fails if given *0*.

Errors

3026 ?0 is not a numeric literal
7100 ?0 must be numeric literal > 0

See Also *mk_ℕ*, *prim_suc_conv*

SML

val **THEN_CAN** : (*CANON* ∗ *CANON*) −> *CANON*

Description *THEN_CAN* is a canonicalisation function combinator written as an infix operator. (*can1 THEN_CAN can2*)*thm* is the result of applying *can2* to each of the theorems in the list *can1 thm* and then flattening the resulting list of lists.

See Also *CANON*

SML

val **THEN_C** : (*CONV* ∗ *CONV*) −> *CONV*;

Description Combine the effect of two successful conversions.

Rule

$$\frac{}{\Gamma \vdash t = t'''} \quad \begin{array}{l} (c1\colon CONV)\ THEN\_C\ (c2\colon CONV) \\ \ulcorner t\urcorner \end{array}$$

where *c1 t* returns '$\Gamma 1 \vdash t = t'$', *c2 t'* returns '$\Gamma 2 \vdash t'' = t'''$', t' and t'' are α-convertible and Γ equals $\Gamma 1 \cup \Gamma 2$.

See Also *EVERY_C* (the iterated version of this function), as well as *THEN_TRY_C*, *AND-_OR_C*, and *ORELSE_C*

Errors

7101 Result of first conversion, ?0, not an equational theorem
7102 LHS (if any) of result of second conversion, ?0, not α−convertible to RHS of first, ?1

Errors If any, as the failures of *c1* and *c2* applied to *t* and *t'* respectively.

SML

val **THEN_LIST_CAN** : (*CANON* ∗ *CANON list*) −> *CANON*

Description *THEN_LIST_CAN* is a canonicalisation function combinator written as an infix operator. (*can1 THEN_LIST_CAN cans*)*thm* is the result of applying each element of the list *cans* to the corresponding element of the list *can1 thm* and then flattening the resulting list of lists.

See Also *CANON*

Errors

26204 wrong number of canonicalisation functions in the list

SML

val **THEN_TRY_C** : (*CONV* ∗ *CONV*) −> *CONV*;

Description Combine the effect of two conversions, ignoring the failure of the second if necessary. That is, if the first conversion results in an equational theorem whose RHS can have the second conversion applied, and the two resulting theorems composed, then that composition; otherwise the result of the first conversion alone is returned.

See Also *THEN_C*, *AND_OR_C*, *ORELSE_C*

Errors As the failure of *c1*.

SML

val **TOP_MAP_C** : *CONV* −> *CONV*;

Description *TOP_MAP_C conv tm* traverses *tm* from its root node to its leaves. It will repeat the application of *conv*, until failure, on each subterm encountered en route. It then descends through the sub-term that results from the repeated application of the conversion. If the descent causes any change, on "coming back out" to the sub-term the conversional will attempt to reapply *conv*, and if successful will then (recursively) reapply *TOP_MAP_C conv* once more. If *conv* cannot be reapplied then the conversional continues to ascend back to the root.

It traverses from left to right, though this should only matter for conversions that work by side-effect. It fails if the conversion is applied nowhere within the term.

Errors

7005 Conversion fails on term and all its subterms

SML

val **TRY_C** : *CONV* −> *CONV*;

Description Attempt to apply a conversion, and if it fails, apply *refl_conv*.

SML

val **t_thm** : *THM*;

Description "True" is true.

Theorem

$$\frac{}{\vdash T} \qquad t\_thm$$

SML

val **undisch_rule** : *THM* −> *THM* ;

Description Undischarge the antecedent of an implicative theorem into the assumption list.

Rule

$$\frac{\Gamma \vdash a \Rightarrow b}{\Gamma \cup \{a\} \vdash b} \qquad undisch\_rule$$

Errors

7011 ?0 is not of the form: ‘$\Gamma \vdash a \Rightarrow b$‘

SML

val **varstruct_variant** : *TERM list* −> *TERM* −> *TERM*;

Description *varstruct_variant avoid vs* will recreate the variable structure *vs* using only names that are not found in the *avoid* list of variables, and also renaming to avoid duplicate variable names in the structure. Variant names are found using *string_variant* (q.v.). If there are duplicates to be renamed, then the original name will be the rightmost in the variable structure.

Errors

3007 ?0 is not a term variable
4016 ?0 is not an allowed variable structure

Message 3007 applies to the avoid list, 27060 to the variable structure.

SML

val **v_∃_intro** : *TERM* −> *THM* −> *THM* ;

Description Introduce an existential quantified variable structure into a theorem.

Rule

$$\frac{\Gamma \vdash t[x,y,...]}{\Gamma \vdash \exists\ vs[x,y,...]\bullet\ t[x,y,...]} \qquad v\_\exists\_intro\ \ulcorner vs[x,y,...]\urcorner$$

where ⌜vs[x,y,...]⌝ is a varstruct built from variables ⌜x⌝, ⌜y⌝, etc, which may contain duplicates.

Uses If the functionality is sufficient, this is superior in efficiency to both *∃_intro* and *simple_∃_intro* (q.v.).

Errors

4016 ?0 is not an allowed variable structure

SML

val **⇔_elim** : *THM* −> (*THM* ∗ *THM*);

Description Split a bi-implicative theorem into two implicative theorems.

Rule

$$\frac{\Gamma \vdash t1 \Leftrightarrow t2}{\Gamma \vdash t1 \Rightarrow t2;\ \Gamma \vdash t2 \Rightarrow t1} \qquad \Leftrightarrow\_elim$$

Errors

7062 ?0 is not of the form: ‘Γ ⊢ t1 ⇔ t2‘

SML

val **⇔_intro** : *THM* −> *THM* −> *THM*;

Description Join two implicative theorems into an bi-implicative theorem.

Rule

$$\frac{\Gamma 1 \vdash t1 \Rightarrow t2;\ \Gamma 2 \vdash t1' \Rightarrow t2'}{\Gamma 1 \cup \Gamma 2 \vdash t1 \Leftrightarrow t2} \qquad \Leftrightarrow\_intro$$

where *t1* and *t1′* are α-convertible, as are *t2* and *t2′*.

Errors

7040 ?0 is not of the form: ‘Γ ⊢ t1 ⇒ t2‘
7064 ?0 and ?1 are not of the form: ‘Γ ⊢ t1 ⇒ t2; Γ ⊢ t2a ⇒ t1a‘ where ⌜t1⌝ and ⌜t1a⌝, ⌜t2⌝ and ⌜t2a⌝, are α−convertible

SML

```
val ⇔_match_mp_rule : THM -> THM -> THM;
```

Description A matching Modus Ponens for ⇔.

Rule

$$\frac{\Gamma 1 \vdash \forall\ x1\ ...\bullet\ t1 \Leftrightarrow t2;\ \Gamma 2 \vdash t1'}{\Gamma 1' \cup \Gamma 2 \vdash t2'} \quad \Leftrightarrow\_match\_mp\_rule$$

where we type instantiate, generalise and specialise both conclusion and assumptions to get the first theorem's LHS to match the conclusion of the second theorem. Universal quantification, or the lack of it, in the first theorem makes no difference to the matching.

This may be partially evaluated with only first argument.

See Also *⇒_elim* (Modus Ponens on ⇒), *simple_⇔_match_mp_rule* *⇔_mp_rule* *⇔_match-_mp_rule1*

Errors

7044 Cannot match ?0 and ?1

SML

```
val ⇔_match_mp_rule1 : THM -> THM -> THM;
```

Description A matching Modus Ponens for ⇔ that doesn't affect assumption lists.

Rule

$$\frac{\Gamma 1 \vdash \forall\ x1\ ...\bullet\ t1 \Leftrightarrow t2;\ \Gamma 2 \vdash t1'}{\Gamma 1 \cup \Gamma 2 \vdash t2'} \quad \Leftrightarrow\_match\_mp\_rule1$$

where $t1'$ is an instance of $t1$ under type instantiation and substitution for the x_i and the free variables of the first theorem, and where $t2'$ is the corresponding instance of $t2$. No type instantiation or substitution will occur in the assumptions of either theorem.

This may be partially evaluated with only first argument.

See Also *⇒_elim* (Modus Ponens on ⇒), *simple_⇔_match_mp_rule1*

Errors

7044 Cannot match ?0 and ?1
7046 ?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u ⇔ v‘

SML

```
val ⇔_mp_rule : THM -> THM -> THM;
```

Description This is reminiscent of Modus Ponens, but upon bi-implicative theorems.

Rule

$$\frac{\Gamma 1 \vdash t1 \Leftrightarrow t2;\ \Gamma 2 \vdash t1'}{\Gamma 1 \cup \Gamma 2 \vdash t2} \quad \Leftrightarrow\_mp\_rule$$

where $t1$ and $t1'$ must be α-convertible.

A built-in inference rule.

See Also *⇒_elim* (true Modus Ponens, on ⇒), *⇔_match_mp_rule* (a "matching" version of *⇔_mp_rule*)

Errors

6024 ?0 and ?1 are not of the form: ‘Γ1 ⊢ t1 ⇔ t2‘ and ‘Γ2 ⊢ t1′‘
where ⌜t1⌝ and ⌜t1′⌝ are α−convertible
6030 ?0 is not of the form: ‘Γ ⊢ t1 ⇔ t2‘

SML
$val\ \Leftrightarrow$_**t_elim** : $THM\ {-}{>}\ THM;$

Description We can always eliminate $\ldots \Leftrightarrow T$.

Rule

$$\frac{\Gamma \vdash t \Leftrightarrow T}{\Gamma \vdash t} \qquad \Leftrightarrow\_t\_elim$$

Errors
7106 ?0 not of the form '$\Gamma \vdash t \Leftrightarrow T$'

SML
$val\ \Leftrightarrow$_**t_intro** : $THM\ {-}{>}\ THM;$

Description The conclusion of a theorem is equal to T.

Rule

$$\frac{\Gamma \vdash t}{\Gamma \vdash t \Leftrightarrow T} \qquad \Leftrightarrow\_t\_intro$$

SML
$val\ \wedge$_**intro** : $THM\ {-}{>}\ THM\ {-}{>}\ THM;$

Description Conjoin two theorems.

Rule

$$\frac{\Gamma 1 \vdash t1;\ \Gamma 2 \vdash t2}{\Gamma 1 \cup \Gamma 2 \vdash t1 \wedge t2} \qquad \wedge\_intro$$

SML
$val\ \wedge$_**left_elim** : $THM\ {-}{>}\ THM;$

Description Give the left conjunct of a conjunction.

Rule

$$\frac{\Gamma \vdash t1 \wedge t2}{\Gamma \vdash t1} \qquad \wedge\_left\_elim$$

Errors
7007 ?0 is not of the form: '$\Gamma \vdash t1 \wedge t2$'

SML
```
val ∧_rewrite_canon : THM -> THM list
val simple_¬_rewrite_canon : THM -> THM list
val ⇔_t_rewrite_canon : THM -> THM list
val f_rewrite_canon : THM -> THM list
val simple_∀_rewrite_canon : THM -> THM list
```

Description These are some of the standard canonicalisation functions used for breaking theorems up into lists of equations for use in rewriting. They perform the following transformations:

```
∧_rewrite_canon              (Γ ⊢ t1 ∧ t2)        = Γ ⊢ t1 ; Γ ⊢ t2
simple_¬_rewrite_canon       (Γ ⊢ ¬(t1 ∨ t2))     = (Γ ⊢ ¬t1 ∧ ¬t2)
simple_¬_rewrite_canon       (Γ ⊢ ¬∃x•t)          = (Γ ⊢ ∀x•¬t)
simple_¬_rewrite_canon       (Γ ⊢ ¬ ¬t)           = (Γ ⊢ t)
simple_¬_rewrite_canon       (Γ ⊢ ¬t)             = (Γ ⊢ t ⇔ F)
⇔_t_rewrite_canon        (Γ ⊢ t1 = t2)        = < failure >
⇔_t_rewrite_canon        (Γ ⊢ t)                  = (Γ ⊢ t ⇔ T)
f_rewrite_canon              (Γ ⊢ F)                  = (Γ ⊢ ∀ x • x)
simple_∀_rewrite_canon       (Γ ⊢ ∀x•t)           = Γ ⊢ t
```

Note that the functions whose names begin with *simple* do not handle paired quantifiers. Versions which do handle these quantifiers are also available.

See Also *¬_rewrite_canon*, *∀_rewrite_canon*.

Errors
```
26203 the conclusion of the theorem is already an equation
```

SML
```
val ∧_right_elim : THM -> THM;
```

Description Give the right conjunct of a conjunction.

Rule

$$\frac{\Gamma \vdash t1 \land t2}{\Gamma \vdash t2} \quad \land\_right\_elim$$

Errors
```
7007 ?0 is not of the form: ‘Γ ⊢ t1 ∧ t2‘
```

SML
```
val ∧_thm : THM;
```

Description Expanded form of definition of ∧

Theorem

$$\frac{}{\forall\ t1\ t2 \bullet (t1 \land t2) \Leftrightarrow (\forall\ b \bullet (t1 \Rightarrow t2 \Rightarrow b) \Rightarrow b)} \quad \land\_thm$$

SML
```
val ∧_⇒_rule : THM -> THM;
```

Description A theorem whose conclusion is an implication from a conjunction is an equivalent to one whose conclusion is an implication of an implication.

Rule

$$\frac{\Gamma \vdash (a \land b) \Rightarrow c}{\Gamma \vdash a \Rightarrow b \Rightarrow c} \quad \land\_\Rightarrow\_rule$$

Errors
```
7009 ?0 is not of the form: ‘Γ ⊢ (a ∧ b) ⇒ c‘
```

SML

$val\ \vee$_**cancel_rule** : $THM\ {-}{>}\ THM\ {-}{>}\ THM;$

Description If we know a disjunction is true, and one of its disjuncts is false, then the other must be true. If the second theorem is the negation of both disjuncts, then the second disjunct will be eliminated. (modus tollendo ponens)

Rule

$$\frac{\Gamma 1 \vdash t1 \vee t2;\ \Gamma 2 \vdash \neg t1'}{\Gamma 1 \cup \Gamma 2 \vdash t2} \quad \vee\_cancel\_rule$$

And:

Rule

$$\frac{\Gamma 1 \vdash t1 \vee t2;\ \Gamma 2 \vdash \neg t2'}{\Gamma 1 \cup \Gamma 2 \vdash t1} \quad \vee\_cancel\_rule$$

where $t1'$ and $t1$ are α-convertible, as are $t2$ and $t2'$.

Errors

7010 *?0 is not of the form:* '$\Gamma \vdash t1 \vee t2$'

7050 *?0 and ?1 are not of the form:* '$\Gamma 1 \vdash t1 \vee t2$' *and* '$\Gamma 2 \vdash \neg t3$' *where* ⌜$t3$⌝ *is* $\alpha-$*convertible to* ⌜$t1$⌝ *or* ⌜$t2$⌝

SML

$val\ \vee$_**elim** : $THM\ {-}{>}\ THM\ {-}{>}\ THM\ {-}{>}\ THM;$

Description Given a disjunctive theorem, and two further theorems, each containing one of the disjuncts in their assumptions, but with the same conclusion, we may eliminate the disjunct assumption from the second of the theorems.

Rule

$$\frac{\begin{array}{c}\Gamma 1 \vdash t1 \vee t2\\ \Gamma 2,\ t1' \vdash t\\ \Gamma 3,\ t2' \vdash t'\end{array}}{\Gamma 1 \cup \Gamma 2 \cup \Gamma 3 \vdash t} \quad \vee\_elim$$

where $t1$ and $t1'$ are α-convertible, as are $t2$ and $t2'$, and t and t'. Actually, $t1'$ and $t2'$ do not have to be present in the assumption lists for this function to work.

Errors

7010 *?0 is not of the form:* '$\Gamma \vdash t1 \vee t2$'

7083 *?0, ?1 and ?2 are not of the form:* '$\Gamma 1 \vdash t1 \vee t2$', '$\Gamma 2,\ t1a \vdash t3$' *and* '$\Gamma 3,\ t2a \vdash t3a$', *where* ⌜$t1$⌝ *and* ⌜$t1a$⌝, ⌜$t2$⌝ *and* ⌜$t2a$⌝, ⌜$t3$⌝ *and* ⌜$t3a$⌝ *are each* $\alpha-$*convertible*

SML

$val\ \vee$_**left_intro** : $TERM\ {-}{>}\ THM\ {-}{>}\ THM;$

Description Introduce a disjunct to the left of a theorem's conclusion.

Rule

$$\frac{\Gamma \vdash b}{\Gamma \vdash a \vee b} \quad \begin{array}{l}\vee\_left\_intro\\ \ulcorner a \urcorner\end{array}$$

Errors

3031 *?0 is not of type* ⌜:$BOOL$⌝

SML

val ∨_**right_intro** : *TERM −> THM −> THM*;

Description Introduce a disjunct to the right of a theorem's conclusion.

Rule

$$\frac{\Gamma \vdash b}{\Gamma \vdash b \lor a} \quad \lor\_right\_intro \ \ulcorner a \urcorner$$

Errors

3031 ?0 is not of type ⌜:BOOL⌝

SML

val ∨_**thm** : *THM*;

Description Expanded form of definition of ∨

Theorem

$$\frac{}{\forall\ t1\ t2 \bullet (t1 \lor t2) \Leftrightarrow (\forall\ b \bullet (t1 \Rightarrow b) \Rightarrow (t2 \Rightarrow b) \Rightarrow b)} \quad \lor\_thm$$

SML

val ¬_**elim** : *TERM −> THM −> THM −> THM*;

Description Given two contradictory theorems with the same assumptions, conclude any other fact from the assumptions: input theorems may be in either order.

Rule

$$\frac{\Gamma 1 \vdash a\ ;\ \Gamma 2 \vdash \neg\ a}{\Gamma 1 \cup \Gamma 2 \vdash b} \quad \neg\_elim \ \ulcorner b \urcorner$$

Errors

3031 ?0 is not of type ⌜:BOOL⌝
7004 ?0 and ?1 are not of the form: 'Γ1 ⊢ a' and 'Γ2 ⊢ ¬ a'

SML

val ¬_**eq_sym_rule** : *THM −> THM* ;

Description If a is not equal to b then b is not equal to a.

Rule

$$\frac{\Gamma \vdash \neg(a = b)}{\Gamma \vdash \neg(b = a)} \quad \neg\_eq\_sym\_rule$$

Errors

7091 ?0 is not of form: 'Γ ⊢ ¬(a = b)'

SML

val ¬_**intro** : *TERM −> THM −> THM −> THM*;

Description Given two theorems with contradictory conclusions (up to α-convertibility), their assumptions must be inconsistent, and thus any member of the lists (or indeed, anything else) may be proven false on the assumption of the remainder (reductio ad absurdum).

Rule

$$\frac{\Gamma 1 \vdash b\ ;\ \Gamma 2 \vdash \neg\ b}{(\Gamma 1 \cup \Gamma 2) \setminus \{a\} \vdash \neg\ a} \quad \neg\_intro \ \ulcorner a \urcorner$$

Works up to α-conversion, and input theorems may be in either order.

Errors

3031 ?0 is not of type ⌜:BOOL⌝
7004 ?0 and ?1 are not of the form: 'Γ1 ⊢ a' and 'Γ2 ⊢ ¬ a'

SML

val **¬_simple_∀_conv** : *CONV*;

Description Move ¬ into a ∀ construct.

Rule

$$\frac{}{\vdash (\neg\ (\forall\ x \bullet t[x])) \Leftrightarrow \exists\ x \bullet \neg\ t[x]}$$ *¬_simple_∀_conv* ⌜¬ (∀ x • t[x])⌝

This will work with any simple universal quantifier.

Errors

7036 ?0 not of the form: ⌜¬ (∀ x • t[x])⌝

SML

val **¬_simple_∃_conv** : *CONV*;

Description Move ¬ into an ∃ construct.

Rule

$$\frac{}{\vdash (\neg\ (\exists\ x \bullet t[x])) \Leftrightarrow \forall\ x \bullet \neg\ t[x]}$$ *¬_simple_∃_conv* ⌜¬ (∃ x • t[x])⌝

This will work with any simple existential quantifier.

Errors

7058 ?0 is not of the form: ⌜¬ (∃ x • t[x])⌝
where ⌜x⌝ is a variable

SML

val **¬_thm1** : *THM*;

Description "Not t if and only if t is false."

Theorem

$$\frac{}{\vdash \forall\ t \bullet (\neg\ t) \Leftrightarrow (t \Leftrightarrow F)}$$ *¬_thm1*

SML

val **¬_thm** : *THM*;

Description Expanded form of definition of ¬:

Theorem

$$\frac{}{\forall\ t \bullet (\neg\ t) \Leftrightarrow (t \Rightarrow F)}$$ *¬_thm*

SML

val **¬_t_thm** : *THM*;

Description "Not true is false".

Theorem

$$\frac{}{\neg\ T \Leftrightarrow F}$$ *¬_t_thm*

SML

val **¬_¬_conv** : *CONV*;

Description A double negation is redundant.

Conversion

$$\frac{}{\Gamma \vdash \neg\ (\neg\ t) \Leftrightarrow t}$$ *¬_¬_conv* ⌜¬ (¬ t)⌝

Errors

7022 ?0 is not of the form:⌜¬ (¬ t)⌝

SML
```
val ¬_¬_elim : THM -> THM;
```

Description A double negation is redundant.

Rule

$$\frac{\Gamma \vdash \neg\ (\neg\ t)}{\Gamma \vdash t} \quad \neg\_\neg\_elim$$

Errors

7006 ?0 is not of the form: ‘Γ ⊢ ¬ (¬ t)‘

SML
```
val ¬_¬_intro : THM -> THM;
```

Description We may always introduce a double negation.

Rule

$$\frac{\Gamma \vdash t}{\Gamma \vdash \neg\ (\neg\ t)} \quad \neg\_\neg\_intro$$

SML
```
val ¬_∀_conv : CONV;
```

Description Move ¬ into a ∀ construct.

Rule

$$\frac{}{\vdash (\neg\ (\forall\ x\bullet\ t[x])) \Leftrightarrow \exists\ x\bullet\ \neg\ t[x]} \quad \begin{array}{l}\neg\_\forall\_conv \\ \ulcorner\neg\ (\forall\ x\bullet\ t[x])\urcorner\end{array}$$

See Also *¬_simple_∀_conv* which only works with simple ∀-abstractions, *¬_∃_conv*

Errors

27019 ?0 not of the form: ⌜¬ (∀ x• t[x])⌝
where ⌜x⌝ is a varstruct

SML
```
val ¬_∀_thm : THM;
```

Description Used in pushing negations through simple universal quantifications.

Theorem

$$\frac{}{\vdash \forall\ p\bullet\ \neg\ \$\forall\ p \Leftrightarrow (\exists\ x\bullet\ \neg\ p\ x)} \quad \neg\_\forall\_thm$$

SML
```
val ¬_∃_conv : CONV;
```

Description Move ¬ into an ∃ construct.

Rule

$$\frac{}{\vdash (\neg\ (\exists\ x\bullet\ t[x])) \Leftrightarrow \forall\ x\bullet\ \neg\ t[x]} \quad \begin{array}{l}\neg\_\exists\_conv \\ \ulcorner\neg\ (\exists\ x\bullet\ t[x])\urcorner\end{array}$$

See Also *¬_simple_∃_conv* which only works with simple ∃-abstractions, *¬_∀_conv*

Errors

27020 ?0 is not of the form: ⌜¬ (∃ x• t[x])⌝
where ⌜x⌝ is a varstruct

SML

val ¬_∃_**thm** : *THM*;

Description Used in pushing negations through simple existential quantifications.

Theorem

$$\frac{}{\vdash \forall\ p\bullet\ \neg\ \$\exists\ p \Leftrightarrow (\forall\ x\bullet\ \neg\ p\ x)} \quad \neg\_\exists\_thm$$

SML

val ⇒_**elim** : *THM* −> *THM* −> *THM*;
val ⇒_**mp_rule** : *THM* −> *THM* −> *THM*;

Description Modus Ponens (which is why we introduce the alias *⇒_mp_rule*, though *⇒_elim* is shorter, conventional, and the preferred name).

Rule

$$\frac{\Gamma 1 \vdash t1 \Rightarrow t2;\ \Gamma 2 \vdash t1'}{\Gamma 1 \cup \Gamma 2 \vdash t2} \quad \Rightarrow\_elim$$

where *t1* and *t1′* must be α-convertible. A primitive inference rule.

See Also *⇔_mp_rule*(Modus Ponens on ⇔), *⇒_match_mp_rule* (a "matching" version of this function).

Errors

6010 ?0 is not of the form: ‘Γ ⊢ t1 ⇒ t2‘
6011 ?0 and ?1 are not of the forms: ‘Γ1 ⊢ t1 ⇒ t2‘ and ‘Γ2 ⊢ t1′‘ where ⌜t1⌝ and ⌜t1′⌝ are α−convertible

SML

val ⇒_**intro** : *TERM* −> *THM* −> *THM*;

Description Prove an implicative theorem, removing, if α-convertibly present, the antecedent of the implication from the assumption list.

Rule

$$\frac{\Gamma \vdash t2}{\Gamma - \{t1\} \vdash t1 \Rightarrow t2} \quad \begin{array}{l}\Rightarrow\_intro\\ \ulcorner t1 \urcorner\end{array}$$

A primitive inference rule.

See Also *disch_rule* (which fails if term not in assumption list)

Errors

3031 ?0 is not of type ⌜:BOOL⌝

SML

$val\ \Rightarrow$**_match_mp_rule** : $THM\ {->}\ THM\ {->}\ THM$;

Description A matching Modus Ponens rule for an implicative theorem.

Rule

$$\frac{\Gamma 1 \vdash \forall\ x1\ ...\bullet\ t1 \Rightarrow t2;\ \Gamma 2 \vdash t1'}{\Gamma 1' \cup \Gamma 2 \vdash t2'} \qquad \Rightarrow\_match\_mp\_rule$$

where we type instantiate, generalise and specialise to get the first theorem's antecedent to match the conclusion of the second theorem. Universal quantification, or the lack of it, in the first theorem makes no difference to the matching.

This may be partially evaluated with only the first argument.

See Also *⇒_match_mp_rule1*, *⇒_elim*

Errors

7044	*Cannot match ?0 and ?1*

SML

$val\ \Rightarrow$**_match_mp_rule1** : $THM\ {->}\ THM\ {->}\ THM$;
$val\ \Rightarrow$**_match_mp_rule2** : $THM\ {->}\ THM\ {->}\ THM$;

Description Two variants of a matching Modus Ponens rule for an implicative theorem.

Rule

$$\frac{\Gamma 1 \vdash \forall\ x1\ ...\bullet\ t1 \Rightarrow t2;\ \Gamma 2 \vdash t1'}{\Gamma 1 \cup \Gamma 2 \vdash t2'} \qquad \Rightarrow\_match\_mp\_rule$$

where $t1'$ is an instance of $t1$ under type instantiation and substitution for the x_i and the free variables of the first theorem, and where $t2'$ is the corresponding instance of $t2$. The type instantiations and substitutions are obtained by matching $t1$ and $t1'$ using *term_match*.

⇒_match_mp_rule2 is just like *⇒_match_mp_rule1* except that the instantiations and substitutions returned by *term_match* are extended to replace type variables that do not occur in $t1$ or in $\Gamma 1$ and *x_i* that do not occur free in $t1$ by fresh variables to avoid clashes with each other and with the type variables and free variables of $\Gamma 1$ and $\Gamma 2$.

Types in the assumptions of the theorems will not be instantiated.

Both rules may be partially evaluated with only the first argument.

Errors

7044	*Cannot match ?0 and ?1*
7045	*?0 is not of the form '$\Gamma \vdash \forall\ x1\ ...\ xn \bullet u \Rightarrow v$'*

SML

$val\ \Rightarrow$**_trans_rule** : $THM\ {->}\ THM\ {->}\ THM$;

Description Transitivity of ⇒.

Rule

$$\frac{\Gamma 1 \vdash t1 \Rightarrow t2;\ \Gamma 2 \vdash t2' \Rightarrow t3}{\Gamma 1 \cup \Gamma 2 \vdash t1 \Rightarrow t3} \qquad \Rightarrow\_trans\_rule$$

where $t2$ and $t2'$ are α-convertible.

Errors

7040	*?0 is not of the form: '$\Gamma \vdash t1 \Rightarrow t2$'*
7042	*?0 and ?1 are not of the form: '$\Gamma 1 \vdash t1 \Rightarrow t2$' and '$\Gamma 2 \vdash t2a \Rightarrow t3$' where ⌜t2⌝ and ⌜t2a⌝ are α−convertible*

SML

$val\ \Rightarrow\_\wedge\_$**rule** : $THM\ {-}{>}\ THM;$

Description A theorem whose conclusion is an implication of an implication is equivalent to one whose conclusion is a conjunction and an implication.

Rule

$$\frac{\Gamma \vdash a \Rightarrow b \Rightarrow c}{\Gamma \vdash (a \wedge b) \Rightarrow c} \quad \Rightarrow\_\wedge\_rule$$

Errors

7008 ?0 is not of the form: ‘$\Gamma \vdash a \Rightarrow b \Rightarrow c$‘

SML

$val\ \forall\_$**arb_elim** : $THM\ {-}{>}\ THM;$

Description Specialise a universally quantified theorem with a machine generated variable or variable structure.

Rule

$$\frac{\Gamma \vdash \forall\ vs[x,y,...]\bullet\ p[x,y,...]}{\Gamma \vdash p[x',y',...]} \quad \forall\_arb\_elim$$

where x', y', etc, are not variables (free or bound) in p or Γ, created by *gen_vars*(q.v).

See Also $\forall\_elim$

Errors

27011 ?0 is not of the form: ‘$\Gamma \vdash \forall\ x\bullet\ t$‘ *where* $\ulcorner x \urcorner$ *is a varstruct*

SML

$val\ \forall\_$**asm_rule** : $TERM\ {-}{>}\ TERM\ {-}{>}\ THM\ {-}{>}\ THM;$

Description Generalise an assumption (Left $\forall$ introduction).

Rule

$$\frac{\Gamma,\ p'[x] \vdash q[x]}{\Gamma,\ \forall\ x\bullet\ p'[x] \vdash q[x]} \quad \begin{array}{l}\forall\_asm\_rule\\ \ulcorner x \urcorner\\ \ulcorner p[x] \urcorner\end{array}$$

where p and p' are α-convertible. x **may** be free in Γ. The function will work even if $p'[x]$ is not present in the assumption list.

Errors

4016 ?0 is not an allowed variable structure

SML

val ∀_**elim** : *TERM* −> *THM* −> *THM*;

Description Specialise a universally quantified theorem with a given value, instantiating the type of the theorem as necessary.

Rule

$$\frac{\Gamma \vdash \forall\ x \bullet\ t2[x]}{\Gamma \vdash\ t2'[t1]} \qquad \begin{matrix}\forall\_elim \\ \ulcorner t1 \urcorner\end{matrix}$$

where $t2'$ is renamed from $t2$ to prevent bound variable capture and possibly type instantiated, and x is a varstruct, instantiable to the structure of $t1$. The value $t1$ will be expanded using *Fst* and *Snd* as necessary to match the structure of $\ulcorner x \urcorner$.

See Also *list_∀_elim*, *all_∀_elim*.

Errors

27011 ?0 is not of the form: ‘Γ ⊢ ∀ x• t‘ where ⌜x⌝ is a varstruct
27012 ?0 is not of the form: ‘Γ ⊢ ∀ x• t‘ where the type of ⌜x⌝
is instantiable to the type of ?1
27013 ?0 is not of the form: ‘Γ ⊢ ∀ x• t‘ where the type of ⌜x⌝
is instantiable to the type of ?1 without instantiating
type variables in the assumptions

SML

val ∀_**intro** : *TERM* −> *THM* −> *THM*;

Description Introduce a universally quantified theorem.

Rule

$$\frac{\Gamma \vdash\ t}{\Gamma \vdash \forall\ x' \bullet\ t} \qquad \begin{matrix}\forall\_intro \\ \ulcorner x \urcorner\end{matrix}$$

Where $\ulcorner x' \urcorner$ is an allowed variable structure based on $\ulcorner x \urcorner$, but with duplicate variables renamed, the original name being rightmost in the resulting variable structure.

See Also *list_∀_intro*, *all_∀_intro*.

Errors

4016 ?0 is not an allowed variable structure
6005 ?0 occurs free in assumption list

SML
```
val ∀_reorder_conv : TERM -> CONV;
```

Description Reorder universal quantifications.

Rule

$$\frac{}{(\forall\ y1\ ..\ ym\bullet\ t2) \Leftrightarrow (\forall\ x1\ ..\ xn\bullet\ t1)} \quad \begin{array}{l}\forall\_reorder\_conv\\ \ulcorner\forall\ x1\ ..\ xn\bullet\ t1\urcorner\\ \ulcorner\forall\ y1\ ..\ ym\bullet\ t2\urcorner\end{array}$$

where the x_i and y_i are varstructs, and the reordering, restructuring (by pairing) and renaming requested is provable by this function. The presence of redundant quantifiers, including duplicates, is also handled.

Example
```
:> ∀_reorder_conv ⌜∀ (x,q) z• x ∧ z⌝ ⌜∀ (z,z,y) x• x ∧ z⌝;
val it = ⊢ (∀ (z, z, y) x• x ∧ z) ⇔ (∀ (x, q) z• x ∧ z) : THM
Note that before more sophisticated attempts, the conversion
will try $α\_conv$ on the two term arguments.
```

See Also *∃_reorder_conv*

Errors
```
27050 Cannot prove equality of ?0 and ?1
```

SML
```
val ∀_uncurry_conv : CONV;
```

Description Convert a paired universally quantified term into simple universal quantifications of the same term.

Conversion

$$\frac{}{\Gamma \vdash \forall\ vs[x,y,...]\bullet\ f[x,y,...] = \forall\ x\ y\ ...\bullet\ f[x,y,...]} \quad \begin{array}{l}\forall\_uncurry\_conv\\ \ulcorner\forall\ vs[x,y,...]\bullet\ f[x,y,...]\urcorner\end{array}$$

where $vs[x, y, ...]$ is an allowed variable structure with variables $x, y, ...$ It may not be a simple variable.

See Also *λ_varstruct_conv*, *all_∀_uncurry_conv*.

Errors
```
27038 ?0 is not of the form: ⌜∀ (x,y)• f⌝
```

SML
```
val ∀_⇔_rule : TERM -> THM -> THM;
```

Description Universally quantify a variable on both sides of an equivalence.

Rule

$$\frac{\Gamma \vdash p[x] \Leftrightarrow q[x]}{\Gamma \vdash (\forall\ x\bullet\ p[x]) \Leftrightarrow (\forall\ x\bullet\ q[x])} \quad \begin{array}{l}\forall\_\Leftrightarrow\_rule\\ \ulcorner x\urcorner\end{array}$$

where x is a varstruct.

Errors
```
6005 ?0 occurs free in assumption list
6020 ?0 is not of the form: ‘Γ ⊢ t1 = t2‘
7062 ?0 is not of the form: ‘Γ ⊢ t1 ⇔ t2‘
4016 ?0 is not an allowed variable structure
```

SML

val ∃_**asm_rule** : *TERM* −> *TERM* −> *THM* −> *THM*;

Description Existentially quantify an assumption (Left ∃ introduction).

Rule

$$\frac{\Gamma,\ p'[x] \vdash q}{\Gamma,\ \exists\ x\bullet\ p'[x] \vdash q} \quad \begin{array}{l}\exists\_asm\_rule\\ \ulcorner x \urcorner\\ \ulcorner p[x] \urcorner\end{array}$$

where p and p' are α-convertible. where the variables of the varstruct x are not free in Γ or q. The assumption need not be present for the rule to apply.

Errors

3015 ?1 is not of type ⌜:BOOL⌝
4016 ?0 is not an allowed variable structure
6005 ?0 occurs free in assumption list
27052 ?0 has members appearing free in ?1 other than in assumption ?2

Message 3015 is just passed on from low level functions, which is why it has "?1" not "?0".

SML

val ∃_**elim** : *TERM* −> *THM* −> *THM* −> *THM*;

Description Eliminate an existential quantifier by reference to an arbitrary varstruct satisfying the predicate.

Rule

$$\frac{\Gamma 1 \vdash \exists\ vs[x1,x2,...]\bullet\ t1[x1,x2,...];\quad \Gamma 2,\ t1[y1,y2,...] \vdash t2}{\Gamma 1 \cup \Gamma 2 \vdash t2} \quad \begin{array}{l}\exists\_elim\\ \ulcorner vs[y1,y2,...] \urcorner\end{array}$$

$t1[y1, y2, ...]$ need not actually be present in the assumptions of the second theorem. The y_i must be free variables, none of whom are present elsewhere in the second theorem, or in the conclusion of the first. The y_i may contain duplicates as long as the end pattern matches the x_i in required duplicates. The term argument may be a less complex variable structure than the bound variable structure of the theorem, as *Fst* and *Snd* are used to make them match. For example, the following rule holds true:

Rule

$$\frac{\Gamma 1 \vdash \exists\ (p,q)\bullet\ t1[p,q];\quad \Gamma 2,\ t1[Fst\ x,\ Snd\ x] \vdash t2}{\Gamma 1 \cup \Gamma 2 \vdash t2} \quad \begin{array}{l}\exists\_elim\\ \ulcorner x \urcorner\end{array}$$

Errors

27042 ?0 does not match the bound varstruct of ?1
27046 ?0 is not of the form ‘Γ ⊢ ∃ vs• t‘
27051 ?0 has members appearing free in conclusion of ?1
27052 ?0 has members appearing free in ?1 other than in assumption ?2

SML

val ∃_**intro_thm** : *THM*;

Description Introduction of existential quantification.

Theorem

$$\frac{}{\vdash \forall\ P\ x\bullet\ P\ x \Rightarrow \$\exists\ P} \quad \exists\_intro\_thm$$

SML
```
val ∃_intro : TERM -> THM -> THM ;
```

Description Introduce an existential quantifier by reference to a witness.

Rule

$$\frac{\Gamma \vdash t[t1,t2,...]}{\Gamma \vdash \exists\ vs[x',y',...]\bullet\ t[x,y,...]} \qquad \begin{array}{l}\exists\_intro\\ \ulcorner\exists\ vs[x,y,...]\bullet\ t[x,y,...]\urcorner\end{array}$$

where ⌜vs[x,y,...]⌝ is varstruct built from variables ⌜x⌝, ⌜y⌝, etc, and the ⌜x'⌝ are renamed if duplicated inside the varstruct, all but the rightmost being so renamed.

Errors

4020 *?0 is not of form:* ⌜∃ vs• t⌝
7047 *?0 cannot be matched to conclusion of theorem ?1*

SML
```
val ∃_reorder_conv : TERM -> CONV;
```

Description Reorder existential quantifications.

Rule

$$\frac{}{(\exists\ y1\ ..\ ym\bullet\ t2) \Leftrightarrow (\exists\ x1\ ..\ xn\bullet\ t1)} \qquad \begin{array}{l}\exists\_reorder\_conv\\ \ulcorner\exists\ x1\ ..\ xn\bullet\ t1\urcorner\\ \ulcorner\exists\ y1\ ..\ ym\bullet\ t2\urcorner\end{array}$$

where the x_i and y_i are varstructs, and the reordering, restructuring (by pairing) and renaming requested is provable by this function. The presence of redundant quantifiers, including duplicates, is also handled.

Example
```
:> ∃_reorder_conv ⌜∃ (x,q) z• x ∧ z⌝ ⌜∃ (z,z,y) x• x ∧ z⌝;
val it = ⊢ (∃ (z, z, y) x• x ∧ z) ⇔ (∃ (x, q) z• x ∧ z) : THM
Note that before more sophisticated attempts, the conversion
will try $α\_conv$ on the two term arguments.
```

See Also *∀_reorder_conv*

Errors

27050 Cannot prove equality of ?0 and ?1

SML
```
val ∃_uncurry_conv : CONV;
```

Description Convert a paired existentially quantified term into simple universal quantifications of the same term.

Conversion

$$\frac{}{\begin{array}{c}\vdash \exists\ vs[x,y,...]\bullet\ f[x,y,...] =\\ \exists\ x\ y\ ...\bullet\ f[x,y,...]\end{array}} \qquad \begin{array}{l}\exists\_uncurry\_conv\\ \ulcorner\exists\ vs[x,y,...]\bullet\ f[x,\ y,...]\urcorner\end{array}$$

where $vs[x, y, ...]$ is an allowed variable structure with variables $x, y, ...$ It may not be a simple variable.

See Also *λ_varstruct_conv*, *all_∃_uncurry_conv*, *∀_uncurry_conv*.

Errors

27047 ?0 is not of the form: ⌜∃ (x,y)• f⌝

SML

```
val ∃_ϵ_conv : CONV;
```

Description Give that ϵ of a predicate satisfies the predicate by reference to an $\exists$ construct. It can properly handle paired existence.

Rule

$$\frac{}{\Gamma \vdash (\exists\ x\bullet\ p[x]) = p(\epsilon\ x\bullet\ p\ x)} \quad \begin{array}{l} \exists\_\epsilon\_conv \\ \ulcorner \exists\ x\bullet\ p[x] \urcorner \end{array}$$

If x is formed by paired then the *Fst* and *Snd* are used to extract the appropriate bits of the ϵ-term for distribution in $p[\epsilon\ x\bullet\ p\ x]$.

See Also *∃_ϵ_rule*

Errors

27024 ?0 is not of the form: ‘Γ ⊢ ∃ x• p[x]‘
where ⌜x⌝ is a varstruct

SML

```
val ∃_ϵ_rule : THM -> THM;
```

Description Give that ϵ of a predicate satisfies the predicate by reference to an $\exists$ construct. It can properly handle paired existence.

Rule

$$\frac{\Gamma \vdash \exists\ x\bullet\ p[x]}{\Gamma \vdash p[\epsilon\ x\bullet\ p\ x]} \quad \exists\_\epsilon\_rule$$

If x is formed by paired then the *Fst* and *Snd* are used to extract the appropriate bits of the ϵ-term for distribution in $p[\epsilon\ x\bullet\ p\ x]$.

See Also *∃_ϵ_conv*

Errors

27024 ?0 is not of the form: ‘Γ ⊢ ∃ x• p[x]‘
where ⌜x⌝ is a varstruct

SML

```
val ∃₁_conv : CONV;
```

Description This is a conversion which turns a unique existential quantifier into an equivalent existential quantifier

Conversion

$$\frac{}{\begin{array}{l} \vdash (\exists_1 vs[x1,...]\bullet t[x1,...]) \Leftrightarrow \\ \quad (\exists vs[x1,...]\bullet t[x1,...] \land \\ \quad \forall vs[x1',...]\bullet t[x1',...] \Rightarrow \\ \quad vs[x1',...] = vs[x1,...]) \end{array}} \quad \begin{array}{l} \exists_1\_conv \\ \ulcorner \exists_1 vs[x1,...]\bullet t[x1,...] \urcorner \end{array}$$

Uses Tactic and conversion programming.

See Also *strip_tac*, *simple_∃_1_conv*

Errors

27053 ?0 is not of the form: ⌜$\exists_1$ vs•t⌝

SML

$val\ \exists_1\text{\_}\mathbf{elim} : THM \;{-}{>}\; THM;$

Description Express a ∃_1 in terms of ∃ and a uniqueness property.

Rule

$$\frac{\Gamma \vdash \exists_1\ vs[a,b,...] \bullet P[a,b,...]}{\begin{array}{c}\Gamma \vdash \exists\ vs[a,b,...] \bullet P[a,b,...] \wedge \\ \forall\ vs[a',b',...] \bullet P[x1,x2,...] \\ \Rightarrow \\ vs[a',b',...] = vs[a,b,...]\end{array}} \quad \exists_1\text{\_}elim$$

where the a', etc, are variants of the a.

Errors

27022 ?0 is not of the form: ‘Γ ⊢ ∃₁ x• P[x]‘
where ⌜x⌝ is a varstruct

SML

$val\ \exists_1\text{\_}\mathbf{intro} : THM \;{-}{>}\; THM \;{-}{>}\; THM;$

Description Introduce ∃_1 by reference to a witness, and a uniqueness theorem.

Rule

$$\frac{\begin{array}{c}\Gamma 1 \vdash P'[t'] \\ \Gamma 2 \vdash \forall\ x \bullet P[x] \Rightarrow x = t\end{array}}{\Gamma 1 \cup \Gamma 2 \vdash \exists_1\ x \bullet P[x]} \quad \exists_1\text{\_}intro$$

Where P' is α-convertible to P, and t' is α-convertible to t. Notice that for the resulting theorem we take the varstruct, x, and the form of the predicate, P, from the second theorem.

Errors

27021 ?0 and ?1 are not of the form: ‘Γ1 ⊢ Pa[ta]‘ and
‘Γ2 ⊢ ∀ vs[x,y,..]• P[x,y...] ⇒ vs[x,y,..] = t‘
where ⌜Pa⌝ and ⌜P⌝, ⌜ta⌝ and ⌜t⌝ are α−convertible
and ⌜x⌝ is a varstruct
27054 ?0 not of the form: ‘Γ ⊢ ∀ vs[x,y,..]• P[x,y...] ⇒ vs[x,y,..] = t‘

SML

$val\ \exists_1\text{\_}\mathbf{thm} : THM;$

Description Expanded form of definition of $\exists_1$

Theorem

$$\frac{}{\begin{array}{c}\vdash \forall\ P \bullet (\$\exists_1\ P) \Leftrightarrow \\ (\exists\ t \bullet (P\ t) \wedge \\ (\forall\ x \bullet (P\ x) \Rightarrow x = t))\end{array}} \quad \exists_1\text{\_}thm$$

SML

$val\ \alpha\text{\_}\mathbf{conv} : TERM \;{-}{>}\; CONV;$

Description Returns a theorem that two terms are equal, should they be α-convertible.

Rule

$$\frac{}{\vdash t1 = t2} \quad \begin{array}{l}\alpha\text{\_}conv \\ \ulcorner t2 \urcorner \\ \ulcorner t1 \urcorner\end{array}$$

Errors

3012 ?0 and ?1 do not have the same types
7034 ?0 and ?1 are not α−convertible

SML

$val\ \beta\_\mathbf{conv} : CONV;$

Description Apply a β-reduction to an abstraction.

Rule

$$\frac{}{\vdash ((\lambda\ x\bullet\ t[x])y) = t'[y]} \quad \begin{array}{l}\beta\_conv \\ \ulcorner(\lambda\ x\bullet\ t1[x])t2\urcorner\end{array}$$

where x may be any varstruct allowed by the ICL HOL syntax, y is an instance of this structure, and t' is α-convertible to t, changed to avoid variable capture.

When the bound variable structure has a pair, where the value applied to does not, then *Fst* and *Snd* are introduced as necessary, e.g.:

Example

$\beta\_conv\ \ulcorner(\lambda\ (x,y)\bullet\ f\ x\ y)\ p\urcorner =$
$\quad \vdash (\lambda\ (x,y)\bullet\ f\ x\ y)\ p = f\ (Fst\ p)\ (Snd\ p)$

See Also *simple_β_conv*, *β_rule*

Errors

27008 ?0 is not of the form: $\ulcorner(\lambda\ x\bullet\ t1[x])t2\urcorner$
where $\ulcorner x\urcorner$ *is a varstruct*

SML

$val\ \beta\_\mathbf{rule} : THM\ {-}{>}\ THM;$

Description An elimination rule for λ, which can handle paired abstractions.

Rule

$$\frac{\Gamma \vdash (\lambda\ x\bullet\ t[x])\ y}{\Gamma \vdash t[y]} \quad \beta\_rule$$

See Also *β_conv*

Errors

27007 ?0 is not of the form: ‘$\Gamma \vdash (\lambda\ x\bullet\ t[x])\ y$‘
where $\ulcorner x\urcorner$ *is a varstruct*

SML
$val\ \epsilon\_\mathbf{elim\_rule} : TERM \rightarrow THM \rightarrow THM \rightarrow THM;$

Description Given that ϵ of a predicate satisfies that predicate, then in a different theorem we may eliminate an assumption that claims an otherwise unused variable structure satisfies the predicate.

Rule

$$\frac{\Gamma 1 \vdash t'\ (\$\epsilon\ t''); \quad \Gamma 2,\ t\ vs \vdash s}{\Gamma 1 \cup \Gamma 2 \vdash s} \quad \epsilon\_elim\_rule\ \ulcorner vs \urcorner$$

where t, t' and t'' are α-convertible, and vs is a varstruct, with no duplicates, and with its free variables occurring nowhere else in the second theorem, or in the conclusion of the first. In fact, ($\$\epsilon\ t''$) here can be any term, it is not constrained to be an application of the choice function.

Errors

4016 *?0 is not an allowed variable structure*
7019 *?0 is not of the form: '$\Gamma \vdash t1(\epsilon\ t1)$'*
7054 *?0 is not of same type as choice sub−term of first theorem*
27043 *?0 is repeated in the varstruct ?1*
27045 *Arguments ?0; ?1 and ?2 not of the form $\ulcorner vs \urcorner$; '$\Gamma 1 \vdash t\ (\epsilon\ t)$' and '$\Gamma 2,\ (t\ vs) \vdash s$'*
27051 *?0 has members appearing free in conclusion of ?1*
27052 *?0 has members appearing free in ?1 other than in assumption ?2*

SML
$val\ \epsilon\_\mathbf{intro\_rule} : THM \rightarrow THM;$

Description Given a theorem whose conclusion is a function application, we know that the "function" is a predicate, and the rule states that ϵ of this predicate will satisfy the predicate.

Rule

$$\frac{\Gamma \vdash t1\ t2}{\Gamma \vdash t1\ (\$\epsilon\ t1)} \quad \epsilon\_intro\_rule$$

Errors

7016 *?0 is not of the form: '$\Gamma \vdash t1\ t2$'*

SML
$val\ \eta\_\mathbf{conv} : CONV;$

Description The rule for η conversion.

Conversion

$$\frac{}{\vdash (\lambda\ vs \bullet\ t\ vs) = t} \quad \eta\_conv\ \ulcorner \lambda\ vs \bullet\ t\ vs \urcorner$$

where t contains no free instances of the variables of varstruct vs.

Errors

27018 *?0 is not of the form: $\ulcorner \lambda\ vs \bullet\ t\ vs' \urcorner$ where $\ulcorner vs \urcorner$ is a varstruct*
27023 *?0 is not of the form: $\ulcorner \lambda\ vs \bullet\ t\ vs \urcorner$ where $\ulcorner t \urcorner$ should not contain $\ulcorner vs \urcorner$*

SML

$val\ \lambda\_\mathbf{C} : CONV\ {-}{>}\ CONV;$

Description Apply a conversion to the body of an abstraction:

Rule

$$\frac{}{\vdash (\lambda\ x\bullet\ p[x]) = (\lambda\ x\bullet\ pa[x])} \quad \begin{array}{l}\lambda\_C \\ (c : CONV) \\ \ulcorner\lambda\ x\bullet\ p\urcorner\end{array}$$

where $c\ p[x]$ gives '$\vdash p[x] = pa[x]$'.

Errors

4002 ?0 is not of form: $\ulcorner\lambda\ vs\bullet\ t\urcorner$
7104 Result of conversion, ?0, ill−formed

Also as the failure of the conversion.

SML

$val\ \lambda\_\mathbf{eq\_rule} : TERM\ {-}{>}\ THM\ {-}{>}\ THM;$

Description Given an equational theorem, return the equation formed by abstracting the term argument (which must be an allowed variable structure) from both sides.

Rule

$$\frac{\Gamma \vdash t1[x] = t2[x]}{\Gamma \vdash (\lambda\ x\bullet\ t1[x]) = (\lambda\ x\bullet\ t2[x])} \quad \begin{array}{l}\lambda\_eq\_rule \\ \ulcorner x\urcorner\end{array}$$

Errors

4016 ?0 is not an allowed variable structure
6005 ?0 occurs free in assumption list
6020 ?0 is not of the form: '$\Gamma \vdash t1 = t2$'

SML

$val\ \lambda\_\mathbf{pair\_conv} : CONV;$

Description This conversion eliminates abstraction over pairs in favour of abstraction over elements of pairs. The bound variables of the resulting λ-abstraction do not have pair types.

Rule

$$\frac{}{\begin{array}{c}\vdash (\lambda\ v\bullet\ t) = \\ (\lambda\ (v1,\ v2)\bullet\ t'[(v1,\ v2)/v])\end{array}} \quad \begin{array}{l}\lambda\_pair\_conv \\ \ulcorner\lambda\ v{:}'a \times 'b\bullet\ t\urcorner\end{array}$$

Rule

$$\frac{}{\begin{array}{c}\vdash (\lambda\ (v,\ w)\bullet\ t) = \\ (\lambda\ ((v1,\ v2),\ (w1,\ w2))\bullet \\ \quad t[(v1,\ v2)/v,\ (w1,\ w2)/v])\end{array}} \quad \begin{array}{l}\lambda\_pair\_conv \\ \ulcorner\lambda\ (v,\ w){:}('a \times 'b) \times ('c \times 'd)\bullet\ t\urcorner\end{array}$$

and so on.

Errors

27055 The type of ?0 is not of the form $\sigma \times \tau$

SML

$val\ \lambda\_\mathbf{rule} : TERM \rightarrow THM \rightarrow THM;$

Description An introduction rule for λ:

Rule

$$\frac{\Gamma \vdash s[t]}{\Gamma \vdash (\lambda\ x \bullet s[x])\ t} \qquad \lambda\_rule \quad \ulcorner t \urcorner$$

where x is a machine generated variable.

SML

$val\ \lambda\_\mathbf{varstruct\_conv} : TERM \rightarrow CONV;$

Description This conversion is a generalisation of α_conv allowing one to convert a λ-abstraction into an equivalent λ-abstraction that differs only in the form of the varstruct and the corresponding use of *Fst* in the *Snd* in the body of the abstraction.

Rule

$$\frac{}{\vdash (\lambda\ vs2[x2,y2,...]\bullet\ t[x2,y2,...]) = (\lambda\ vs1[x1,y1,...]\bullet\ t'[x1,y1,...])} \qquad \lambda\_varstruct\_conv \quad \ulcorner \lambda\ vs1[x1,y1,...]\bullet\ t[x1,y1,...] \urcorner \quad \ulcorner \lambda\ vs2[x2,y2,...]\bullet\ t'[x2,y2,...] \urcorner$$

Where the types of $vs1[x1, y1, ...]$ and $vs2[x2, y2, ...]$ are the same, and t' and t differ only in applications of *Fst* and *Snd* to the bound variables.

For example,

Rule

$$\frac{}{\vdash ((\lambda\ x\bullet\ Fst\ x + Snd\ x = 1) = (\lambda\ (a,\ b)\bullet\ a + b = 1)} \qquad \lambda\_varstruct\_conv \quad \ulcorner \lambda\ (a,\ b)\bullet\ a + b = 1 \urcorner \quad \ulcorner \lambda\ x\bullet\ Fst\ x + Snd\ x = 1 \urcorner$$

See Also α_conv for a more limited form of renaming.

Errors

27050 Cannot prove equality of ?0 and ?1

7.2 Subgoal Package

SML

signature **SubgoalPackage** = *sig*

Description This provides the subgoal package, which provides an interactive backward proof mechanism, based on the application of tactics.

Errors

30009 There are no goals to prove
30017 Label ?0 has no corresponding goal
30023 ?0 cannot be interpreted as a goal
30028 Label may not contain ?0, as less than 1
30041 Label ?0 has been superseded
30042 Label may not contain 0
30043 Label ?0 has been achieved
30045 Label cannot be empty
30055 The last change to the subgoal package state was made in a context which is no longer valid
30056 The current goal contains distinct free variables with the same names but different types, the names being ?0, and a typing context is being maintained. These free variables have not been put in the typing context
30059 The current goal contains two or more distinct free variables with the same name but different types, the name being ?0, and a typing context is being maintained. These free variables have not been put in the typing context
30061 The tactic generated an invalid proof (?0). The goal state has not been changed

These messages are common to various functions in this document. Message 30055 indicates that the goal state theorem failed the *valid_thm* test: this could be a theory out of scope, a deletion of a definition, etc. Messages 30056 and 30057 are just for the user's information, though they should give cause to worry.

SML

(∗ **pp′TS** ∗)

Description The theory will contain a constant named *pp′TS*, defined by a definition with key "pp'TS".

Definition

$\vdash \forall\ x \bullet (pp'TS\ x) \Leftrightarrow x$

This is used in creating a term form goal. Using this constant explicitly within the subgoal package may cause unexpected behaviour.

Uses THe definition may be used when analysing goal state theorems, or using *modify_goal_state_thm* (q.v.) - both operations are only for the advanced user or extender of the system.

SML

(∗ **subgoal_package_quiet** : *bool* ∗)

Description This is a system control, handled by *set_flag*. If set to false (the default) then the package narrates its progress as described in the design of its components. If set to true then the package will cause no output other than the actual results of functions. This includes, e.g., *print_goal* and *apply_tactic*.

Uses For running the package offline.

SML
|(∗ **subgoal_package_ti_context** : *bool* ∗)

Description *subgoal_package_ti_context* is a system control flag, as handled by *set_flag*, etc. If set to true (the default) then the type context will be set and maintained, via *set_ti_context*(q.v.), to be just the free variables of the current goal, each time the current goal changes. If false, then the type context will be cleared and left unchanged by goal state changes. If the current goal has free variables with the same name and differing types this will cause *set_ti_context* to ignore those variables, raising the comment message 30056.

SML
|(∗ **tactic_subgoal_warning** : *integer control* ∗)

Description Warning 30018 will be issued by *apply_tactic* (and *a*) if the tactic requests more subgoals than the number set by this control. This allows the user to avoid processing and printing large numbers of subgoals when these are probably unwanted. The default value is 20. If the value is less than zero then the warning will never be issued.

SML
|(∗ **undo_buffer_length** : *int* ∗)

Description This is a system control, handled by *set_int_control*, etc, which sets the maximum number of entries that can be held on the undo buffer for each main goal: i.e. how many tactic applications, etc, may be undone. It is initially set to 12, and cannot be made negative. Any changes to this parameter will take immediate effect upon the undo buffers stored for all the main goals, i.e. if necessary they will be shortened at the point of changing the value, rather than at the point of, e.g., applying a new tactic.

SML
|*type* **GOAL_STATE**;

Description This is an abstract data type that embodies a goal state, in particular it contains which goals are yet to be achieved and a theorem embedding the inference work so far. The subgoal package has a current goal state, a stack of goal states for different main goals, and a buffer of goal states to allow some operations to be undone.

See Also *print_goal_state*

SML

```
val apply_tactic : TACTIC -> unit;
val a : TACTIC -> unit;
```

Description *apply_tactic* applies a tactic to the current goal, and *a* is an alias for it. If successful, the previous goal state will be put in the undo buffer, and the new goal state, current goal, etc, will be based on the tactic's application. If the tactic returns some subgoals then the "first" of these will become the new current goal. If there is only one subgoal it will inherit the label of the previous current goal, otherwise if the old label was "label" then it will be considered in the goal state as superseded, and the new subgoals will be labeled "label.1", "label.2", etc. If it produces a theorem that achieves the current goal (i.e. the list of subgoals is empty), then the "next" goal will become the current one, and the previous goal's label will be noted as achieved.

The subgoals created, or if none, the "next" goal, will be displayed, using the format of *print_goal*(q.v), but with goal labels also given. Following the display of the new goals the subgoal package will issue warning messages about these goals if they are somehow "suspicious": for example it will warn if the goal state is not changed by applying the tactic.

Warning 30018 will be issued if the tactic requests more subgoals than the number set by control *tactic_subgoal_warning*. This allows the user to avoid processing and printing large numbers of subgoals when these are probably unwanted.

See Also *print_goal* for the display format of the goals.

Errors

30007 There is no current goal
30008 Result of tactic, ?0, did not match the current goal
30018 Tactic has requested ?0 subgoals, which exceeds the threshold set by tactic_subgoal_warning

SML

```
val drop_main_goal : unit -> GOAL;
```

Description Pop the current goal state from the main goal stack throwing away it and any work upon it, and making the previous entry on the stack the new current goal state, displaying the current top goal, if appropriate. The function returns the main goal dropped.

Errors

30010 The subgoal package is not in use

SML

```
val get_asm : int -> TERM;
```

Description *get_asm n* returns the *n*th assumption of the current goal.

Errors

30026 There is no current goal
30027 There is no assumption ?0 in the current goal

SML
```
val modify_goal_state_thm : (THM -> THM) -> ((string list * GOAL)list) -> unit;
```

Description *modify_goal_state_thm rule label* is a powerful hook into the subgoal package that works as follows:

1. Extract the goal state theorem
2. Apply a user-supplied inference rule *rule* to the theorem.
3. Make a new goal state, in which the goal state theorem is this new theorem.
4. In the new goal state any goals found (up to α-conversion) in the association list *label* will be labelled with their corresponding labels in the association list. Multiple entries for the same goal in the list will cause the labels to be accumulated, resulting in duplicated goals in the new goal state. If *top_goals*() (q.v.) is used for this association list then all unchanged goals will gain their original labels.
5. Label otherwise unlabelled goals with unused single natural number labels (the first available ones from the list "1", "2",...)
6. Treat this new goal state as if it had been created by a tactics application, e.g. it becomes the current goal state, the previous goal state is put on the undo list, the user is told the next goal to prove, etc.

This will issue a warning on its use should the main goal have changed, and on attempting to extract an achieved, or goal state, theorem from a goal state that is derived from the modified one. This is so that the user is warned that the result of an apparently successful *pop_thm* is not an achievement of the initially set main goal.

Uses This function is intended for system builders wishing to write extensions to the package that change the overall proof tree, not an individual goal.

Errors
```
30039 Two labels clash: ?0 and ?1
30040 Duplicate labels ?0 given for different terms
30051 Inference rule returned '?0' which is not a goal state theorem
```

SML
```
val pending_reset_subgoal_package : unit -> unit -> unit;
```

Description This function, applied to () takes a snapshot of the current subgoal package state - its stack of goal states, undo and redo buffers, and implicitly the current goal label, etc. This snapshot, if then applied to () will overwrite the then current subgoal package state with the snapshot. This does not reset, e.g., the current theory to the one at the time of taking the snapshot, so care must be taken in using this function.

Uses Primarily in saving the subgoal package state between sessions of ProofPower, via *save_and_quit*.

SML
```
val pop_thm : unit -> THM;
```

Description If the top achieved theorem is available (i.e. the theorem whose sequent is the main goal has been achieved), this function returns it, and then pops the previous goal state (if any) off the main goal stack, restoring its current goal and labelling. If present, the new current top goal will be displayed in the format used by *print_goal*. If the current proof is incomplete the function fails, having no effect.

If the user wishes to examine the top achieved theorem without popping the main goal stack, then they should use *top_thm* (q.v.).

The user will be informed if main goal has changed from the initially set main goal, by using *modify_goal_state_thm*(q.v).

See Also *save_pop_thm*, *top_thm*

Errors
```
30010 The subgoal package is not in use
30011 The current proof is incomplete
```

SML
```
val print_current_goal : unit -> unit;
```

Description Displays, with its label, the current goal of the current goal state: the goal to which a tactic will be applied.

Errors
```
30026 There is no current goal
```

SML
```
val print_goal_state : GOAL_STATE -> unit;
```

Description Display the given goal state. This displays the main goal, the goals yet to be proven, and the current goal.

SML
```
val print_goal : GOAL -> unit;
```

Description Display a goal (i.e. a conclusion and a list of assumptions) in the manner of the other subgoal package functions. This presents the list of assumptions in the goal first, numbered by their position, and in reverse order, and then the conclusion, distinguished from the assumptions by a turnstile.

Example
```
(*      3 *)   ⌜a ⇒ ¬ b⌝
(*      2 *)   ⌜a ⇒
                a ⇒
                a ⇒ b⌝
(*      1 *)   ⌜¬ b ⇒ a⌝

(*     ?⊢ *)   ⌜a ∨ b⌝
```

where ⌜$\neg\ b \Rightarrow a$⌝ is the first assumption, and the second assumption is too long to fit on one line. Then with no assumptions:

Example
```
(*     ?⊢ *)   ⌜a ∨ b⌝
```

SML

val **push_goal_state_thm** : *THM* −> *unit*;

Description Given a theorem that is of the form of a goal state theorem (e.g. gained by *top_goal_state_thm*, q.v.), set a new current main goal to be the conclusion of the input theorem (viewed as a term form goal). The current goal in the new goal state will be the first assumption of the input theorem, viewed as a term form goal. If it is the only assumption of the theorem argument then the corresponding goal will have label ""; otherwise label "1", and the other assumptions of the theorem will become subsequent goals with labels "2","3",... This new goal state is pushed onto the main goal stack. The current undo buffer will also be stacked, and a new empty one made current.

Uses For the advanced user, interested in partial proof.

Errors

30005 ?0 cannot be viewed as a goal state theorem
30058 Two distinct variables with name ?0 occur free in the goal

SML

val **push_goal_state** : *GOAL_STATE* −> *unit*;

Description If the value given is "well-formed", then this function pushes the current goal state onto the main goal stack, and sets the given value as the current goal state. The most likely reason that a goal state value is ill-formed is that it is not being pushed in the same context as it was formed, e.g. it was formed in a theory that is now out of scope, e.g. because the user has changed theory since the states creation. The current undo buffer will also be stacked, and a new empty one made current.

See Also *top_goal_state*

SML

val **push_goal** : *GOAL* −> *unit*;

Description Sets a new current main goal, creating an appropriate goal state and pushing it onto the main goal stack. The current (and only) goal in the new goal state will be the main goal, with label "". The current goal will be displayed. The current undo buffer will also be stacked, and a new empty one made current.

See Also *set_goal*

Errors

30002 The conclusion of the goal, ?0, is not of type BOOL
30003 An assumption of the goal, ?0, is not of type BOOL
30004 Two assumption of the goal (?0 and ?1) are α*−convertible*
30058 Two distinct variables with name ?0 occur free in the goal

SML

val **redo** : *unit* −> *unit*;

Description If the last command to affect the goal state was an *undo*(q.v) then this command will undo its effect (including leaving the undo buffer in its previous form, without mention of the *undo* or *redo*).

Errors

30014 The last command to affect the goal state was not an undo

SML
```
val save_pop_thm : string -> THM;
```

Description If the top achieved theorem is available (i.e. the theorem whose sequent is the main goal has been achieved), this function returns it, as well as saving it under the given string key on the current theory, and then pops the previous goal state (if any) of the main goal stack, restoring its current goal and labelling. If present, the new current top goal will be displayed in the format used by *print_goal*. If the current proof is incomplete, or the key is already used in the current theory, the function fails, having no effect.

The user will be informed if main goal has changed from the initially set main goal, by using *modify_goal_state_thm*(q.v).

The user will be informed if main goal has changed from the initially set main goal, by using *modify_goal_state_thm*(q.v).

See Also *pop_thm*, *top_thm*

Errors
```
30010 The subgoal package is not in use
30011 The current proof is incomplete
```

Failures also as *save_thm*, but given as originating from this function.

SML
```
val set_goal : GOAL -> unit;
```

Description This first discards, if it exists, the current main goal (but not any previously pushed main goals). It then sets a new current main goal, creating an appropriate goal state and pushing it onto the main goal stack. The current (and only) goal in the new goal state will be the main goal, with label "". The current goal will be displayed. The current undo buffer will also be stacked, and a new empty one made current.

Defn
```
set_goal gl = (drop_main_goal() handle (Fail _) => ();
        push_goal gl);
```

Uses In restarting a proof that has "gone wrong", perhaps by

```
set_goal(top_main_goal());
```

See Also *push_goal*

Errors
```
30002 The conclusion of the goal, ?0, is not of type BOOL
30003 An assumption of the goal, ?0, is not of type BOOL
30004 Two assumption of the goal (?0 and ?1) are α−convertible
30058 Two distinct variables with name ?0 occur free in the goal
```

SML
```
val set_labelled_goal : string -> unit;
```

Description If the string is a valid label in the current goal state, then set the corresponding goal as the current goal, and then display it.

Errors
```
30010 The subgoal package is not in use
30016 ?0 is not of the form "n1.n2....nm"
```

SML
```
val simplify_goal_state_thm : THM -> THM;
```

Description This will simplify a goal state theorem (e.g from *top_goal_state_thm*, q.v.), stripping off assumptions from the conclusion of the theorem up to the turnstile place marker, then removing the place marker itself in both conclusion and assumptions.

Uses For the advanced user, interested in partial proofs.

Errors
```
30005 ?0 cannot be viewed as a goal state theorem
```

SML
```
val subgoal_package_size : unit -> int;
```

Description This returns the size of the subgoal package's storage, in words - where one word is four bytes.

This facility is not available in all versions of ProofPower. The function will produce the following warning message and return −*1* in this case

Errors
```
30060 This function is not supported in this version of ProofPower
```

SML
```
val top_current_label : unit -> string;
```

Description Returns the label of the current goal: the goal to which a tactic will be applied.

Errors
```
30026 There is no current goal
```

SML
```
val top_goals : unit -> (string list * GOAL)list;
```

Description Returns all the goals yet to be achieved, and their associated labels (they may have more than one), in the current goal state.

Uses To determine what goals are left to achieve.

Errors
```
30010 The subgoal package is not in use
```

SML
```
val top_goal_state_thm : unit -> THM;
```

Description This returns the goal state theorem of the current goal state. It is a partial proof of the main goal, though in a somewhat unwieldy form, as it encodes the main goal, and its other goals in a term form. It may be simplified by using *simplify_goal_state_thm*(q.v). The theorem is suitable for setting a new main goal, by using *push_goal_state_thm*(q.v). The user is informed if the goal state has achieved its theorem The user will also be informed if main goal has changed from the initially set main goal, by using *modify_goal_state_thm*(q.v).

Uses For the advanced user, interested in partial proofs.

Errors
```
30010 The subgoal package is not in use
```

SML
```
val top_goal_state : unit -> GOAL_STATE;
```

Description This provides the current goal state as a value: note that a goal state does not contain an undo buffer, and thus function does not return the current undo buffer.

See Also *push_goal_state*

Errors
```
30010 The subgoal package is not in use
```

SML
```
val top_goal : unit -> GOAL;
```

Description Returns the current goal of the current goal state: the goal to which a tactic will be applied.

Errors
```
30026 There is no current goal
```

SML
```
val top_labelled_goal : string -> GOAL;
```

Description Returns the goal with the given label, should it exist in the current goal state. Note that superseded and achieved goals are not available from the goal state.

Errors
```
30016 ?0 is not of the form "n1.n2....nm"
```

SML
```
val top_main_goal : unit -> GOAL;
```

Description Return the current main goal: the objective of the current proof attempt.

Errors
```
30025 There is no current main goal
```

SML
```
val top_thm : unit -> THM;
```

Description If the top achieved theorem (i.e. the theorem whose sequent is the main goal has been achieved) is available, this function returns it, without affecting the current goal state. If the current proof is incomplete the function fails.

The user will be informed if main goal has changed from the initially set main goal, by using *modify_goal_state_thm*(q.v).

See Also *pop_thm*, *save_pop_thm*

Errors
```
30010 The subgoal package is not in use
30011 The current proof is incomplete
```

SML

```
val undo : int -> unit;
```

Description *undo n* will take the *n*th entry from the undo buffer, if there are sufficient, as the current goal state. Attempting to go past the end of the buffer will cause a failure, rather than a partial undoing. A single *undo* command can itself be undone by *redo*(q.v), but otherwise entries on the undo buffer between its start and the *n*th entry will be discarded.

Note that the undo buffer is stacked on starting a new main goal (e.g. with *push_goal*), and unstacked on popping the current main goal (e.g. with *pop_thm* or *drop_main_goal*).

Errors

```
30010 The subgoal package is not in use
30012 Attempted to undo ?0 time?1 with only ?2 entr?3 in the undo buffer
30013 Must undo a positive number of times
```

7.3 General Tactics and Tacticals

SML
```
signature Tactics1 = sig
```

Description This provides the first group of tactics and tacticals in ICL HOL.

SML
```
signature Tactics2 = sig
```

Description This provides the second group of tactics and tacticals in ICL HOL. These are mainly concerned with the predicate calculus.

SML
```
signature Tactics3 = sig
```

Description This provides a third group of tactics. They are primarily concerned with adding handling for paired abstractions.

SML
```
type GOAL      (* = SEQ *);
type PROOF     (* = THM list -> THM *);
type TACTIC    (* = GOAL -> (GOAL list * PROOF) *);
```

Description *TACTIC* is the type of tactics. The types *GOAL* and *PROOF* help to abbreviate its definition.

SML
```
type THM_TACTIC        (* = THM -> TACTIC *);

type THM_TACTICAL      (* = THM_TACTIC -> THM_TACTIC *);
```

Description These are the types of theorem tactics and theorem tacticals.

SML
```
val accept_tac : THM -> TACTIC;
```

Description Prove a goal by a theorem which is α-convertible to it.

Tactic

$$\frac{\{\ \Gamma 2\ \}\ t2}{} \qquad \begin{array}{l} accept\_tac \\ \Gamma 1 \vdash t1 \end{array}$$

where *t1* and *t2* are α-convertible.

Errors
```
9102 ?0 is not α−convertible to the goals conclusion ?1
```

SML
```
val all_asm_ante_tac : TACTIC;
```

Description Apply *asm_ante_tac* to every assumption in turn:

Tactic

$$\frac{\{\ t1,\ ...,\ tn\ \}\ t}{\{\}\ tn \Rightarrow ... \Rightarrow t1 \Rightarrow t} \qquad all\_asm\_ante\_tac$$

α-equivalent assumptions will only appear once in the resulting goal. Notice that the first assumption becomes the rightmost antecedent.

See Also *asm_ante_tac*, *list_asm_ante_tac*

Errors
```
28055 The conclusion or an assumption of goal does not have type ⌜:BOOL⌝
```

SML

```
val all_var_elim_asm_tac : TACTIC;
val all_var_elim_asm_tac1 : TACTIC;
val ALL_VAR_ELIM_ASM_T : (THM -> TACTIC) -> TACTIC;
val ALL_VAR_ELIM_ASM_T1 : (THM -> TACTIC) -> TACTIC;
```

Description These tactics and tacticals do variable elimination with all the appropriate assumptions of the goal. They process one or more assumptions of the form: ⌜*var* = *value*⌝ or ⌜*value* = *var*⌝, where *var* is a variable and the subterm *value* satisfies a tactic-specific requirement, eliminating the variable *var* in favour of the *value*.

If an assumption is an equation of variables, which all of the listed tactics accept, then the tactic will strip digits and the current variant suffix from the right of the two variable names, and will choose to eliminate the variable with the shortest remaining name string, taking eliminating the left hand side variable if the strings are of equal length (this is a heuristic). If the variables are the same then the assumption is just discarded with no further effect.

all_var_elim_asm_tac will first extract all the goal's assumptions, holding them in a "pool". It will examine each assumption of the required form in turn, starting at the assumptions from the head of the assumption list. To eliminate a variable *var* using an assumption it requires that the *value* to which it is equated is also a variable, or an isolated constant (this is more restrictive than *var_elim_asm_tac*). All the occurrences of the variable will be eliminated from the rest of the assumptions in the pool, and from the conclusion of the goal, and the assumption discarded from the pool. Each of the assumptions in the pool will be examined once, as the process described so far will only exceptionally introduce new equations that can be used for variable elimination.

Finally, the remaining assumptions in the pool will be returned to the goal's assumption list - if an individual assumption is unchanged then it will be returned by *check_asm_tac*, otherwise it will be stripped back into the assumption list by *strip_asm_tac*. This stripping may result in further possible variable eliminations being enabled, and indeed certain fairly unlikely combinations of assumptions and proof contexts may result in *REPEAT all_var_elim_asm_tac* not halting. *ALL_VAR_ELIM_ASM_T* allows the users choice of function to be applied to the modified assumptions, rather than *strip_asm_tac*.

all_var_elim_asm_tac1 works as *all_var_elim_asm_tac*, except that an assumption will be used to eliminate a variable *var* if the *value* to which it is equated does not contain *var* free (i.e. its requirement is as *var_elim_asm_tac*). *ALL_VAR_ELIM_ASM_T1* allows the users choice of function to be applied to the modified assumptions.

All the functions fail if they find no assumptions that can be used to eliminate variables.

Uses General purpose, and in *basic_prove_tac*.

See Also *prop_eq_prove_tac* for more sophisticated approach to these kinds of problems.

Errors

```
29028 This tactic is unable to eliminate any variable
```

SML

```
val all_β_tac : TACTIC;
```

Description This tactic will β-reduce all β-redexes in the goal's conclusion, including those redexes introduced by preceding β-reductions in the same tactic application.

Uses In most proof contexts β-reduction will be a side effect of rewriting: this tactic is intended for cases where rewriting would do "too much".

See Also *all_β_rule*, *all_β_conv*

Errors

```
27049 ?0 contains no β−redexes
```

SML

```
val all_ϵ_tac : TACTIC;
val ALL_ϵ_T : (THM −> TACTIC) −> TACTIC;
```

Description *all_ϵ_tac* applies *ϵ_tac* to all subterms of the conclusion of the goal of the form $\epsilon x \bullet t$. *ALL_ϵ_T* is similar but uses *ϵ_T* rather than *ϵ_tac*. The effect is to set the corresponding terms of the form $\exists x \bullet t$ as lemmas, and to derive new assumptions of the form $t[\epsilon x \bullet t/x]$.

Tactic

$$\frac{\{\ \Gamma\ \}\ t[\epsilon x_1 \bullet t_1/y_1,\ ...,\ \epsilon x_k \bullet t_k/y_k]}{\begin{array}{c}\{\ \Gamma\ \}\ \exists x_1 \bullet t_1;\ ...;\ \{\ \Gamma\ \}\ \exists x_k \bullet t_k;\\ \{strip\ \ t_1[\epsilon x_1 \bullet t_1/x_1],\ ...,\\ strip\ \ t_k[\epsilon x_k \bullet t_k/x_k],\ \Gamma\ \}\ t\end{array}} \qquad \epsilon\_tac$$

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable. This occurs when the use of the choice function is in some sense irrelevant to the truth of the goal, e.g., $(\epsilon x \bullet T) = (\epsilon x \bullet T)$.

SML

```
val ante_tac : THM −> TACTIC;
```

Description Replace a goal with conclusion *t2* by *t1⇒t2*, where the antecedent, *t1*, of the implication is the conclusion of a theorem:

Tactic

$$\frac{\{\ \Gamma 2\ \}\ t2}{\{\ \Gamma 2\ \}\ t1\ \Rightarrow\ t2} \qquad ante\_tac\ (\Gamma 1 \vdash t1)$$

where the assumptions, $\Gamma 1$, of the theorem are contained in the assumptions, $\Gamma 2$ of the goal.

Uses This is often useful if one needs to transform the conclusion of theorem e.g. by rewriting with the assumptions.

See Also *asm_tac*, *strip_asm_tac*

Errors

```
28027 Conclusion of goal does not have type ⌜:BOOL⌝
```

SML

val **asm_ante_tac** : *TERM −> TACTIC* ;

Description Bring a term out of the assumption list into the goal as the antecedent of an implication.

Rule

$$\frac{\{\ \Gamma,\ t1'\ \}\ t2}{\{\ \Gamma\ \} \vdash t1 \Rightarrow t2} \qquad \begin{array}{l} asm\_ante\_tac \\ \ulcorner t1 \urcorner \end{array}$$

where *t1* and *t1′* are α-convertible. Note that all assumptions α-convertible with *t1* are removed.

Uses Typically to make the assumption amenable to manipulation, e.g. by a rewriting tactic.

See Also *list_asm_ante_tac*, *all_asm_ante_tac*, *swap_asm_concl_tac*, *DROP_ASM_T*.

Errors

28052 Term ?0 is not in the assumptions
28055 The conclusion or an assumption of goal does not have type ⌜:BOOL⌝

SML

val **asm_tac** : *THM −> TACTIC*;

Description *asm_tac thm* is a tactic which adds the conclusion of the theorem, *thm*, into the assumptions of a goal:

Tactic

$$\frac{\{\ \Gamma 2\ \}\ t2}{\{\ t1,\ \Gamma 2\ \}\ t2} \qquad \begin{array}{l} asm\_tac \\ \Gamma 1 \vdash t1 \end{array}$$

SML

```
val back_chain_tac : THM list -> TACTIC;
val bc_tac : THM list -> TACTIC;
```

Description *back_chain_tac* is a tactic which uses theorems whose conclusions are possibly universally quantified implications or bi-implications, to reason backwards from the conclusion of a goal. (*bc_tac* is an alias for *back_chain_tac*.) The tactic repeatedly performs the following steps:

1. Scan the list of theorems looking for an implication, $t1 \Rightarrow t2$, or a bi-implication $t1 \Leftrightarrow t2$ for which the conclusion of the goal is a substitution instance, $t2'$ say, of $t2$. If no such theorem is found then stop.

2. If in step 1, an applicable theorem, say *thm*, has been found reduce the goal to the corresponding instance of *t1* (or an existentially quantified version thereof) using *bc_thm_tac*, q.v.

3. Repeatedly apply *∀_tac* or *∧_tac* until neither of these is applicable.

4. Delete *thm* from the list of theorems and return to step 1.

In step 4, only the first appearance of *thm* is removed from the list, so that one can arrange for a theorem to be used more than once by the tactic by putting several copies of it in the list.

For example:

Example

$$\frac{\{\ \Gamma\ \}\ t3'}{\{\ \Gamma\ \}\ t4';\ \{\ \Gamma\ \}\ t5'} \qquad \begin{array}{l} \Rightarrow\_tac \\ [\Gamma 1 \vdash t1 \land (\forall x \bullet t2) \Rightarrow t3, \\ \Gamma 2 \vdash t4 \Leftrightarrow t1, \\ \Gamma 3 \vdash t5 \Rightarrow t2, \end{array}$$

(Here $t3'$ is some substitution instance of $t3$ and $t4'$ and $t5'$ are the corresponding instances of $t4$ and $t5$.)

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

See Also *bc_thm_tac* (which is used to perform step 2).

Errors

29012 *Theorem ?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u ⇔ v‘*
or ‘Γ ⊢ ∀ x1 ... xn • u ⇒ v‘

SML

val **back_chain_thm_tac** : *THM −> TACTIC*;
val **bc_thm_tac** : *THM −> TACTIC*;

Description *back_chain_thm_tac* is a tactic which uses a theorem whose conclusion is a possibly universally quantified implication or bi-implication to chain backwards one step from the conclusion of a goal. (*bc_thm_tac* is an alias for *back_chain_thm_tac*.) The effect is as follows:

Tactic

$$\frac{\{\ \Gamma\ \}\ t2'}{\{\ \Gamma\ \}\ t1'} \qquad \begin{array}{l} bc\_thm\_tac \\ \Gamma 1 \vdash t1 \Rightarrow t2 \end{array}$$

Tactic

$$\frac{\{\ \Gamma\ \}\ t2'}{\{\ \Gamma\ \}\ t1'} \qquad \begin{array}{l} bc\_thm\_tac \\ \Gamma 1 \vdash t1 \Leftrightarrow t2 \end{array}$$

where $t2'$ is an instance (under type instantiation and substitution) of $t2$ and $t1'$ is the corresponding instance of $t1$. If $t1'$ contains free variables which do not appear in the assumptions of the instantiated theorem or in $t2'$, then the new subgoal $t1'$ will be existentially quantified over these variables. For example,

Example

$$\frac{\{\ \Gamma\ \}\ a < b}{\{\ \Gamma\ \}\ \exists\ i\bullet\ a < i \land i < b} \qquad \begin{array}{l} bc\_thm\_tac \\ \vdash \forall\ m\ i\ n\bullet\ m < i \land i < n \Rightarrow m < n \end{array}$$

Note that, bi-implications are in effect treated as right-to-left rewrite rules at the top level by this tactic. The standard rewriting mechanisms may be used for left-to-right rewriting.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

See Also *back_chain_tac* (which supplies a more general facility).

Errors

29011 Conclusion of the goal is not an instance of: *?0*
29012 Theorem ?0 is not of the form ‘$\Gamma \vdash \forall\ x1\ ...\ xn \bullet u \Leftrightarrow v$‘
or ‘$\Gamma \vdash \forall\ x1\ ...\ xn \bullet u \Rightarrow v$‘

SML

val **bad_proof** : *string −> 'a*

Description *bad_proof name* is equivalent to *error name 9001* []. *bad_proof* is for use in low level tactical programming to report the error situation when the proof generated by a tactic is supplied with the wrong number of arguments. (This will not happen for the usual use of tactics with *tac_proof* or within the subgoal package):

Errors

9001 the proof of the subgoals has produced the wrong number of theorems

Uses Specialised low-level tactic programming.

SML
```
val CASES_T2 : TERM -> (THM -> TACTIC) ->
        (THM -> TACTIC) -> TACTIC;
```

Description Do a case split on a given boolean term using two tactic generating functions:

$CASES\_T2\ t1\ ttac1\ ttac2\ (\{\Gamma\}\ t2) = ttac1(t1 \vdash t1)(\{\Gamma\}\ t2)\ ;\ ttac2(\neg t1 \vdash \neg t1)(\{\Gamma\}\ t2)$

See Also *cases_tac*, ∨_*THEN*, *CASES_T*

Errors

28022 ?0 is not boolean

SML
```
val cases_tac : TERM -> TACTIC;
```

Description Do a case split on a given boolean term.

Tactic

$$\frac{\{\ \Gamma\ \}\ t2}{\{strip\ t1,\ \Gamma\ \}\ t2;\ \{strip\ \neg t1,\ \Gamma\ \}\ t2} \qquad \begin{array}{l} cases\_tac \\ \ulcorner t1 \urcorner \end{array}$$

See Also *CASES_T*, ∨_*THEN*

Errors

28022 ?0 is not boolean

SML
```
val CASES_T : TERM -> (THM -> TACTIC) -> TACTIC;
```

Description Do a case split on a given boolean term using a tactic generating function:

$CASES\_T\ t1\ ttac\ (\{\Gamma\}\ t2) = ttac(t1 \vdash t1)(\{\Gamma\}\ t2)\ ;\ ttac(\neg t1 \vdash \neg t1)(\{\Gamma\}\ t2)$

See Also *cases_tac*, ∨_*THEN*, *CASES_T2*

Errors

28022 ?0 is not boolean

SML
```
val CHANGED_T : TACTIC -> TACTIC;
```

Description *CHANGED_T tac* is a tactic which applies *tac* to the goal and fails if this results in a single subgoal which is α−convertible to the original goal.

Uses *CHANGED_T* can be a useful way of ensuring termination of, e.g., rewriting tactics.

Errors

9601 the tactic did not change the goal

SML
```
val check_asm_tac : THM -> TACTIC;
```

Description *check_asm_tac thm* is a tactic which checks the form of the theorem, *thm*, and then takes the first applicable action from the following table:

thm	action
$\Gamma \vdash t$	proves goal if its conclusion is t
$\Gamma \vdash T$	as *id_tac* (i.e. the theorem is discarded)
$\Gamma \vdash F$	proves goal
$\Gamma \vdash \neg t$	proves goal if t in assumptions, else as *asm_tac*
$\Gamma \vdash t$	proves goal if $\neg t$ in assumptions, else as *asm_tac*

During the search through the assumptions in the last two cases, *check_asm_tac* also checks to see whether any of the assumptions is equal to the conclusion of the goal, and if so proves the goal. It also checks to see if the conclusion of the theorem is already an assumption, in which case the tactic has no effect. When all the assumptions have been examined, if none of the above actions is applicable, the conclusion of the theorem is added to the assumption list.

Uses Tactic programming.

See Also *strip_asm_tac*, *strip_tac*.

SML
```
val concl_in_asms_tac : TACTIC;
```

Description *concl_in_asms_tac* is a tactic which checks whether the conclusion of the goal is also in the assumptions, and if so proves the goal.

Tactic

$$\frac{\{\ \Gamma,\ t\ \}\ t'}{}\qquad concl\_in\_asms\_tac$$

where t and t' are α-convertible.

Uses Tactic programming.

See Also *strip_tac*.

Errors
```
28002 Goal does not appear in the assumptions
```

SML
```
val COND_T : (GOAL -> bool) -> TACTIC -> TACTIC -> TACTIC;
```

Description *COND_T p tac1 tac2* is a tactic which acts as *tac1* if the predicate *p* holds for the goal, otherwise it acts as *tac2*.

Example
```
COND_T (is_¬ o snd) (cases_tac ⌜X:BOOL⌝) strip_tac
```

is a tactic which does a case split on ⌜X⌝ if the goal is a negation and behaves as *strip_tac* otherwise.

Uses For constructing larger tactics, in cases where the more common idiom using *ORELSE* would not have the desired effect.

See Also *ORELSE*

Errors As determined by the arguments.

SML
val **contr_tac** : *TACTIC*;

Description A form of proof by contradiction: t holds if $\neg t \vdash F$.

(The name stands for classical contradiction, as opposed to the intuitionistic contradiction proof of *i_contr_tac*.)

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{strip\ \neg t,\ \Gamma\}\ F} \quad contr\_tac$$

Uses Proof by contradiction.

See Also *strip_tac*, *¬_tac*.

Errors
28027 Conclusion of goal does not have type ⌜:BOOL⌝

SML
val **CONTR_T** : (*THM −> TACTIC*) *−> TACTIC*;

Description A form of proof by contradiction as a tactical. *CONTR_T thmtac* is a tactic which attempts to solve a goal (Γ, t), by applying $thmtac(\neg t \vdash \neg t)$ to the goal (Γ, F).

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{thmtac\ (\neg t\ \vdash\ \neg t)\ (\{\Gamma\}\ F)} \quad CONTR\_T\ thmtac$$

Uses Proof by contradiction in combination with a theorem tactic.

See Also *contr_tac*, *¬_T*.

Errors
28027 Conclusion of goal does not have type ⌜:BOOL⌝

SML
val **conv_tac** : *CONV −> TACTIC*;

Description *conv_tac conv* is a tactic which applies the conversion *conv* to the conclusion of a goal, and replaces the conclusion of the goal with the right-hand side of the resulting equational theorem if this is successful:

Tactic

$$\frac{\{\ \Gamma 2\ \}\ t2}{\{\ \Gamma 2\ \}\ t1} \quad asm\_tac\ conv$$

where $conv\ t2 = (\Gamma 1 \vdash t2 = t1)$.

Errors
9400 the conversion returned ‘?0‘ which is not of the form:
‘... ⊢ ?1 ⇔ ...‘

SML

val **CONV_THEN** : *CONV −> THM_TACTICAL*;

Description *CONV_THEN conv thmtac* is a theorem tactic which first uses *conv* to transform the conclusion of a theorem and then acts as *thmtac*.

(CONV_THEN) conv thmtac thm = thmtac (conv thm)

Uses For use in programming theorem tacticals. The function may be partially evaluated with only its conversion, theorem tactic and theorem arguments.

Errors

9400 the conversion returned '?0' which is not of the form:
'... ⊢ ?1 ⇔ ...'

SML

val **discard_tac** : *'a −> TACTIC*;
val **k_id_tac** : *'a −> TACTIC*;

Description A tactic that discards its argument, but otherwise has no effect. *k_id_tac* is an alias for *discard_tac*.

Uses Can be used to remove unwanted assumptions : *a (POP_ASM_T discard_tac)* discards the top-most assumption. This usage of *discard_tac* may strengthen the goal. ie it may result in unprovable subgoals even when the original goal was provable.

SML

val **DROP_ASMS_T** : *(THM list −> TACTIC) −> TACTIC*;

Description *DROP_ASMS_T thmstac* is a tactic which applies *asm_rule* to each assumption of the subgoal, giving a list of theorems, *thms* say, then removes all the assumptions of the goal and then acts as *thmstac thms*.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{thmstac\ (map\ asm\_rule\ \Gamma)\ (\{\}\ t)} \qquad DROP\_ASMS\_T\ \ thmstac$$

Uses To use all the assumptions as theorems.

Errors As for *thmstac*.

SML

val **DROP_ASM_T** : *TERM −> (THM −> TACTIC) −> TACTIC*;

Description *DROP_ASM_T asm thmtac* is a tactic which removes *asm* from the assumption list and then acts as *thmtac(asm⊢asm)*.

Tactic

$$\frac{\{\ \Gamma,\ asm'\ \}\ t}{thmtac\ (asm\ \vdash\ asm)\ (\{\ \Gamma\ \}\ t)} \qquad DROP\_ASM\_T\ \ \ulcorner asm \urcorner\ \ thmtac$$

where *asm* and *asm′* are α-convertible.

Uses To use an assumption as a theorem

Errors

9301 the term ?0 is not in the assumption list

SML

```
val DROP_FILTER_ASMS_T : (TERM -> bool) ->
        (THM list -> TACTIC) -> TACTIC;
```

Description *DROP_FILTER_ASMS_T pred thmstac* is a tactic which applies *asm_rule* to each assumption of the subgoal that satisfies *pred*, giving a list of theorems, *thms* say, then removes all the selected assumptions of the goal and then acts as *thmstac thms*.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{thmstac\ (map\ asm\_rule\ (\Gamma \cap pred))\ (\{\Gamma \setminus pred\}\ t)}\quad DROP\_FILTER\_ASMS\_T\ pred\ thmstac$$

Uses To use all the selected assumptions as theorems.

Errors As for *thmstac*.

SML

```
val DROP_NTH_ASM_T : int -> (THM -> TACTIC) -> TACTIC;
```

Description *DROP_NTH_ASM_T i thmtac* is a tactic which applies *asm_rule* to the *i*-th assumption of the goal, giving a theorem, *thm* say, and then removes *asm* from the assumptions and acts as *thmtac thm*.

Assumptions are numbered *1, 2 . . .*, so that, e.g., *DROP_NTH_ASM_T 1* is the same as *POP_ASM_T*

Tactic

$$\frac{\{\ a1,\ ...,\ an\ \}\ t}{thmtac\ (asm\_rule\ [ai])\ (\{\ \Gamma \setminus ai\}\ t)}\quad DROP\_NTH\_ASM\_T\ i\ thmtac$$

Uses To use an assumption as a theorem, treating the assumption list as an array.

Errors

9303 the index ?0 is out of range

SML
```
val eq_sym_asm_tac : TERM -> TACTIC;
val eq_sym_nth_asm_tac : int -> TACTIC;
```

Description THese two tactics identify an assumption (either by being equal to the term argument, or by index number). They take it from the assumption list, use symmetry upon it to reverse any equations (or bi-implications) (though equations embedded within other equations will not be reversed), and then strip the result into the assumption list. The tactics fail if there are no equations to reverse.

Tactic

$$\frac{\{\Gamma 1,\ t[x = y,\ p = q,...],\ \Gamma 2\}\ cnc}{\{strip\ t[y = x,\ q = p,\ ...],\ \Gamma 1,\ \Gamma 2\}\ cnc} \qquad \begin{array}{l} eq\_sym\_asm\_tac \\ t[x = y,\ p = q,...] \end{array}$$

Tactic

$$\frac{\begin{array}{c}\{t1,...,tn{-}1,\ tn[x = y,\ p = q,...],\\ tn{+}1,...\}\ cnc\end{array}}{\begin{array}{c}\{strip\ tn[y = x,\ q = p,\ ...],\\ t1,...,tn{-}1,\ tn{+}1,...\}\ cnc\end{array}} \qquad \begin{array}{l} eq\_sym\_nth\_asm\_tac \\ n \end{array}$$

Definition
```
fun eq_sym_asm_tac asm = DROP_ASM_T asm
        (strip_asm_tac o conv_rule(ONCE_MAP_C eq_sym_conv));
fun eq_sym_nth_asm_tac n = DROP_NTH_ASM_T n
        (strip_asm_tac o conv_rule(ONCE_MAP_C eq_sym_conv));
```

Example

Assumption	Becomes
⌜x = y⌝	⌜y = x⌝
⌜∀ x y • x ⇔ y⌝	⌜∀ x y • y ⇔ x⌝
⌜f(x = (p = q))⌝	⌜f((p = q) = x)⌝
⌜x = y ∧ p = q⌝	⌜x = y⌝, ⌜p = q⌝

Errors

9301	*the term ?0 is not in the assumption list*
9303	*the index ?0 is out of range*
28053	*?0 contains no equations*

SML
```
val EVERY_TTCL : THM_TACTICAL list -> THM_TACTICAL;
```

Description *EVERY_TTCL* is a theorem tactical combinator.

```
EVERY_TTCL [ttcl1, ttcl2, ...] = ttcl1 THEN_TTCL ttcl2 THEN_TTCL ...
```

EVERY_TTCL [] acts as *ID_THEN*.

Uses For use in programming theorem tacticals.

SML
```
val EVERY_T : TACTIC list -> TACTIC;
val EVERY : TACTIC list -> TACTIC;
```

Description *EVERY_T tlist* is a tactic that applies the head of *tlist* to its subgoal, and recursively applies the tail of *tlist* to each resulting subgoal. *EVERY* is an alias for *EVERY_T*. *EVERY* [] is equal to *id_tac*.

Example

```
EVERY [∀_tac, ∧_tac, ∀_tac]
        is equivalent to
∀_tac THEN ∧_tac THEN ∀_tac
```

Errors As for the tactics in the list.

SML
```
val fail_tac : TACTIC;
```

Description A tactic that always fails. This is the identity for the tactical *ORELSE_T*

Uses For constructing larger tactics.

Errors
```
9201    failed as requested
```

SML
```
val FAIL_THEN : THM_TACTICAL;
```

Description This is a theorem tactical which always fails at the point it receives its theorem (having already been given a theorem tactic). It acts as the identity for the theorem tactical combinator *ORELSE_TTCL*.

Uses For use in programming theorem tacticals.

Errors
```
9401    failed as requested
```

SML
```
val fail_with_tac : string -> int -> (unit -> string) list -> TACTIC;
```

Description *fail_with_tac area msg inserts* is a tactic that always fails, reporting an error message via the call *fail area msg inserts*.

Uses For constructing larger tactics.

See Also *fail*

Errors As determined by the arguments.

SML
```
val FAIL_WITH_THEN : string -> int -> (unit -> string) list -> THM_TACTICAL;
```

Description *FAIL_WITH_THEN area msg inserts* is a theorem tactical that always fails when given its theorem (having already been given a theorem tactic), reporting an error message via the call *fail area msg inserts*.

Uses For constructing larger theorem tacticals.

See Also *fail*

Errors As determined by the arguments.

SML

```
val FIRST_TTCL : THM_TACTICAL list -> THM_TACTICAL;
```

Description *FIRST_TTCL* is a theorem tactical combinator. *FIRST_TTCL* [] fails on being applied to its theorem tactic and then theorem.

```
FIRST_TTCL [ttcl1, ttcl2, ...] =
        ttcl1 ORELSE_TTCL ttcl2 ORELSE_TTCL ...
```

Uses For use in programming theorem tacticals.

Errors

```
9402    the list of theorem tactics is empty
```

SML

```
val FIRST_T : TACTIC list -> TACTIC;
val FIRST : TACTIC list -> TACTIC;
```

Description *FIRST_T tlist* is a tactic that attempts to apply each tactics in *tlist* until one succeeds, or all fail. The first successful application will be the result of the tactic, and it fails if all the attempts fail. *FIRST* is an alias for *FIRST_T*. *FIRST* [] fails on being applied to any goal.

Errors

```
9105    the list of tactics is empty
```

Also as the failure of last member of a non-empty list.

SML

```
val forward_chain_tac : THM list -> TACTIC;
val fc_tac : THM list -> TACTIC;
val all_forward_chain_tac : THM list -> TACTIC;
val all_fc_tac : THM list -> TACTIC;
val asm_forward_chain_tac : THM list -> TACTIC;
val asm_fc_tac : THM list -> TACTIC;
val all_asm_forward_chain_tac : THM list -> TACTIC;
val all_asm_fc_tac : THM list -> TACTIC;
```

Description These are tactics which use theorems whose conclusions are implications, or from which implications can be derived using the canonicalisation function *fc_canon*, q.v., to reason forwards from the assumptions of a goal. (The names with *fc* are aliases for the corresponding ones with *forward_chain*.)

The basic step is to take a theorem of the form $\Gamma \vdash t1 \Rightarrow t2$ and an assumption of the form $t1'$ where $t1'$ is a substitution instance of $t1$ and to deduce the corresponding instance of $t2'$. The new theorem, $\Delta \vdash t2'$ say, may then be stripped into the assumptions.

In the case of *fc_tac* the implicative theorem is always derived from the list of theorems given as an argument. In the case of *asm_fc_tac* the assumptions are also used. In all of the tactics the rule *fc_canon* is used to derive an implicative canonical form from the candidate implicative theorems. Normally combination of an implicative theorem and an assumption is then tried in turn and all resulting theorems are stripped into the assumptions of the goal. However, if the chaining results contain a theorem whose conclusion is ⌜F⌝ then the first such found will be stripped into the assumptions, and all other theorems discarded.

If one of the implications has the form $t1 \Rightarrow t2 \Rightarrow t3$ or $t1 \land t2 \Rightarrow t3$ and if assumptions matching $t1$ and $t2$ are available, *fc_tac* or *asm_fc_tac* will derive an intermediate implication $t2 \Rightarrow t3$ and *asm_fc_tac* could then be used to derive $t3$. The variants with *all_* may be used to derive $t3$ directly without generating any intermediate implications in the assumptions. They work like the corresponding tactic without *all_* but any theorems which are derived which are themselves implications are not stripped into the assumptions but instead are used recursively to derive further theorems. When no new implications are derivable all of the non-implicative theorems derived during the process are stripped into the assumptions.

Note that the use of *fc_canon* implies that conversions from the proof context are applied to generate implications. E.g., in an appropriate proof-context covering set theory, $a \subseteq b$ might be treated as the implication $\forall x \bullet x \in a \Rightarrow x \in b$. Also variables which appear free in a theorem are not considered as candidates for instantiation (in order to give some control over the number of results generated). The tacticals, *FC_T1* and *ASM_FC_T1* may be used to avoid the use of *fc_canon*.

For example, the tactic:

```
asm_fc_tac[] THEN asm_fc_tac[]
```

will prove the goal:

```
{p x, ∀x•p x ⇒ q x, ∀x•q x ⇒ r x} r x.
```

See Also *bc_tac*, *FC_T*, *ASM_FC_T*, *FC_T1*, *ASM_FC_T1*.

SML

```
val FORWARD_CHAIN_T :
        (THM list -> TACTIC) -> THM list -> TACTIC;
val FC_T :
        (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_FORWARD_CHAIN_T :
        (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_FC_T :
        (THM list -> TACTIC) -> THM list -> TACTIC;
val ASM_FORWARD_CHAIN_T :
        (THM list -> TACTIC) -> THM list -> TACTIC;
val ASM_FC_T :
        (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_ASM_FORWARD_CHAIN_T :
        (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_ASM_FC_T :
        (THM list -> TACTIC) -> THM list -> TACTIC;
```

Description These are tacticals which use theorems whose conclusions are implications, or from which implications can be derived, to reason forwards from the assumptions of a goal. (The tacticals with *FC* are aliases for the corresponding ones with *FORWARD_CHAIN*.)

The description of *fc_tac* should be consulted for the basic forward chaining algorithms used. The significance of the final argument and of the presence or absence of *ASM* and *ALL* in the name is exactly as for *fc_tac* and its relatives.

The tacticals allow variation of the tactic generating function used to process the theorems derived by the forward inference. The tactic generating function to be used is given as the first argument.

Examples *fc_tac* is the same as: *FC_T (MAP_EVERY strip_asm_tac)*.

To rewrite the goal with the results of the forward inference one could use *FC_T rewrite_tac*.

See Also *fc_tac*, *asm_fc_tac*, *bc_tac*, *FC_T1*.

SML
```
val FORWARD_CHAIN_T1 :
    (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val FC_T1 :
    (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_FORWARD_CHAIN_T1 :
    (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_FC_T1 :
    (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ASM_FORWARD_CHAIN_T1 :
    (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ASM_FC_T1 :
    (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_ASM_FORWARD_CHAIN_T1 :
    (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
val ALL_ASM_FC_T1 :
    (THM -> THM list) -> (THM list -> TACTIC) -> THM list -> TACTIC;
```

Description These are tacticals which use theorems whose conclusions are implications, or from which implications can be derived, to reason forwards from the assumptions of a goal. (The tacticals with *FC* are aliases for the corresponding ones with *FORWARD_CHAIN*.)

The description of *fc_tac* should be consulted for the basic forward chaining algorithms used. The significance of the final argument and of the presence or absence of *ASM* and *ALL* in the name is exactly as for *fc_tac* and its relatives.

The tacticals allow variation of the canonicalisation function used to obtain implications from the argument theorems and of the tactic generating function used to process the theorems derived by the forward inference. The canonicalisation function to use is the first argument and the tactic generating function is the second. (Related tacticals with names ending in *T* rather than *T1* are also available for the simpler case when wants to use the same canonicalisation function as *fc_tac* and just to vary the tactic generating function.)

Examples If the theorem argument comprises only implications which are to be used without canonicalisation, one might use: *FC_T1 id_canon (MAP_EVERY strip_asm_tac)*.

If one has an instance of *t1* as an assumption and one wishes to use the bi-implication in a theorem of the form $\vdash t1 \Rightarrow (t2 \Leftrightarrow t3)$ for rewriting, one might use *FC_T1 id_canon rewrite_tac*.

See Also *fc_tac*, *asm_fc_tac*, *bc_tac*, *FC_T*.

SML
```
val f_thm_tac : THM -> TACTIC;
```

Description Prove a goal by using a theorem of the form $\Gamma \vdash F$.

Tactic

$$\frac{\{\ \Gamma 2\ \}\ t}{} \qquad f\_thm\_tac\ (\Gamma 1 \vdash F)$$

where the assumptions, $\Gamma 1$, of the theorem are contained in the assumptions, $\Gamma 2$, of the goal.

Errors
```
28021 ?0 does not have the form Γ ⊢ F
```

Uses In tactic programming, to use a theorem which shows that the assumptions are contradictory.

See Also *strip_asm_tac*.

SML

```
val GEN_INDUCTION_T : THM -> (THM -> TACTIC) -> TERM -> TACTIC;
val gen_induction_tac : THM -> TERM -> TACTIC;
```

Description These give general means for constructing an induction tactic from an induction principle formulated as a theorem. The term argument is the induction variable, which must be free in the conclusion of the goal to which the tactic is applied but not in the assumptions.

GEN_INDUCTION_T causes any inductive hypotheses (see below) to be passed to a tactic generating function.

gen_induction_tac thm is the same as *GEN_INDUCTION_T thm strip_asm_tac*.

The discussion below is for the tactic computed by the call *GEN_INDUCTION_T thm ttac y* applied to a goal with conclusion t.

The induction principle, *thm* has the form:

$\vdash \forall p \bullet a \Rightarrow \forall x \bullet p\ x$

E.g. the usual principle of induction for the natural numbers:

$\vdash \forall\ p \bullet\ p\ 0 \land (\forall\ n \bullet\ p\ n \Rightarrow p\ (n + 1)) \Rightarrow (\forall\ n \bullet\ p\ n)$

The induction tactic takes the following steps:

1. Use ∀-elimination on *thm*, (with the term ⌜$\lambda y \bullet t$⌝) and β-reduction to give an implicative theorem, $\vdash a' \Rightarrow t$ and use it to reduce the goal to a subgoal with conclusion a'.
2. Repeatedly apply *∧_tac* and then repeatedly apply *∀_tac*.
3. To any of the resulting subgoals whose principal connective corresponds to an an implication in *thm* apply *⇒_T ttac*. E.g., with the usual principle of induction for the natural numbers as formulated above *⇒_T ttac*is applied in the inductive step but not in the base case, even if the conclusion of the goal is an implication.

The tactic also renames bound variables so that names which begin with the name of the variable in the theorem now begin with the name of the induction variable passed to the tactic.

Errors

29021 ?0 does not have the form '⊢ ∀p•a ⇒ ∀x•p x'
29023 The type of ?0 is not an instance of ?1
29024 ?0 is not a variable
29025 ?0 appears free in the assumptions of the goal
29026 ?0 does not appear free in the conclusion of the goal

SML

```
val GEN_INDUCTION_T1 : THM -> (THM -> TACTIC) -> TACTIC;
val gen_induction_tac1 : THM -> TACTIC;
```

Description These give a means for constructing an induction tactic from an induction principle formulated as a theorem, in cases where the induction variable can be inferred from the form of the theorem and the goal. They are in other respects very like *GEN_INDUCTION_T* and *gen_induction_tac thm*, q.v.

The induction theorem must be a theorem of the form:

$\vdash \forall p \bullet a \Rightarrow \forall x \bullet t[p\ x/b]$

Where t contains at least one occurrence of x. For example,

$\vdash \forall\ p\ \bullet\ p\ \{\}\ \wedge\ (\forall\ a\ x\bullet\ a \in Finite\ \wedge\ p\ a\ \wedge\ \neg\ x \in a \Rightarrow p\ (\{x\} \cup a))$
$\quad \Rightarrow (\forall\ a\bullet\ a \in Finite \Rightarrow p\ a)$

(for which t is $a \in Finite \Rightarrow b$).

The induction tactic matches the conclusion, c, of the goal with t, uses the result to derive a theorem of the form $\vdash a' \Rightarrow c$ and then proceeds exactly like the corresponding induction tactic produced by *GEN_INDUCTION_T* and *gen_induction_tac thm* q.v.

Errors

29007 ?0 does not have the form '$\vdash\forall p{:}\tau\rightarrow BOOL\bullet a \Rightarrow \forall x\bullet t[p\ x/b]$'
(where ⌜x⌝ must also appear in ⌜t⌝ other than as an argument of ⌜p⌝)
29009 The conclusion of the goal cannot be rewritten in the form ?0
29014 The term ?0 which matches the induction variable is not a variable

SML

```
val GET_ASMS_T : (THM list -> TACTIC) -> TACTIC;
```

Description *GET_ASMS_T thmstac* is a tactic which applies *asm_rule* to each assumption of the goal, giving a list of theorems, *thms* say, and then acts as *thmstac thms*.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{thmstac\ (map\ asm\_rule\ \Gamma)\ (\{\ a1,...,an\ \}\ t)} \qquad GET\_ASMS\_T\ thmstac$$

Uses To use all the assumptions as theorems.

Errors As for *thmstac*.

SML

```
val GET_ASM_T : TERM -> (THM -> TACTIC) -> TACTIC;
```

Description *GET_ASM_T asm thmtac* is a tactic which checks that *asm* is in the assumption list and then acts as *thmtac(asm⊢asm)*.

Tactic

$$\frac{\{\ \Gamma,\ asm'\ \}\ t}{thmtac\ (asm \vdash asm)\ (\{\ \Gamma,\ asm'\ \}\ t)} \qquad GET\_ASM\_T\ ⌜asm⌝\ thmtac$$

where *asm* and *asm′* are α-convertible.

Uses To use an assumption as a theorem

Errors

9301 the term ?0 is not in the assumption list

SML
```
val GET_FILTER_ASMS_T : (TERM -> bool) ->
        (THM list -> TACTIC) -> TACTIC;
```

Description *GET_FILTER_ASMS_T pred thmstac* is a tactic which applies *asm_rule* to each assumption of the subgoal that satisfies *pred*, giving a list of theorems, *thms* say and then acts as *thmstac thms*.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{thmstac\ (map\ asm\_rule\ (\Gamma \cap pred))\ (\{\ \Gamma\ \}\ t)}$$

GET_FILTER_ASMS_T
pred
thmstac

Uses To use all the selected assumptions as theorems.

Errors As for *thmstac*.

SML
```
val GET_NTH_ASM_T : int -> (THM -> TACTIC) -> TACTIC;
```

Description *GET_NTH_ASM_T i thmtac* is a tactic which applies *asm_rule* to the *i*-th assumption of the goal, giving a theorem, *thm* say, and then acts as *thmtac thm*.

Assumptions are numbered *1, 2 . . .*, so that, e.g., *GET_NTH_ASM_T 1* is the same as *TOP_ASM_T*

Tactic

$$\frac{\{\ a1,\ ...,\ an\ \}\ t}{thmtac\ (asm\_rule\ [ai])\ (\{\ \Gamma\ \}\ t)}$$

GET_NTH_ASM_T
i
thmtac

Uses To use an assumption as a theorem, treating the assumption list as an array.

Errors
```
9303   the index ?0 is out of range
```

SML
```
val id_tac : TACTIC;
```

Description A tactic that always succeeds, having no effect. This is the identity for the tactical *THEN_T*.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{\ \Gamma\ \}\ t}$$

id_tac

Uses For constructing larger tactics.

SML
```
val ID_THEN : THM_TACTICAL;
```

Description This is the identity for the theorem tactical combinator *THEN_TTCL*.

```
(ID_THEN) thmtac = thmtac
```

Uses For use in programming theorem tacticals.

SML

```
val IF_T2 : (THM -> TACTIC) -> (THM -> TACTIC) -> TACTIC;
```

Description Reduce a conditional by applying tactic generating functions to the two cases for the selector.

Tactic

$$\frac{\{\ \Gamma\ \}\ if\ a\ then\ tt\ else\ et}{ttac1\{\ a,\ \Gamma\ \} \vdash tt;\ ttac2\{\ \neg a,\ \Gamma\ \} \vdash et} \qquad \begin{array}{l} IF\_T2 \\ ttac1\ \ ttac2 \end{array}$$

See Also *⇔_T*, *STRIP_CONCL_T*

Errors

28071 Goal is not of the form: { Γ } if a then tt else et

SML

```
val if_tac : TACTIC;
```

Description Reduce a conditional subgoal by performing a case split on the selector.

Tactic

$$\frac{\{\ \Gamma\ \}\ if\ a\ then\ tt\ else\ et}{\{strip\ a,\ \Gamma\ \}\ tt\ ;\ \ \{strip\ \neg a,\ \Gamma\ \}\ et} \qquad if\_tac$$

See Also *strip_tac*

Errors

28071 Goal is not of the form: { Γ } if a then tt else et

SML

```
val IF_THEN2 : (THM -> TACTIC) -> (THM -> TACTIC) ->
        (THM -> TACTIC);
```

Description A theorem tactical to apply given theorem tactics to the result of eliminating the conditional from a theorem with a conditional as its conclusion.

$IF\_THEN\ ttac\ (\Gamma \vdash if\ a\ then\ tt\ else\ et) = ttac1\ (a, \Gamma \vdash tt)\ THEN\ ttac2\ (\neg a, \Gamma \vdash et)$

The function is partially evaluated with only the theorem tactic and theorem arguments.

See Also *IF_THEN*, *STRIP_THM_THEN*

Errors

7012 ?0 is not of the form: ‘Γ ⊢ if tc then tt else te‘

SML

```
val IF_THEN : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical to apply a given theorem tactic to the result of eliminating the conditional from a theorem with a conditional as its conclusion.

$IF\_THEN\ ttac\ (\Gamma \vdash if\ a\ then\ tt\ else\ et) = ttac\ (\Gamma \vdash a \Rightarrow tt)\ THEN\ ttac\ (\Gamma \vdash \neg\ a \Rightarrow et)$

The function is partially evaluated with only the theorem tactic and theorem arguments.

See Also *IF_THEN2*, *STRIP_THM_THEN*

Errors

7012 ?0 is not of the form: ‘Γ ⊢ if tc then tt else te‘

SML

val **IF_T** : (*THM* −> *TACTIC*) −> *TACTIC*;

Description Reduce a conditional by applying a tactic generating function to the two cases for the selector.

Tactic

$$\frac{\{\ \Gamma\ \}\ if\ \ a\ \ then\ \ tt\ \ else\ \ et}{ttac\{\ a,\ \Gamma\ \} \vdash tt;\ ttac\{\ \neg a,\ \Gamma\ \} \vdash et} \qquad \begin{array}{l} IF\_T \\ ttac \end{array}$$

See Also *IF_T2*, *STRIP_CONCL_T*

Errors

28071 Goal is not of the form: { *Γ* } *if a then tt else et*

SML

val **intro_∀_tac** : (*TERM* ∗ *TERM*) −> *TACTIC*;
val **intro_∀_tac1** : *TERM* −> *TACTIC*;

Description Sometimes it is helpful to prove a goal by proving a more general conjecture has the goal as a special case. *intro_∀_tac* introduces a universal quantifier into the conclusion of a goal in order to do this.

Tactic

$$\frac{\{\ \Gamma\ \}\ t[t1/x]}{\{\ \Gamma\ \}\ \forall x \bullet t} \qquad intro\_\forall\_tac\ (t1,\ x)$$

where t is a term in which x appears free and where either $t1$ the same as x or x does not appear free in the conclusion, $t[t1/x]$, of the original goal.

Note that $t1$ need not be a variable, e.g.,

Example

$$\frac{\{\ \Gamma\ \}\ 1 + 2 > 0 \Rightarrow \neg 1 + 2 = 0}{\{\ \Gamma\ \}\ \forall i \bullet i > 0 \Rightarrow \neg i = 0} \qquad intro\_\forall\_tac\ (\ulcorner 1{+}2 \urcorner,\ \ulcorner i{:}\mathbb{N} \urcorner)$$

intro_∀_tac1 is for use in the common case where one simply wants to take replace the goal by its universal closure over some variable. *intro_∀_tac1* ⌜*x*⌝ is equivalent to *intro_∀_tac* (⌜*x*⌝, ⌜*x*⌝).

N.B. these tactics may strengthen the goal, i.e. they may result in unprovable subgoals even when the original goal was provable.

Uses The most common use is in preparation for an inductive proof when it is necessary to generalise the conclusion in order to give stronger assumptions in the inductive step or steps.

See Also *∀_reorder_conv*

Errors

28082 ?0 does not appear free in the goal
28083 ?0 appears free in the goal and is not the same as ?1

SML

val **i_contr_tac** : *TACTIC*;

Description Prove a goal by showing that the assumptions are contradictory, in an intuitionistic manner.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{\ \Gamma\ \}\ F} \qquad i\_contr\_tac$$

Uses If a proof is to be carried out by showing the assumptions inconsistent, then the conclusion of the subgoal is irrelevant and may be removed.

SML

val **lemma_tac** : *TERM* −> *TACTIC*;

Description Introduce a lemma (the term argument) to be proved, and then added as an assumption.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{\ \Gamma\ \}\ t2}{\{\ \Gamma\ \}\ t1;\quad \{strip\ t1,\ \Gamma\}\ t2} \qquad lemma\_tac\ \ulcorner t1 \urcorner$$

See Also *LEMMA_T*

Errors

9603 the term ?0 is not boolean

SML

val **LEMMA_T** : *TERM* −> (*THM* −> *TACTIC*) −> *TACTIC*;

Description *LEMMA_T newsg thmtac* is a tactic which sets *newsg* as a new subgoal and applies *thmtac*(*newsg*⊢*newsg*) to the original goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{\ \Gamma\ \}\ newsg;\quad thmtac(newsg \vdash newsg)\ (\{\ \Gamma\ \}\ t)} \qquad LEMMA\_T\ newsg\ thmtac$$

Uses For use in tactic programming and in interactive use where *lemma_tac* is not appropriate.

Errors

9603 the term ?0 is not boolean

See Also *lemma_tac*.

SML
```
val list_asm_ante_tac : TERM list -> TACTIC;
```

Description Repeatedly apply *asm_ante_tac*.

Rule

$$\frac{\{\ \Gamma,\ t1,\ ...,\ tn\ \}\ t}{\{\ \Gamma\ \}\ t1\ \Rightarrow\ ...\ \Rightarrow\ tn\ \Rightarrow\ t}\quad \begin{array}{l}list\_asm\_ante\_tac\\ [\ulcorner t1\urcorner,...,\ulcorner tn\urcorner]\end{array}$$

α-equivalent assumptions will only appear once in the resulting goal, in their rightmost position, (which also means that duplicates in the list are ignored).

See Also *asm_ante_tac*, *all_asm_ante_tac*

Errors

28052 Term ?0 is not in the assumptions
28055 The conclusion or an assumption of goal does not have type ⌜:BOOL⌝

SML
```
val LIST_DROP_ASM_T : TERM list -> (THM list -> TACTIC) -> TACTIC;
```

Description *LIST_DROP_ASM_T* [*asm_1*, *asm_2*, ...] *thmtac* is a tactic which removes the *asm_1*, *asm_2*, ... from the assumption list and then acts as

thmtac[(*asm_1*⊢*asm_1*), (*asm_2*⊢*asm_2*), ...]

Tactic

$$\frac{\{\ \Gamma,\ asm1',...\ \}\ t}{thmtac\ [(asm1\ \vdash\ asm1),...]\ (\{\ \Gamma\ \}\ t)}\quad \begin{array}{l}LIST\_DROP\_ASM\_T\\ [\ulcorner asm1\urcorner,...]\\ thmtac\end{array}$$

where *asm_i* and *asm_i′* are α-convertible.

Uses To use assumptions as theorems

Errors

9301 the term ?0 is not in the assumption list

SML
```
val LIST_DROP_NTH_ASM_T : int list ->
        (THM list -> TACTIC) -> TACTIC;
```

Description *LIST_DROP_NTH_ASM_T* [*i*, *j*, ...] *thmtac* is a tactic which applies *asm_rule* to the *i*-th, *j*-th, etc assumptions of the goal, giving theorems, *thm_i*, *thm_j*, etc, say, and then removes the *asm_i*, *asm_j* from the assumptions and acts as *thmtac* [*thm_i*, *thm_j*, ...].

Tactic

$$\frac{\{\ a1,\ ...,\ an\ \}\ t}{\begin{array}{c}thmtac\ [(asm\_rule\ [ai]),\\ (asm\_rule\ [aj]),...]\\ (\{\ \Gamma\ \setminus\ \{ai,\ aj,\ ...\}\}\ t)\end{array}}\quad \begin{array}{l}DROP\_NTH\_ASM\_T\\ [i,j,...]\\ thmtac\end{array}$$

Uses To use assumptions as theorems, treating the assumption list as an array.

Errors

9303 the index ?0 is out of range

SML
```
val LIST_GET_ASM_T : TERM list -> (THM list -> TACTIC) -> TACTIC;
```

Description *LIST_GET_ASM_T* [*asm_1*, *asm_2*, ...] *thmtac* is a tactic which checks that all the *asm_1*, *asm_2*, ... are in the assumption list and then acts as

thmtac[(*asm_1*⊢*asm_1*), (*asm_2*⊢*asm_2*), ...]

Tactic

$$\frac{\{\ \Gamma,\ asm1',\ ...\ \}\ t}{thmtac\ [(asm1\ \vdash\ asm1),...]\ (\{\ \Gamma,\ asm',...\ \}\ t)}$$

LIST_GET_ASM_T [⌜*asm1*⌝,...] *thmtac*

where *asm_i* and *asm_i′* are α-convertible.

Uses To use a list of assumptions as theorems

Errors

9301	*the term ?0 is not in the assumption list*

SML
```
val LIST_GET_NTH_ASM_T : int list -> (THM list -> TACTIC) -> TACTIC;
```

Description *LIST_GET_NTH_ASM_T* [*i*, *j*, ...] *thmtac* is a tactic which applies *asm_rule* to the *i*-th, *j*-th, etc, assumption of the goal, giving theorems, *thm_i*, *thm_j*, etc, say, and then acts as *thmtac* [*thm_i*, *thm_j*, ...].

Tactic

$$\frac{\{\ a1,\ ...,\ an\ \}\ t}{thmtac\ [(asm\_rule\ [ai]),...]\ (\{\ \Gamma\ \}\ t)}$$

LIST_GET_NTH_ASM_T [*i*,...] *thmtac*

Uses To use assumptions as theorems, treating the assumption list as an array.

Errors

9303	*the index ?0 is out of range*

SML
```
val list_simple_∃_tac : TERM list -> TACTIC ;
```

Description Provide a list of witnesses for an interated existential subgoal.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{\ \Gamma\ \}\ \exists\ x1,x2...\ \bullet\ t2[x1,x2,...]}{\{\ \Gamma\ \}\ t2[t1',t2'...]}$$

list_simple_∃_tac [⌜*t1*⌝,⌜*t2*⌝,...]

where $t1', t2', \ldots$ are $t1, t2, \ldots$, type instantiated to have the same type as $x1, x2, \ldots$.

See Also *simple_∃_tac*

Errors

29008	*Cannot match witness ?0 to variable ?1*
29015	*The list of witnesses is longer than the list of existentially quantified variables in ?0*
29016	*The list of witnesses is empty*
29017	*Goal is not of the form*: { Γ } ∃ x • t2[x]

SML

```
val list_swap_asm_concl_tac : TERM list -> TACTIC;
val list_swap_nth_asm_concl_tac : int list -> TACTIC;
```

Description Strip the negation of current goal into the assumption list and make some assumptions, suitably negated, into a disjunction forming the current goal. If the list is empty then the conclusion will become ⌜F⌝.

Tactic

$$\frac{\{\ \Gamma\ \}\ t2}{\{strip\ ⌜\neg\ t2⌝,\ \Gamma - \{⌜t1⌝,...,⌜tn⌝\}\}\ \ \neg\ t1 \lor ... \lor \neg\ tn}$$ *list_swap_asm_concl_tac* [⌜*t1*⌝,...,⌜*tn*⌝]

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{strip\ ⌜\neg\ t⌝,\ \Gamma - \{⌜tp⌝,\ ...,\ ⌜tq⌝\}\}\ \ \neg\ tp \lor ... \neg\ tq}$$ *list_swap_nth_asm_concl_tac* [*p*,...,*q*]

If any assumption is a negated term then the double negation will be eliminated.

See Also Other *swap* and *SWAP* functions.

Errors

9303 *the index ?0 is out of range*
28052 *Term ?0 is not in the assumptions*

SML

```
val LIST_SWAP_ASM_CONCL_T
        : TERM list -> (THM -> TACTIC) -> TACTIC;
val LIST_SWAP_NTH_ASM_CONCL_T
        : int list -> (THM -> TACTIC) -> TACTIC;
```

Description Process the negation of current goal with the supplied theorem tactic and make some assumptions, suitably negated, into a disjunction forming the current goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{ttac(asm\_rule\ ⌜\neg t⌝\ )\ (\{\Gamma - \{⌜tp⌝,...⌜tq⌝\}\}\ \neg\ t1)}$$ *LIST_SWAP_ASM_CONCL_T* [⌜*tp*⌝,...⌜*tq*⌝] *ttac*

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{ttac(asm\_rule\ ⌜\neg t⌝\ )\ (\{\Gamma - \{⌜tp⌝,\ ...,\ ⌜tq⌝\}\}\ \neg\ tm)}$$ *LIST_SWAP_NTH_ASM_CONCL_T* [*p*, ..., *q*] *ttac*

If an assumption is a negated term then the double negation will be eliminated. If the list is empty then the conclusion (before applying the tactic argument) will become ⌜F⌝.

See Also Other *swap* and *SWAP* functions.

Errors

9303 *the index ?0 is out of range*
28052 *Term ?0 is not in the assumptions*
28027 *Conclusion of goal does not have type ⌜:BOOL⌝*

SML
```
val MAP_EVERY_T : ( 'a -> TACTIC) -> 'a list -> TACTIC;
val MAP_EVERY : ( 'a -> TACTIC) -> 'a list -> TACTIC;
```

Description *MAP_EVERY_T mapf alist* maps *mapf* over *alist*, and then applies the resulting list of tactics to the goal in sequence (in the same manner as *EVERY*, q.v.). *MAP_EVERY* is an alias for *MAP_EVERY_T*.

Errors As the individual items generated by mapping the tactic over the list.

SML
```
val MAP_FIRST_T : ('a -> TACTIC) -> 'a list -> TACTIC;
val MAP_FIRST : ('a -> TACTIC) -> 'a list -> TACTIC;
```

Description *MAP_FIRST_T mapf alist* maps *mapf* over *alist*, and then attempts to apply each resulting tactic in order, until one succeeds or all fail (in the same manner as *FIRST*, q.v.). *MAP_FIRST* is an alias for *MAP_FIRST_T*.

Errors As the last tactic.

SML
```
val map_shape : (('a list -> 'b) * int) list -> 'a list -> 'b list
```

Description *map_shape* is a means of composing functions on lists. It is intended for composing the proofs produced by tactics in tacticals such as *THEN*. Its effect is as follows:

```
     map_shape [(f1, n1), (f2, n2)... ] [a1, a2, ...]
=    [f1[a1, ..., a(n1)], f2[a(n1+1), ..., a(n1+n2)], ...]
```

where, if there are not enough *a_i*, then unused *f_j* are ignored and the last *f_j* to be used may receive less than *n_j* elements in its argument. (This case is not expected to occur in the application of *map_shape* in tactic programming.)

Uses Specialised low-level tactic programming.

SML
```
val ORELSE_TTCL : (THM_TACTICAL * THM_TACTICAL) -> THM_TACTICAL;
```

Description *ORELSE_TTCL* is a theorem tactical combinator. It is an infix operator. (*tcl1 ORELSE_TTCL tcl2*)*th* is *tcl1 th* unless evaluation of *tcl1 th* fails, in which case it is *tcl2 th*.

Uses For use in programming theorem tacticals.

SML
```
val ORELSE_T : (TACTIC * TACTIC) -> TACTIC;
val ORELSE : (TACTIC * TACTIC) -> TACTIC;
```

Description *ORELSE_T* is a tactical used as an infix operator. *tac1 ORELSE_T tac2* is a tactic which behaves as *tac1* unless application of *tac1* fails, in which case it behaves as *tac2*. *ORELSE* is an alias for *ORELSE_T*

See Also *LIST_ORELSE_T*

Errors As the failure of *tac2*.

SML

```
val pair_rw_canon : CANON;
```

Description This is the rewrite canonicalisation function for the theory of pairs, defined as

```
val pair_rw_canon =
        REWRITE_CAN
        (REPEAT_CAN(FIRST_CAN [
        ∀_rewrite_canon,
        ∧_rewrite_canon,
        ¬_rewrite_canon,
        f_rewrite_canon,
        ⇔_t_rewrite_canon]));
```

This is the repeated application of the first applicable operation in the following list:

1. stripping universal quantifiers (paired or simple);
2. dividing conjunctive theorems into their conjuncts;
3. changing $\vdash \neg(t1 \lor t2)$ to $\neg t1 \land \neg t2$;
4. changing $\vdash \neg \exists vs \bullet t$ to $\forall vs \bullet \neg t$;
5. changing $\vdash \neg\neg t$ to $t \Leftrightarrow F$;
6. changing $\vdash \neg t$ to $t \Leftrightarrow F$;
7. if none of the above apply, changing $\vdash t$ to $\vdash t \Leftrightarrow T$.

Finally, after all this canonicalisation we then universally quantify the resulting theorems in all free variables other than those that were free in the original.

SML

```
val POP_ASM_T : (THM -> TACTIC) -> TACTIC;
```

Description *POP_ASM_T thmtac* is a tactic which removes the top entry, *asm* say, from the assumption list and then acts as $thmtac(asm \vdash asm)$.

Tactic

$$\frac{\{\ asm,\ \Gamma\ \}\ t}{thmtac\ (asm \vdash asm)\ (\{\ \Gamma\ \}\ t)} \qquad \begin{array}{l} POP\_ASM\_T \\ \ulcorner asm \urcorner \\ thmtac \end{array}$$

Uses To use an assumption as a theorem

Errors

```
9302   the assumption list is empty
```

SML

```
val prim_rewrite_tac : CONV NET -> CANON -> (THM -> TERM * CONV) OPT ->
        (CONV -> CONV) -> EQN_CXT -> THM list -> TACTIC;
```

Description This is the tactic based on *prim_rewrite_conv* (q.v.), with the same parameters as that function, except for the last argument:

Tactic

$$\frac{\{\ \Gamma\ \} \vdash t}{\{\ \Gamma\ \} \vdash t'}$$

prim_rewrite_tac
(*initial_net*: *CONV NET*)
(*canon* : *CANON*)
(*epp* : (*THM* -> *TERM* * *CONV*) *OPT*)
(*traverse* : *CONV* -> *CONV*)
(*with_eqn_cxt* : *EQN_CXT*)
(*with_thms* : *THM list*)

where ⌜t'⌝ is the result of rewriting ⌜t⌝ in the manner prescribed by the arguments.

SML

```
val prove_tac : THM list -> TACTIC;
```

Description This tactic is an automatic proof procedure appropriate to the current proof context.

At the point of applying this tactic to its theorems it will access the current setting of proof context field *pr_tac*, apply it to the theorem list immediately, and then to the goal, with its assumptions temporarily removed when available (i.e. the result is partially evaluated with only the list of theorems). The original assumptions will be returned to the resulting subgoals using *check_asm_tac*.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\begin{array}{c} current\_ad\_pr\_tac()thms(\{\},\ t) \\ THEN\ MAP\_EVERY\ check\_asm\_tac\ \Gamma \end{array}}$$

prove_tac
thms

See Also *PC_T1* to defer accessing the proof context until application to the goal; and, *asm_prove_tac* for the form that does react to the presence of assumptions.

Errors

```
51021 The current proof context was created in theory ?0 at a
      point now either not in scope, deleted or modified
```

and as the proof context setting.

It is possible that if *prove_tac* does not prove all its subgoals, then there may be an identification of newly introduced variables with free variables in the assumptions that were temporarily put to one side. This will lead to failures in the execution of the proof parts of the tactics that constitute the current proof context's automatic prover. Such a failure may not give particularly helpful messages concerning the cause of the failure. The problem is avoided by using *asm_prove_tac*, or by a call to *rename_tac* to change the offending variable names.

SML

|*val* **prove_thm** : (*string* ∗ *TERM* ∗ *TACTIC*) −> *THM*;

Description *prove_thm* (*key*, *gl*, *tac*) applies the tactic *tac* to the goal ([], *tm*), and, if the tactic succeeds in proving the goal saves the theorem under the key given, and returns the resulting theorem.

prove_thm performs α-conversion as necessary to ensure that the theorem returned has the same form as the specified goal. In circumstances where these adjustments are known not to be necessary, *simple_tac_proof* may be used to avoid the overhead.

Defn

|*prove_thm* (*key*, *tm*, *tac*) = *save_thm*(*key*, *tac_proof*(([],*tm*).*tac*));

Uses The subgoal package is the normal interactive mechanism for developing proofs using tactics. *prove_thm* is typically used in tactic programming and other proof procedures, in cases where it is necessary to ensure that the correct goal is proved and saved.

Errors

9501	*the tactic returned unsolved subgoals*: ?*0*
9502	*evaluation of the tactic failed*: ?*0*
9503	*the proof returned by the tactic failed*: ?*0*
9504	*the proof returned by the tactic proved* ?*0* *which could not be converted into the desired goal.*
9507	*the conclusion* ?*0* *is not of type* ⌜:*BOOL*⌝

See Also *simple_tac_proof*, *prove_thm*.

SML

|*val* **prove_∃_tac** : *TACTIC*;

Description This tactic is an automatic proof procedure for existential proofs, appropriate to the current proof context.

At the point of applying this tactic to a goal it will access the current setting of proof context field *prove_∃*, apply it to the goal, with its assumptions temporarily removed, using *conv_tac*. The original assumptions will be returned to the resulting subgoals using *check_asm_tac*.

Tactic

{ *Γ* } *t*	*prove_∃_tac*
═════════════	
conv_tac(*current_ad_cs_∃_conv*())({}, *t*) *THEN MAP_EVERY check_asm_tac Γ*	*thms*

See Also *asm_prove_∃_tac* that does react to any assumptions that are present.

Errors

51021	*The current proof context was created in theory* ?*0* *at a point now either not in scope*, *deleted or modified*

and as the proof context setting.

SML

val **rename_tac** : (*TERM* ∗ *string*)*list* −> *TACTIC*;

Description *rename_tac* renames variables (bound or free) in a goal. It is typically used when a goal contains several variables with the same name or to introduce names which are better mnemonics. For the latter purpose, the argument controls the algorithm used to make variants of the names.

The renaming affects both the conclusions and the assumptions of the goal. Variables are renamed to ensure that the new goal satisfies the following conditions:

- No two free variables with different types have the same name.
- No bound variable has the same name as a free variable or a variable which is bound in an outer scope.
- No variable shall have the same name as any constant in scope.

Before a variable is checked, it is looked up in the *renaming* association list, and if present it is treated as if the name were the corresponding string. The function *variant*, q.v., is used to rename variables.

The function may be partially evaluated with only the renamings argument.

Note that applying the tactic in the subgoal package will give rise to the message "The subgoal <label> is α-convertible to its goal".

For example,

Tactic

```
{ k = 1 }
(∀ i:ℕ×ℕ• ∃i:ℕ•i = 0)
∧ (∀ j:ℕ×ℕ
 • (∃k:ℕ•k = Fst j)
   ∧ ∀j:ℕ•j = k)                     rename_tac
                                     [(⌜j:ℕ×ℕ⌝,"apple"),
                                      (⌜j:ℕ⌝, "banana"),
                                      (⌜k:ℕ⌝, "carrot")]
=====================================
{ apple = 1 }
(∀ i• ∃ i′• i′ = 0)
∧ (∀ apple
   • (∃ carrot′• carrot′ = Fst apple)
     ∧ ∀ banana• banana = carrot)
```

Uses In clarifying goals where the variable names clash or are unparseable or are inconvenient.

Errors

3007 ?0 is not a term variable

SML

val **REPEAT_N_T** : *int* −> *TACTIC* −> *TACTIC*;
val **REPEAT_N** : *int* −> *TACTIC* −> *TACTIC*;

Description *REPEAT_N_T n* is a tactical which repeatedly applies its tactic argument *n* times. Unlike *REPEAT* it fails if the tactic fails. If *n* is not greater than 0 then *REPEAT_N_T n tac* is a tactic which has no effect.

REPEAT_N is an alias for *REPEAT_N_T*.

Errors As for the tactic being repeated.

SML
```
val REPEAT_TTCL : THM_TACTICAL -> THM_TACTICAL;
```

Description *REPEAT_TTCL ttcl* is a theorem tactical which applies *ttcl* repeatedly until it fails.

Uses For use in programming theorem tacticals. As for the argument theorem tactic.

SML
```
val REPEAT_T : TACTIC -> TACTIC;
val REPEAT : TACTIC -> TACTIC;
```

Description *REPEAT_T* is a tactical which repeatedly applies its tactic argument until it fails. This may cause an infinite loop of evaluation, or even no change, if the tactic fails on the first application. *REPEAT* is an alias for *REPEAT_T*.

SML
```
val REPEAT_UNTIL_T1 : (GOAL -> bool) -> TACTIC -> TACTIC;
val REPEAT_UNTIL1 : (GOAL -> bool) -> TACTIC -> TACTIC;
```

Description *REPEAT_UNTIL1_T1 p tac* is a tactical which repeatedly applies its *tac* until all outstanding subgoals either satisfy the predicate *p* or cause *tac* to fail.

If the goal already satisfies *p*, then *REPEAT_UNTIL1_T1 p tac* is a tactic which has no effect.

REPEAT_UNTIL1 is an alias for *REPEAT_UNTIL1_T1*.

Example
```
REPEAT_UNTIL1 (is_or o snd) strip_tac
```

will repeatedly apply *strip_tac* until all outstanding subgoals have disjunctive conclusions or cause *strip_tac* to fail.

SML
```
val REPEAT_UNTIL_T : (TERM -> bool) -> TACTIC -> TACTIC;
val REPEAT_UNTIL : (TERM -> bool) -> TACTIC -> TACTIC;
```

Description *REPEAT_UNTIL1_T p tac* is a tactical which repeatedly applies its *tac* until all outstanding subgoals either have conclusions which satisfy the predicate *p* or cause *tac* to fail.

If the conclusion of the goal already satisfies *p*, then *REPEAT_UNTIL1_T1 p tac* is a tactic which has no effect.

REPEAT_UNTIL is an alias for *REPEAT_UNTIL_T*.

Example
```
REPEAT_UNTIL is_or strip_tac
```

will repeatedly apply *strip_tac* until all outstanding subgoals have disjunctive conclusions or cause *strip_tac* to fail.

SML
```
val rewrite_tac : THM list -> TACTIC;
val pure_rewrite_tac : THM list -> TACTIC;
val once_rewrite_tac : THM list -> TACTIC;
val pure_once_rewrite_tac : THM list -> TACTIC;
val asm_rewrite_tac : THM list -> TACTIC;
val pure_asm_rewrite_tac : THM list -> TACTIC;
val once_asm_rewrite_tac : THM list -> TACTIC;
val pure_once_asm_rewrite_tac : THM list -> TACTIC;
```

Description These are the rewriting tactics. They use the canonicalisation rule held by the current proof context (see, e.g., *push_pc*) to preprocess the theorem list. The context is accessed at the point when the rules are given a list of theorems.

If a tactic is "pure" then there is no default rewriting, otherwise the default rewriting conversion net held by the current proof context will be used in addition to user supplied material.

If a tactic is "once" then rewriting will proceed from the root of the of the conclusion of the theorem to be rewritten, towards the leaves, and will not descend through any rewritten subterm, using *ONCE_MAP_WARN_C*. If not, rewriting will continue, moving from the root to the leaves, repeating if any rewriting is successful, until there is no rewriting redex anywhere within the rewritten conclusion, using *REWRITE_MAP_C*. This may cause non-terminating looping.

If a tactic is "asm" then the theorems rewritten with will include the canonicalised *asm_rule*d assumptions of the goal.

Errors
```
26001 no rewriting occurred
```

Also as error 26003 and warning 26002 of *REWRITE_MAP_C* (q.v.).

SML
```
val rewrite_thm_tac : THM -> TACTIC;
val pure_rewrite_thm_tac : THM -> TACTIC;
val once_rewrite_thm_tac : THM -> TACTIC;
val pure_once_rewrite_thm_tac : THM -> TACTIC;
val asm_rewrite_thm_tac : THM -> TACTIC;
val pure_asm_rewrite_thm_tac : THM -> TACTIC;
val once_asm_rewrite_thm_tac : THM -> TACTIC;
val pure_once_asm_rewrite_thm_tac : THM -> TACTIC;
```

Description These are rewriting tactics parameterised to take only one theorem. This parameterisation is convenient to use with the many tactic generating functions, such as *LEMMA_T*, which take a theorem tactic as an argument.

See, e.g. *rewrite_tac* for the details of the differences between these tactics.

Errors
```
26001 no rewriting occurred
```

Errors will be reported as if they are from the corresponding *_tac*: e.g. from *rewrite_tac* rather than *rewrite_thm_tac*. This allows a simple implementation, and for there to be no functionality change even in errors between using singleton lists with the originals, and these functions. The following warning indicates the result of, perhaps only some, of the rewriting was discarded.

Errors
```
26002 rewriting gave ill-formed results on some subterms
```

SML

val **ROTATE_T** : *int* −> *TACTIC* −> *TACTIC*;

Description *ROTATE_T i tac* is a tactic which first applies *tac* and, if this does not achieve the goal, rotates the resulting subgoals by *i* places. *i* is taken modulo the number of subgoals produced by *tac*.

Thus if the result of *tac* is:

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{\ \Gamma 1\ \}\ t1;\ ...\ \{\ \Gamma k\ \}\ tk} \qquad tac$$

then the result of *ROTATE_T i t* will be:

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{\ \Gamma(i{+}1)\ \}\ t(i{+}1);\ ...,\ \{\ \Gamma k\ \}\ tk;\ \{\ \Gamma 1\ \}\ t1;\ ...\ \{\ \Gamma i\ \}\ ti} \qquad ROTATE\_T\ i\ tac$$

Uses For use in tactic programming to handle tactics which return their subgoals in an inconvenient order for the task at hand.

Errors As for *tac*.

SML

val **simple_tac_proof** : (*GOAL* ∗ *TACTIC*) −> *THM*;

Description *simple_tac_proof*(*gl*, *tac*) applies the tactic *tac* to the goal *gl*, and, if the tactic returns no unsolved subgoals returns the theorem proved by the tactic.

Infelicities in the coding of the tactic may cause the theorem returned to be rather different from the specified goal (in general, a "successful" application of a correctly coded tactic will return a theorem which may require addition of assumptions and α-conversion to give the desired goal). *tac_proof* should be used rather than *simple_tac_proof* if it is important that the theorem should achieve the goal precisely.

Uses In programming tactics or other proof procedures where speed is important and the extra care taken by *tac_proof* is not required.

Errors

9501 the tactic returned unsolved subgoals: ?0
9502 evaluation of the tactic failed: ?0
9503 the proof returned by the tactic failed: ?0

See Also *tac_proof*

SML

val **simple_taut_tac** : *TACTIC*;

Description A tautology prover. If the conclusion of the goal is a tautology then *taut_tac* will prove the goal. A tautology is taken to be any substitution instance of a term which is formed from boolean variables, the constants T and F and the following connectives:

$\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, $\neg$, *if ... then ... else*

and which is true for any assignment of truth values to the variables.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{} \qquad simple\_taut\_tac$$

See Also *strip_tac*

Errors

28121 Conclusion of the goal is not a tautology

SML

val **simple_¬_in_conv** : *CONV*;

Description This is a conversion which moves negations inside other predicate calculus connectives using whichever of the following rules applies:

$$\begin{array}{lcl}
\neg\neg t & = & t \\
\neg(t1 \wedge t2) & = & \neg t1 \vee \neg t2 \\
\neg(t1 \vee t2) & = & \neg t1 \wedge \neg t2 \\
\neg(t1 \Rightarrow t2) & = & t1 \wedge \neg t2 \\
\neg(t1 \Leftrightarrow t2) & = & (t1 \wedge \neg t2) \vee (t2 \wedge \neg t1) \\
\neg(if\ a\ then\ t1\ else\ t2) & & \\
 & = & (if\ a\ then\ \neg t1\ else\ \neg t2) \\
\neg\forall x\bullet t & = & \exists x\bullet\neg t \\
\neg\exists x\bullet t & = & \forall x\bullet\neg t \\
\neg\exists_1 x\bullet t = & \forall x\bullet\neg(t \wedge \forall x'\bullet t[x'] \Rightarrow x' = x) & \\
\neg T & = & F \\
\neg F & = & T
\end{array}$$

It does not handle paired quantifiers.

Uses Tactic and conversion programming. The more general *¬_in_conv* is just as efficient as *simple_¬_in_conv* in cases where both succeed.

See Also *strip_tac*

Errors

28131 No applicable rules for the term ?0

SML
```
val simple_¬_in_tac : TACTIC;
```

Description This is a tactic which moves negations inside other predicate calculus connectives using the following rules:

$\neg\neg t$	$\rightarrow$	t
$\neg(t1 \wedge t2)$	$\rightarrow$	$\neg t1 \vee \neg t2$
$\neg(t1 \vee t2)$	$\rightarrow$	$\neg t1 \wedge \neg t2$
$\neg(t1 \Rightarrow t2)$	$\rightarrow$	$t1 \wedge \neg t2$
$\neg(t1 \Leftrightarrow t2)$	$\rightarrow$	$(t1 \wedge \neg t2) \vee (t2 \wedge \neg t1)$
$\neg\forall x \bullet t$	$\rightarrow$	$\exists x \bullet \neg t$
$\neg\exists x \bullet t$	$\rightarrow$	$\forall x \bullet \neg t$
$\neg\exists_1 x \bullet t \rightarrow$	$\forall x \bullet \neg(t \wedge \forall x' \bullet t[x'] \Rightarrow x' = x)$	
$\neg T$	$\rightarrow$	F
$\neg F$	$\rightarrow$	*goal solved*

It fails with paired quantifiers.

Uses The more general *¬_in_tac* is just as efficient as *simple_¬_in_tac* in cases where both succeed.

See Also *strip_tac*, *contr_tac*, *¬_T*, *¬_in_tac*

Errors
```
28025 No applicable rule for this goal
```

SML
```
val SIMPLE_¬_IN_THEN : THM_TACTICAL;
```

Description This is a theorem tactical which applies a given theorem tactic to the result of transforming a theorem by moving a top level negation inside other predicate calculus connectives using the following rules:

$\neg\neg t$	$\rightarrow$	t
$\neg(t1 \wedge t2)$	$\rightarrow$	$\neg t1 \vee \neg t2$
$\neg(t1 \vee t2)$	$\rightarrow$	$\neg t1 \wedge \neg t2$
$\neg(t1 \Rightarrow t2)$	$\rightarrow$	$t1 \wedge \neg t2$
$\neg(t1 \Leftrightarrow t2)$	$\rightarrow$	$(t1 \wedge \neg t2) \vee (t2 \wedge \neg t1)$
$\neg\forall x \bullet t$	$\rightarrow$	$\exists x \bullet \neg t$
$\neg\exists x \bullet t$	$\rightarrow$	$\forall x \bullet \neg t$
$\neg\exists_1 x \bullet t \rightarrow$	$\forall x \bullet \neg(t \wedge \forall x' \bullet t[x'] \Rightarrow x' = x)$	
$\neg T$	$\rightarrow$	F
$\neg F$	$\rightarrow$	T

The function may be partially evaluated with only its theorem tactic and theorem arguments. It fails with paired quantifiers.

Uses The more general *¬_IN_THEN* is just as efficient as *SIMPLE_¬_IN_THEN* in cases where both succeed.

See Also *strip_tac*, *STRIP_THM_THEN*

Errors
```
28026 No applicable rule for this theorem
```

SML

val **simple_∀_tac** : *TACTIC*;

Description Reduce a universally quantified goal. It fails with paired quantifiers.

Tactic

$$\frac{\{\ \Gamma\ \}\ \forall\ x \bullet t[x]}{\{\ \Gamma\ \}\ t[x']} \qquad \textit{simple\_}\forall\textit{\_tac}$$

where x' is a variant name of x, different from any variable in Γ or t.

Uses Tactic programming. The more general *∀_tac* is just as efficient as *simple_∀_tac* in cases where both succeed.

See Also *∀_tac*

Errors

28081 Goal is not of the form: { Γ } ∀ x • $t[x]$

SML

val **simple_∃_tac** : *TERM* −> *TACTIC* ;

Description Provide a witness for an existential subgoal. It fails with paired quantifiers.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{\ \Gamma\ \}\ \exists\ x \bullet t2[x]}{\{\ \Gamma\ \}\ t2[t1]} \qquad \textit{simple\_}\exists\textit{\_tac}\ \ulcorner t1 \urcorner$$

where *t1* must have the same type as x.

Uses Tactic programming. The more general *∃_tac* is just as efficient as *simple_∃_tac* in cases where both succeed.

Errors

28091 Goal is not of the form: { Γ } ∃ x • $t2[x]$
28092 Term ?0 has the wrong type

SML
```
val SIMPLE_∃_THEN : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical which applies a given theorem tactic to the result of eliminating the outermost quantifier from a theorem of the form $\Gamma \vdash \exists x \bullet t$. It fails with paired quantifiers.

$$SIMPLE\_\exists\_THEN\ thmtac\ (\Gamma \vdash \exists x \bullet t) = thmtac\ (\Gamma \vdash t[x'/x])$$

where ⌜x'⌝ is a variant of ⌜x⌝ which does not appear in Γ or in the assumption or conclusion of the goal. The function is partially evaluated with only the theorem tactic and theorem arguments.

Uses Tactic programming. Note that the more general *∃_THEN* is just as efficient as *SIMPLE_∃_THEN* in cases where both succeed.

Error 28094 normally arises when ⌜x'⌝ is also introduced by the proof of *ttac*, and occurs during the application of the proof of *SIMPLE_∃_THEN*. The bound variable ⌜x⌝ should be renamed to something that doesn't cause this identification of distinct variables, by using *rename_tac*(q.v.).

See Also *∃_THEN*

Errors
```
28093 ?0 is not of the form: ‘Γ ⊢ ∃ x • t‘
28094 Error in proof of SIMPLE_∃_THEN.
      Usually indicates chosen skolem variable ?0 also
      introduced by proof of supplied theorem tactic,
      which gave ‘?1‘, and the two became identified:
      use rename_tac to rename original bound variable ?2
```

SML
```
val simple_∃₁_conv : CONV;
```

Description This is a conversion which turns a unique existential quantifier into an equivalent existential quantifier

Conversion

$$\frac{}{\vdash (\exists_1 x \bullet t[x]) \Leftrightarrow (\exists x \bullet t[x] \land \forall x' \bullet t[x'] \Rightarrow x' = x)} \quad \begin{array}{l} simple\_\exists_1\_conv \\ \ulcorner \exists_1 x \bullet t \urcorner \end{array}$$

Uses Tactic and conversion programming. The more general *∃_1_conv* is just as efficient as *simple_∃_1_conv* in cases where both succeed.

See Also *strip_tac*

Errors
```
4019 ?0 is not of form: ⌜∃₁ v• t⌝
```

SML
```
val simple_∃1_tac : TERM -> TACTIC;
```

Description Simplify a unique existentially quantified goal with a particular witness. It fails with paired quantifiers.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{\ \Gamma\ \}\ simple\_\exists_1\ x \bullet P[x]}{\{\ \Gamma\ \}\ P[t];\quad \{\ \Gamma\ \}\ \forall\ x' \bullet P[x'] \Rightarrow x' = t} \qquad simple\_\exists_1\_tac1\ \ulcorner t \urcorner$$

where x' is a variant of x which does not occur free in t.

Uses Tactic programming. The more general *∃_1_tac* is just as efficient as *simple_∃_1_tac* in cases where both succeed.

Errors

28101 Goal is not of the form: $\{\ \Gamma\ \}\ \exists_1\ x \bullet P[x]$
28092 Term ?0 has the wrong type

SML
```
val SIMPLE_∃1_THEN : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical which applies a given theorem tactic to the result of eliminating the outermost quantifier from a theorem of the form $\Gamma \vdash \exists\_1x \bullet t$. It fails with paired quantifiers.

$$SIMPLE\_\exists_1\_THEN\ thmtac\ (\Gamma \vdash \exists_1 x \bullet t) = \\ \quad thmtac\ (\Gamma \vdash t[x'/x] \wedge \forall x'' \bullet P[x''] \Rightarrow x'' = x)$$

where ⌜x'⌝ and ⌜x''⌝ are distinct variants of ⌜x⌝ which do not appear free in Γ or in the assumptions or conclusion of the goal.

Uses Tactic programming. The more general *∃₁_THEN* is just as efficient as *SIMPLE-_∃₁_THEN* in cases where both succeed.

Errors

28102 ?0 is not of the form: ‘$\Gamma \vdash \exists_1\ x \bullet t$‘

SML
```
val SOLVED_T : TACTIC -> TACTIC;
```

Description *SOLVED_T tac* is a tactic which applies *tac* to the goal and fails if it does not solve the goal. I.e. it fails unless the tactic returns an empty list of subgoals.

SOLVED_T does not check that the proof delivered by the tactic is valid. *tac_proof* may be used to achieve this type of effect.

Uses Tactic programming, for when a tactic that fails to prove a goal is likely to leave an untidy goal state.

See Also *tac_proof*

Errors

9602 the tactic did not solve the goal

SML

```
val spec_asm_tac : TERM -> TERM -> TACTIC;
val list_spec_asm_tac : TERM -> TERM list -> TACTIC;
val spec_nth_asm_tac : int -> TERM -> TACTIC;
val list_spec_nth_asm_tac : int -> TERM list -> TACTIC;
```

Description These are four methods of specialising assumptions, differing by single or lists of values to specialise to, and in the method of selection of the assumption. All of them leave the old assumption in place, and place the instantiated assumption onto the assumption list using *strip_asm_tac*. If the desired behaviour differs from any of those supplied then use *GET_ASM_T* and its cousins to create the desired functionality.

Tactic

```
{ Γ, ⌜∀ vs[x1′,...] • f [x1′,...]⌝ } t              spec_asm_tac
═══════════════════════════════════════              ⌜∀ vs[x1,...] • f [x1,...]⌝
      {strip ⌜f [t1,...]⌝, Γ,                         ⌜tm⌝
   ⌜∀ vs[x1,...] • f [x1,...]⌝} t1
```

The following all handle paired abstractions in a similar manner.

Tactic

```
  { Γ, ⌜∀ x1 ... • f [x1,...]⌝ } t                  list_spec_asm_tac
═══════════════════════════════════════              ⌜∀ x1 ... • f [x1,...]⌝
      {strip ⌜f [t1,...]⌝, Γ,                         [⌜t1⌝,⌜t2⌝,...]
    ⌜∀ x1 x2 ... • f [x1,...]⌝,
               } t
```

Tactic

```
{ Γ1...n−1, ⌜∀ x′ • f [x′]⌝, Γ} t1                  spec_nth_asm_tac
═══════════════════════════════════════              n
     {strip ⌜f [t2]⌝,  Γ1...n−1,                      ⌜t2⌝
          ⌜∀ x′ • f [x′]⌝,
              Γ} t1
```

Tactic

```
{ Γ1...n−1, ⌜∀ x1 ...• f [x1,...]⌝, Γ} t            list_spec_nth_asm_tac
═══════════════════════════════════════              n
    {strip ⌜f [t1,...]⌝, Γ1...n−1,                    [⌜t1⌝,...]
       ⌜∀ x1 ...• f [x1,...]⌝,
              Γ} t
```

Definitions

```
fun spec_asm_tac asm instance =
        GET_ASM_T asm (strip_asm_tac o ∀_elim instance);
fun list_spec_asm_tac asm instances =
        GET_ASM_T asm (strip_asm_tac o list_∀_elim instances);
fun spec_nth_asm_tac n instance =
        GET_NTH_ASM_T n (strip_asm_tac o ∀_elim instance);
fun list_spec_nth_asm_tac n instances =
        GET_NTH_ASM_T n (strip_asm_tac o list_∀_elim instances);
```

Errors As the constituents of the implementing functions.

SML

```
val SPEC_ASM_T : TERM -> TERM -> (THM -> TACTIC) -> TACTIC;
val LIST_SPEC_ASM_T : TERM -> TERM list -> (THM -> TACTIC)
        -> TACTIC;
val SPEC_NTH_ASM_T : int -> TERM -> (THM -> TACTIC) -> TACTIC;
val LIST_SPEC_NTH_ASM_T : int -> TERM list -> (THM -> TACTIC)
        -> TACTIC;
```

Description These are four methods of specialising assumptions, differing by single or lists of values to specialise to, and in the method of selection of the assumption. All of them leave the old assumption in place, and place the instantiated assumption onto the assumption list using their theorem tactic. If the desired behaviour differs from any of those supplied then use *GET_ASM_T* and its cousins to create the desired functionality.

Tactic

$$\frac{\{\ \Gamma,\ \ulcorner\forall\ vs[x1',...]\bullet f\ [x1',...]\urcorner\ \}\ t}{\begin{array}{c}thm\_tac\ (asm\_rule\ \ulcorner f\ [t1,...]\urcorner)\\ \{\Gamma,\ulcorner\forall\ vs[x1,...]\bullet f\ [x1,...]\urcorner\}\ t1\end{array}}$$

SPEC_ASM_T
⌜∀ vs[x1,...] • f [x1,...]⌝
⌜tm⌝
thm_tac

The following all handle paired abstractions in a similar manner.

Tactic

$$\frac{\{\ \Gamma,\ \ulcorner\forall\ x1\ x2\ ...\bullet f\ [x1,x2,...]\urcorner\ \}\ t}{\begin{array}{c}thm\_tac\ (asm\_rule\ \ulcorner f\ [t1,t2,...]\urcorner)\\ \{\Gamma,\ulcorner\forall\ x1\ x2\ ...\bullet f\ [x1,x2,...]\urcorner\}\ t\end{array}}$$

LIST_SPEC_ASM_T
⌜∀ x1 x2 ... • f [x1,x2,...]⌝
[⌜t1⌝,⌜t2⌝,...]
thm_tac

Tactic

$$\frac{\{\ \Gamma 1...n{-}1,\ \ulcorner\forall\ x'\bullet f\ [x']\urcorner,\ \Gamma\}\ t1}{\begin{array}{c}thm\_tac\ (asm\_rule\ \ulcorner f\ [t2]\urcorner)\\ \{\Gamma 1...n{-}1,\ulcorner\forall\ x'\bullet f\ [x']\urcorner,\Gamma\}\ t1\end{array}}$$

SPEC_NTH_ASM_T
n
⌜t2⌝
thm_tac

Tactic

$$\frac{\{\ \Gamma 1...n{-}1,\ \ulcorner\forall\ x1\ ...\bullet f\ [x1,...]\urcorner,\ \Gamma\}\ t}{\begin{array}{c}thm\_tac\ (asm\_rule\ \ulcorner f\ [t1,...]\urcorner)\\ \{\Gamma 1...n{-}1,\ulcorner\forall\ x1\ ...\bullet f\ [x1,...]\urcorner,\ \Gamma\}\\ t\end{array}}$$

LIST_SPEC_NTH_ASM_T
n
[⌜t1⌝,...]
thm_tac

Definitions

```
fun SPEC_ASM_T asm instance thmtac =
        GET_ASM_T asm (thmtac o ∀_elim instance);
fun LIST_SPEC_ASM_T asm instances thmtac =
        GET_ASM_T asm (thmtac o list_∀_elim instances);
fun SPEC_NTH_ASM_T n instance thmtac =
        GET_NTH_ASM_T n (thmtac o ∀_elim instance);
fun LIST_SPEC_NTH_ASM_T n instances thmtac =
        GET_NTH_ASM_T n (thmtac o list_∀_elim instances);
```

Errors As the constituents of the implementing functions.

SML

```
val step_strip_tac : TACTIC;
val step_strip_asm_tac : THM -> TACTIC;
```

Description These functions provide methods of single-stepping through the application of *strip_tac* and *strip_asm_tac* (q.v.).

When stripping the antecedent of an implication ,or a theorem, into the assumption list *strip_tac* and *strip_asm_tac* respectively do all their stripping in one application of the tactic. This is not appropriate behaviour when:

1. Explaining the detailed behaviour of these functions by example applications.
2. Attempting to "debug" a failed or inappropriate stripping.
3. When a partial strip into the assumption list is desired.

The two functions provided give a single-step stripping of antecedents and theorems. They represent sets of objects that are partially stripped into the assumption list by making the conclusion of the resulting goal an implication with the antecedent being the conjunction of the partially stripped objects and the consequent being the unstripped part of the goal. Repeated use of the provided functions closely corresponds to the processing order and effect of *strip_tac* and *strip_asm_tac*. Under certain unusual circumstances the match may not be exact.

Example

$\vdash ?\ ((a \lor b) \land c) \Rightarrow ((a \land c) \lor (b \land c))$
Single steps to:
$\vdash ?\ (a \land c) \Rightarrow ((a \land c) \lor (b \land c))$
and $\vdash ?\ (b \land c) \Rightarrow ((a \land c) \lor (b \land c))$
Each single step to:
$a \vdash ?\ c \Rightarrow ((a \land c) \lor (b \land c))$
and $b \vdash ?\ c \Rightarrow ((a \land c) \lor (b \land c))$
Each single step to:
$a,\ c \vdash ?\ \Rightarrow ((a \land c) \lor (b \land c))$
and $b,\ c \vdash ?\ \Rightarrow ((a \land c) \lor (b \land c))$

These five steps (two on each branch) map onto one call of *strip_tac*.

Errors

28003 There is no stripping technique for ?0 in the current proof context

SML

val **strip_asm_tac** : *THM −> TACTIC*;

Description *strip_asm_tac* is a general purpose tactic for splitting a theorem up into useful pieces using a range of simplification techniques, including a parameterised part, before using it to increase the stock of assumptions.

First, before attempting to use the transformations below, *strip_asm_tac* uses the current proof context's theorem stripping conversion to attempt to rewrite the outermost connective in the theorem.

Then the following simplification techniques will be tried. Using *sat* as an abbreviation for *strip_asm_tac*:

$sat\ (\vdash a \land b) \quad \to \quad sat\ (\vdash a)\ THEN\ sat\ (\vdash b)$
$sat\ (\exists x \bullet a) \quad \to \quad sat\ (a[x'/x] \vdash a[x'/x])$
$sat\ (\vdash a \lor b)(\{\Gamma\}\ t) \quad \to \quad sat\ (a \vdash a)\ (\{\Gamma\}\ t)\ ;\ sat\ (b \vdash b)\ (\{\Gamma\}\ t)$

I.e. *strip_asm_tac* does a case split resulting in two subgoals when it processes a disjunction.

After all of the available simplification techniques have been attempted *strip_asm_tac* then proceeds as *check_asm_tac* (q.v.) to use the simplified theorem either to prove the goal or to generate additional assumptions.

See Also *STRIP_THM_THEN*, used to implement this function. *check_asm_tac*, *strip_tac*, *strip_asm_conv*.

SML

val **strip_concl_conv** : *CONV*;
val **strip_asm_conv** : *CONV*;

Description *strip_concl_conv tm*; applies the conclusion stripping conversion from the current proof context, to rewrite the outermost connective in the term *tm*.

strip_asm_conv tm; applies the assumption stripping conversion from the current proof context, to rewrite the outermost connective in the term *tm*.

Errors

28003 There is no stripping technique for ?0 in the current proof context

SML

val **strip_concl_tac** : *TACTIC*;
val **strip_tac** : *TACTIC*;

Description *strip_concl_tac*, more usually known by its alias, *strip_tac*, is a general purpose tactic for simplifying away the outermost connective of a goal. It first tries to apply the conclusion stripping conversion from the current proof context, to rewrite the outermost connective in the goal. If that conversion fails, tries to simplify the goal by applying an applicable member of the following collection of tactics (only one could possibly apply):

simple_∀_tac, *∧_tac*,
⇒_T strip_asm_tac, *t_tac*

Failing either being successful, it tries *concl_in_asms_tac* to prove the goal, and failing that, returns the error message below.

Note how new assumptions generated by the tactic are processed using *strip_asm_tac*, which uses the current proof context's theorem stripping conversion. *strip_tac* may produce several new subgoals, or may prove the goal.

REPEAT strip_tac in the proof context "basic_hol" (amongst others) will prove all tautologies automatically. It will, however, not succeed in proving some substitution instances of tautologies involving positive and negative instances of a quantified subterm.

Uses This is the usual way of simplifying a goal involving predicate calculus connectives, and other functions "understood" by the current prof context.

See Also *STRIP_T* and *STRIP_THM_THEN* which are used to implement this function. *taut_tac* for an alternative simplifier. *swap_∨_tac* to rearrange the conclusion for tailored stripping. Also *strip_concl_conv*, *strip_asm_conv*.

Errors

28003 There is no stripping technique for ?0 in the current proof context

SML

val **STRIP_CONCL_T** : (*THM* −> *TACTIC*) −> *TACTIC*;
val **STRIP_T** : (*THM* −> *TACTIC*) −> *TACTIC*;

Description *STRIP_CONCL_T ttac* is a general purpose way of stripping goals and passing any new assumptions generated by the stripping to a tactic generating function, *ttac*. *STRIP-_CONCL_T* attempts to apply the conversion held for it in the current proof context to rewrite the goal. The conversion is extracted from the current proof context by *current_ad_sc_conv*. If that fails it attempts to apply one of the following list of tactics (in order):

simple_∀_tac, *∧_tac*, *⇒_T ttac*, *t_tac*

If none of the above apply it tries *concl_in_asms_tac*, and failing that, return the error message below.

The conversion in the current proof context held by *current_ad_sc_conv* (q.v.) is derived by applying *eqn_cxt_conv* to an equational context in the proof context, extracted by *get_sc_eqn_cxt*.

STRIP_T is an alias for *STRIP_CONCL_T*.

Uses Tactic programming.

See Also *strip_asm_tac*, *strip_tac*, *strip_concl_conv*.

Errors

28003 There is no stripping technique for ?0 in the current proof context

SML

val **STRIP_THM_THEN** : *THM_TACTICAL*;

Description *STRIP_THM_THEN* provides a general purpose way of stripping theorems into primitive constituents before using them in a tactic proof. *STRIP_THM_THEN* attempts to apply the conversion held for the function in the current proof context, which is extracted by *current_ad_st_conv*. to rewrite the theorem. If that fails it attempts to apply a theorem tactical from the following list (in order):

$\wedge$_*THEN*, $\vee$_*THEN*, *SIMPLE*_$\exists$_*THEN*

The conversion in the current proof context got by *current_ad_st_conv* (q.v.) is derived by applying *eqn_cxt_conv* to an equational context in the proof context extracted by *get_st_eqn_cxt*.

The function is partially evaluated with only the theorem tactic and theorem arguments.

Uses Tactic programming.

See Also *strip_asm_tac*, *strip_tac*.

Errors

28003 There is no stripping technique for ?0 in the current proof context

SML

val **swap_asm_concl_tac** : *TERM* −> *TACTIC*;
val **swap_nth_asm_concl_tac** : *int* −> *TACTIC*;

Description Strip the negation of current goal into the assumption list and make an assumption, suitably negated, into the current goal. If the simplifications it does are ignored, *swap_asm_concl_tac asm*is equivalent to

Example

contr_tac THEN asm_ante_tac asm

and *swap_nth_asm_concl_tac n*is equivalent to

Example

contr_tac THEN DROP_NTH_ASM_T n ante_tac

Tactic

$$\frac{\{\ \Gamma,\ \ulcorner t1 \urcorner\ \}\ t2}{\{strip\ \ulcorner \neg\ t2 \urcorner,\ \Gamma\}\ \neg\ t1} \qquad swap\_asm\_concl\_tac\ \ulcorner t1 \urcorner$$

Tactic

$$\frac{\{\ \ulcorner t1 \urcorner,\ ...,\ \ulcorner tm \urcorner,...,\ \ulcorner tn \urcorner\}\ t}{\{strip\ \ulcorner \neg\ t \urcorner,\ \ulcorner t1 \urcorner,...,\ \ulcorner tn \urcorner\}\ \neg\ tm} \qquad swap\_nth\_asm\_concl\_tac\ m$$

If the assumption is a negated term then the double negation will be eliminated.

See Also Other *swap* and *SWAP* functions.

Errors

9303 the index ?0 is out of range
28052 Term ?0 is not in the assumptions

SML

val **SWAP_ASM_CONCL_T** : *TERM* −> (*THM* −> *TACTIC*) −> *TACTIC*;
val **SWAP_NTH_ASM_CONCL_T** : *int* −> (*THM* −> *TACTIC*) −> *TACTIC*;

Description Process the negation of current goal with the supplied theorem tactic and make an assumption, suitably negated, into the current goal. If the simplifications it does are ignored, *SWAP_ASM_CONCL_T asm ttac*is equivalent to

Example

CONTR_T (fn x => asm_ante_tac asm THEN ttac x)

and *SWAP_NTH_ASM_CONCL_T n ttac*is equivalent to

Example

CONTR_T (fn x => (DROP_NTH_ASM_T n ante_tac) THEN ttac x)

Tactic

$$\frac{\{\ \Gamma, \ulcorner t1 \urcorner\ \}\ t2}{ttac(asm\_rule\ \ulcorner\neg t2\urcorner\ )(\{\Gamma\}\ \neg\ t1)}$$

SWAP_ASM_CONCL_T
⌜*t1*⌝
ttac

Tactic

$$\frac{\{\ \ulcorner t1\urcorner, ..., \ulcorner tm\urcorner, ..., \ulcorner tn\urcorner\}\ t}{ttac(asm\_rule\ \ulcorner\neg t\urcorner\ )\ (\{\ulcorner t1\urcorner, ..., \ulcorner tn\urcorner\}\ \neg\ tm)}$$

SWAP_NTH_ASM_CONCL_T
m
ttac

If the assumption is a negated term then the double negation will be eliminated.

See Also Other *swap* and *SWAP* functions.

Errors

9303	*the index ?0 is out of range*
28027	*Conclusion of goal does not have type* ⌜:*BOOL*⌝
28052	*Term ?0 is not in the assumptions*

SML

val **swap_∨_tac** : *TACTIC*;

Description Interchange the disjuncts of a disjunctive goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ a \lor b}{\{\ \Gamma\ \}\ b \lor a}$$

swap_∨_tac

Uses For use in conjunction with *strip_tac* (q.v.) when the reduction of $\{\Gamma\}a \lor b$ to $\{\neg a, \Gamma\}b$ is inappropriate.

See Also *∨_left_tac*, *∨_right_tac*, *swap_∨_tac*, *strip_tac*

Errors

28041	*Goal is not of the form*: { *Γ* } *a* ∨ *b*

SML

val **tac_proof** : (*GOAL* ∗ *TACTIC*) −> *THM*;

Description *tac_proof*(*gl*, *tac*) applies the tactic *tac* to the goal *gl*, and, if the tactic succeeds in proving the goal returns the resulting theorem.

tac_proof performs α-conversion, introduces additional assumptions, and reorders assumptions as necessary to ensure that the theorem returned has the same form as the specified goal (note that this is not possible if the goal has α-equivalent assumptions). In circumstances where these adjustments are known not to be necessary, *simple_tac_proof* may be used to avoid the overhead.

Uses The subgoal package is the normal interactive mechanism for developing proofs using tactics. *tac_proof* is typically used in tactic programming and other proof procedures, in cases where it is necessary to ensure that the correct goal is proved.

Errors

9501	*the tactic returned unsolved subgoals*: ?*0*
9502	*evaluation of the tactic failed*: ?*0*
9503	*the proof returned by the tactic failed*: ?*0*
9504	*the proof returned by the tactic proved* ?*0 which could not be converted into the desired goal.*
9505	*the goal contains alpha−equivalent assumptions* (?*0 and* ?*1*)
9506	*the assumption* ?*0 is not of type* ⌜:*BOOL*⌝
9507	*the conclusion* ?*0 is not of type* ⌜:*BOOL*⌝

See Also *simple_tac_proof*, *prove_thm*.

SML

val **taut_conv** : *CONV*;

Description A tautology prover. A tautology is taken to be any universally quantified substitution instance of a term which is formed from boolean variables, the constants T and F and the following connectives:

∧, ∨, ⇒, ⇔, ¬, *if ... then ... else*

and which is true for any assignment of truth values to the variables. If its argument is a tautologically true term, then the function will return a theorem that the term is equivalent to T.

Conversion

$$\frac{}{\vdash t \Leftrightarrow T} \quad \begin{array}{l} taut\_conv \\ \ulcorner t \urcorner \end{array}$$

See Also *taut_tac*, *taut_rule*, *simple_taut_tac*.

Errors

27037 ?*0 is not tautologically true*

SML
val **taut_rule** : *TERM* −> *THM*;

Description A tautology prover. A tautology is taken to be any universally quantified substitution instance of a term which is formed from boolean variables, the constants T and F and the following connectives:

∧, ∨, ⇒, ⇔, ¬, *if ... then ... else*

and which is true for any assignment of truth values to the variables. If its argument is such a tautology then the function will return that term as a theorem.

Rule

$$\frac{}{\vdash t} \quad \begin{array}{l} taut\_rule \\ \ulcorner t \urcorner \end{array}$$

See Also *taut_tac*, *taut_conv*, *simple_taut_tac*.

Errors
27037 ?0 is not tautologically true

SML
val **taut_tac** : *TACTIC*;

Description A tautology prover. If the conclusion of the goal is a tautology then *taut_tac* will prove the goal. A tautology is taken to be any (perhaps universally quantified) substitution instance of a term which is formed from boolean variables, the constants T and F and the following connectives:

∧, ∨, ⇒, ⇔, ¬, *if ... then ... else*

and which is true for any assignment of truth values to the variables.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{} \quad taut\_tac$$

See Also *strip_tac*, *taut_rule*, *taut_conv*, *simple_taut_tac*.

Errors
29020 Conclusion of the goal is not a universally quantified tautology

SML
val **THEN_LIST_T** : (*TACTIC* ∗ *TACTIC list*) −> *TACTIC*;
val **THEN_LIST** : (*TACTIC* ∗ *TACTIC list*) −> *TACTIC*;

Description *THEN_LIST_T* is a tactical used as an infix operator. *tac THEN_LIST_T tlist* is a tactic that applies *tac*, and then applies the first member of *tlist* to the first resulting subgoal, the second to the second, etc. If there are not the correct number of tactics in the list then an error will be raised. *THEN_LIST* is an alias for *THEN_LIST_T*.

Errors
9101 number of tactics must equal the number of subgoals

As failures of the initial tactic or the tactics in the list.

SML
```
val THEN_T1 : (TACTIC * TACTIC) -> TACTIC;
val THEN1 : (TACTIC * TACTIC) -> TACTIC;
```

Description *THEN_T1* is a tactical used as an infix operator. *tac1 THEN_T1 tac2* is the tactic that applies *tac1* and then applies *tac2* to the first of the resulting subgoals and *id_tac* to any other subgoals. If *tac1* returns no subgoals, then nor will *tac1 THEN_T1 tac2*. *THEN1* is an alias for *THEN_T1*.

It is intended for use in conjunction with induction tactics or tactics like *lemma_tac* for which the first subgoal (i.e., the base case of the induction or the lemma) often has a simple proof.

See Also *THEN*

Errors As the errors of *tac1* and *tac2*.

SML
```
val THEN_TRY_LIST_T : (TACTIC * TACTIC list) -> TACTIC;
val THEN_TRY_LIST : (TACTIC * TACTIC list) -> TACTIC;
```

Description *THEN_TRY_LIST_T* is a tactical used as an infix operator. *tac THEN_TRY-_LIST_T tlist* is a tactic that applies *tac*, and then attempts to apply the first member of *tlist* to the first resulting subgoal, the second to the second, etc. If there are not the correct number of tactics in the list then an error will be raised. If any member of *tlist* fails on a particular subgoal, then that subgoal is returned unchanged. *THEN_LIST* is an alias for *THEN_LIST_T*.

Errors
```
9101 number of tactics must equal the number of subgoals
```

As failures of the initial tactic.

SML
```
val THEN_TRY_TTCL : (THM_TACTICAL * THM_TACTICAL) ->
        THM_TACTICAL;
```

Description *THEN_TRY_TTCL* is a theorem tactical combinator. It is an infix operator which applies the first theorem tactical, and then, if it succeeds, the second theorem tactical, using only the first result if the second fails.

Uses For use in programming theorem tacticals.

SML
```
val THEN_TRY_T : (TACTIC * TACTIC) -> TACTIC;
val THEN_TRY : (TACTIC * TACTIC) -> TACTIC;
```

Description *THEN_TRY_T* is a tactical used as an infix operator. *tac1 THEN_TRY_T tac2* is the tactic that applies *tac1* and then attempts to apply *tac2* to each resulting subgoal (perhaps none). If *tac2* fails on any particular subgoal then that subgoal will be unchanged from the result of *tac1*. If *tac1* fails then the overall tactic fails. *THEN_TRY* is an alias for *THEN_TRY_T*.

Errors As the errors of *tac1*.

SML
```
val THEN_TTCL : (THM_TACTICAL * THM_TACTICAL) -> THM_TACTICAL;
```

Description *THEN_TTCL* is a theorem tactical combinator. It is an infix operator which composes two theorem tacticals using ordinary function composition:

```
(tcl1 THEN_TTCL tcl2) thmtac thm = (tcl1 o tcl2) thmtac thm
```

Uses For use in programming theorem tacticals.

SML
```
val THEN_T : (TACTIC * TACTIC) -> TACTIC;
val THEN : (TACTIC * TACTIC) -> TACTIC;
```

Description *THEN_T* is a tactical used as an infix operator. *tac1 THEN_T tac2* is the tactic that applies *tac1* and then applies *tac2* to each resulting subgoal (perhaps none). *THEN* is an alias for *THEN_T*.

Errors As the errors of *tac1* and *tac2*.

SML
```
val TOP_ASM_T : (THM -> TACTIC) -> TACTIC;
```

Description If the top entry in the assumption list is *asm* say, *TOP_ASM_T thmtac* acts as *thmtac*(*asm*⊢*asm*).

Tactic

$$\frac{\{\ asm,\ \Gamma\ \}\ t}{thmtac\ (asm \vdash asm)\ (\{\ asm,\ \Gamma\ \}\ t)} \qquad \begin{array}{l} TOP\_ASM\_T \\ \ulcorner asm \urcorner \\ thmtac \end{array}$$

Uses To use an assumption as a theorem

Errors

9302 the assumption list is empty

SML
```
val TRY_TTCL : THM_TACTICAL -> THM_TACTICAL;
```

Description *TRY_TTCL ttcl* is a theorem tactical which applies *ttcl* if it can, and otherwise acts as *ID_THEN*.

Uses For use in programming theorem tacticals.

SML
```
val TRY_T : TACTIC -> TACTIC;
val TRY : TACTIC -> TACTIC;
```

Description *TRY_T tac* is a tactic which applies *tac* to the goal and if that fails leaves the goal unchanged. It is the same as *tac ORELSE id_tac*. *TRY* is an alias for *TRY_T*.

SML
```
val t_tac : TACTIC;
```

Description Prove a goal with conclusion '*T*'.

Tactic

$$\frac{\{\ \Gamma\ \}\ T}{} \qquad t\_tac$$

See Also *strip_tac*, *taut_tac*.

Uses Tactic programming.

Errors

28011 Goal does not have the form {Γ}T

SML

```
val var_elim_asm_tac : TERM -> TACTIC;
val var_elim_nth_asm_tac : int -> TACTIC;
val VAR_ELIM_ASM_T : TERM -> (THM -> TACTIC) -> TACTIC;
val VAR_ELIM_NTH_ASM_T : int -> (THM -> TACTIC) -> TACTIC;
```

Description These tactics and tacticals do variable elimination with a chosen assumption of the goal. They take an assumption of the form: ⌜*var* = *value*⌝ or ⌜*value* = *var*⌝, where *var* is a variable and, if the subterm *value* does not contain *var* free, they substitute *value* for the free variable *var* throughout the goal (discarding the original assumption).

If an assumption is an equation of variables, then the tactic will strip digits and the current variant suffix from the right of the two variable names, and will choose to eliminate the variable with the shortest remaining name string, taking eliminating the left hand side variable if the strings are of equal length (this is a heuristic). If the variables are the same then the assumption is just discarded with no further effect.

var_elim_asm_tac will determine whether its term argument is an assumption of the above form. If so, it will substitute for the free variable *var* with *value* throughout the goal, stripping any changed assumptions back into the goal (returning the rest by *check_asm_tac*), and then discard the original assumption. *VAR_ELIM_ASM_T* allows the users choice of function to be applied to the modified assumptions.

var_elim_nth_asm_tac works as *var_elim_asm_tac*, except it takes an integer indicating the "nth" assumption is to be used. *VAR_ELIM_NTH_ASM_T* allows the users choice of function to be applied to the modified assumptions.

See Also *all_var_elim_asm_tac1* and its kin to apply this sort of functionality to all the assumptions simultaneously. *prop_eq_prove_tac* for more sophisticated approach to these kinds of problems.

Errors

9301	*the term ?0 is not in the assumption list*
9303	*the index ?0 is out of range*
29027	*?0 is not of the form ⌜var = ...⌝ or ⌜... = var⌝ where the variable ⌜var⌝ is not free in ⌜...⌝*

SML

```
val ⇔_T2 : (THM -> TACTIC) -> (THM -> TACTIC) -> TACTIC;
```

Description Reduce a bi-implication by passing the operands to tactic generating functions.

Tactic

$$\frac{\{\ \Gamma\ \}\ t1 \Leftrightarrow t2}{ttac1\{\ t1,\ \Gamma\ \} \vdash t2;\ ttac2\{\ t2,\ \Gamma\ \} \vdash t1} \qquad \begin{array}{l}\Leftrightarrow\_T2 \\ ttac1\ \ ttac2\end{array}$$

See Also *⇔_T*, *STRIP_CONCL_T*

Errors

28061	*Goal is not of the form: { Γ } t1 ⇔ t2*

SML
```
val ⇔_tac : TACTIC;
```

Description Reduce a bi-implication to two subgoals.

Tactic

$$\frac{\{\ \Gamma\ \}\ t1 \Leftrightarrow t2}{\{strip\ t1,\ \Gamma\ \} \vdash t2;\ \{strip\ t2,\ \Gamma\ \} \vdash t1} \quad \Leftrightarrow\_tac$$

See Also *strip_tac*, *⇔_T*

Errors
28061 Goal is not of the form: { Γ } t1 ⇔ t2

SML
```
val ⇔_THEN2 : (THM -> TACTIC) -> (THM -> TACTIC) ->
        (THM -> TACTIC);
```

Description A theorem tactical to apply given theorem tactics to the the result of eliminating ⇔ from a theorem of the form *Γ⊢t1⇔t2*.

$$\Leftrightarrow\_THEN2\ ttac1\ ttac2(\Gamma \vdash t1 \Leftrightarrow t2) = ttac1(\Gamma \vdash t1 \Rightarrow t2)\ THEN\ ttac2(\Gamma \vdash t2 \Rightarrow t1)$$

The function is partially evaluated with only the theorem tactic and theorem arguments.

See Also *⇔_THEN*, *STRIP_THM_THEN*

Errors
28062 ?0 is not of the form: ‘Γ ⊢ t1 ⇔ t2‘

SML
```
val ⇔_THEN : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical to apply a given theorem tactic to the result of eliminating ⇔ from a theorem of the form *Γ ⊢ t1 ⇔ t2*.

$$\Leftrightarrow\_THEN\ thmtac\ (\Gamma \vdash t1 \Leftrightarrow t2) = thmtac\ (\Gamma \vdash t1 \Rightarrow t2)\ THEN\ thmtac\ (\Gamma \vdash t2 \Rightarrow t1)$$

The function is partially evaluated with only the theorem tactic and theorem arguments.

See Also *⇔_THEN2*, *STRIP_THM_THEN*

Errors
28062 ?0 is not of the form: ‘Γ ⊢ t1 ⇔ t2‘

SML
```
val ⇔_t_tac : TACTIC;
```

Description Simplifies a goal of the form: ...⇔*T* or *T*⇔....

Tactic

$$\frac{\{\ \Gamma\ \}\ t \Leftrightarrow T}{\{\ \Gamma\ \}\ t} \quad \Leftrightarrow\_t\_tac$$

and

Tactic

$$\frac{\{\ \Gamma\ \}\ T \Leftrightarrow t}{\{\ \Gamma\ \}\ t} \quad \Leftrightarrow\_t\_tac$$

Errors
28012 Goal not of form: { Γ } t ⇔ T or { Γ } T ⇔ t

See Also *strip_tac*

Uses Tactic programming.

SML
```
val ⇔_T : (THM -> TACTIC) -> TACTIC;
```

Description Reduce a bi-implication by passing each operand to a tactic generating function.

Tactic

$$\frac{\{\ \Gamma\ \}\ t1 \Leftrightarrow t2}{ttac\{\ t1,\ \Gamma\ \} \vdash t2;\ ttac\{\ t2,\ \Gamma\ \} \vdash t1} \qquad \begin{array}{l} \Leftrightarrow\_T \\ ttac \end{array}$$

See Also *⇔_T2*, *STRIP_CONCL_T*

Errors
```
28061 Goal is not of the form: { Γ } t1 ⇔ t2
```

SML
```
val ∧_tac : TACTIC;
```

Description Reduce the proof of a conjunction to the proof of its conjuncts.

Tactic

$$\frac{\{\ \Gamma\ \}\ t1 \land t2}{\{\ \Gamma\ \}\ t1;\ \{\ \Gamma\ \}\ t2} \qquad \land\_tac$$

See Also *strip_tac*

Errors
```
28031 Goal is not of the form: { Γ } t1 ∧ t2
```

SML
```
val ∧_THEN2 : (THM -> TACTIC) -> (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical to apply given theorem tactics to the conjuncts of a theorem of the form $\Gamma \vdash t1 \land t2$.

```
∧_THEN2 thmtac1 thmtac2 (Γ ⊢ t1 ∧ t2) = thmtac1 (Γ ⊢ t1) THEN thmtac2 (Γ ⊢ t2)
```

See Also *∧_THEN*, *STRIP_THM_THEN*

Errors
```
28032 ?0 is not of the form: ‘Γ ⊢ t1 ∧ t2‘
```

SML
```
val ∧_THEN : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical to apply a given theorem tactic to the conjuncts of a theorem of the form $\Gamma \vdash t1 \land t2$.

```
∧_THEN thmtac (Γ ⊢ t1 ∧ t2) = thmtac (Γ ⊢ t1) THEN thmtac (Γ ⊢ t2)
```

The function may be partially evaluated with only its theorem tactic and theorem arguments.

See Also *∧_THEN2*, *STRIP_THM_THEN*

Errors
```
28032 ?0 is not of the form: ‘Γ ⊢ t1 ∧ t2‘
```

SML

val $\vee$_**left_tac** : *TACTIC*;

Description Take the left disjunct of the current goal as the subgoal.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{\ \Gamma\ \}\ a \vee b}{\{\ \Gamma\ \}\ a} \qquad \vee\_left\_tac$$

See Also $\vee$_*left_tac*, *swap_*$\vee$*_tac*, *strip_tac*

Errors

28041 Goal is not of the form: { Γ } $a \vee b$

SML

val $\vee$_**right_tac** : *TACTIC*;

Description Take the right disjunct of the current subgoal as the new subgoal.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{\ \Gamma\ \}\ a \vee b}{\{\ \Gamma\ \}\ b} \qquad \vee\_right\_tac$$

See Also $\vee$_*right_tac*, *swap_*$\vee$*_tac*, *strip_tac*

Errors

28041 Goal is not of the form: { Γ } $a \vee b$

SML

val $\vee$_**THEN2** : (*THM* –> *TACTIC*) –> (*THM* –> *TACTIC*) –> (*THM* –> *TACTIC*);

Description A theorem tactical to perform a case split on a given disjunctive theorem applying tactic generating functions to the extra assumption in each branch.

$\vee$_*THEN2 ttac1 ttac2* ($\Delta \vdash t1 \vee t2$) ({Γ} *t*) =
    ttac1 ($t1 \vdash t1$) ({Γ} *t*); *ttac2* ($t2 \vdash t2$)({Γ} *t*)

The function may be partially evaluated with only its theorem tactic and theorem arguments.

See Also *STRIP_THM_THEN*, $\vee$_*THEN*

Errors

28042 ?0 is not of the form: ‘$\Gamma \vdash t1 \vee t2$‘

SML

val $\vee$_**THEN** : (*THM* –> *TACTIC*) –> (*THM* –> *TACTIC*);

Description A theorem tactical to perform a case split on a given disjunctive theorem applying a tactic generating function to the extra assumption in each branch.

$\vee$_*THEN ttac* ($\Delta \vdash t1 \vee t2$) ({Γ} *t*) = *ttac* ($t1 \vdash t1$) ({Γ} *t*); *ttac* ($t2 \vdash t2$)({Γ} *t*)

The function may be partially evaluated with only its theorem tactic and theorem arguments.

See Also *STRIP_THM_THEN*, $\vee$_*THEN2*

Errors

28042 ?0 is not of the form: ‘$\Gamma \vdash t1 \vee t2$‘

SML

```
val ¬_elim_tac : TERM -> TACTIC;
```

Description Proof by showing assumptions give rise to two contradictory subgoals.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{\ \Gamma\ \}\ t2}{\{\ \Gamma\ \}\ t1;\ \{\ \Gamma\ \}\ \neg\ t1} \qquad \neg\_elim\_tac \quad \ulcorner t1 \urcorner$$

The function may be partially evaluated with only its term argument.

Uses In tactic programming. If an assumption has its negation also in the assumption list this will make for a rapid proof. *asm_ante_tac t1 THEN strip_tac* is a more memorable idiom for handling such a case in interactive use but is a little slower.

See Also *strip_tac*

Errors

28022 ?0 is not boolean

SML

```
val ¬_in_conv : CONV;
```

Description This is a conversion which moves a top level negation inside other predicate calculus connectives using whichever one of the following rules applies:

$$\begin{array}{lcl}
\neg\neg t & = & t \\
\neg(t1 \wedge t2) & = & \neg t1 \vee \neg t2 \\
\neg(t1 \vee t2) & = & \neg t1 \wedge \neg t2 \\
\neg(t1 \Rightarrow t2) & = & t1 \wedge \neg t2 \\
\neg(t1 \Leftrightarrow t2) & = & (t1 \wedge \neg t2) \vee (t2 \wedge \neg t1) \\
\neg(if\ a\ then\ t1\ else\ t2) & & \\
 & = & (if\ a\ then\ \neg t1\ else\ \neg t2) \\
\neg\forall vs \bullet t & = & \exists vs \bullet \neg t \\
\neg\exists vs \bullet t & = & \forall vs \bullet \neg t \\
\neg\exists_1 vs \bullet t & = & \forall vs \bullet \neg(t \wedge \forall vs' \bullet t[vs'] \Rightarrow vs' = vs) \\
\neg T & = & F \\
\neg F & = & T
\end{array}$$

Uses Tactic and conversion programming.

See Also *simple_¬_in_conv*, *¬_in_tac*

Errors

28131 No applicable rules for the term ?0

SML
```
val ¬_in_tac : TACTIC;
```

Description This is a tactic which moves a top level negation in the conclusion of the goal inside other predicate calculus connectives using the following rules:

$\neg\neg t$	$\rightarrow$	t
$\neg(t1 \land t2)$	$\rightarrow$	$\neg t1 \lor \neg t2$
$\neg(t1 \lor t2)$	$\rightarrow$	$\neg t1 \land \neg t2$
$\neg(t1 \Rightarrow t2)$	$\rightarrow$	$t1 \land \neg t2$
$\neg(t1 \Leftrightarrow t2)$	$\rightarrow$	$(t1 \land \neg t2) \lor (t2 \land \neg t1)$
$\neg\forall vs\bullet t$	$\rightarrow$	$\exists vs\bullet\neg t$
$\neg\exists vs\bullet t$	$\rightarrow$	$\forall vs\bullet\neg t$
$\neg\exists_1 vs\bullet t$	$\rightarrow$	$\forall vs\bullet\neg(t \land \forall vs'\bullet t[vs'] \Rightarrow vs' = vs)$
$\neg T$	$\rightarrow$	F
$\neg F$	$\rightarrow$	T

Uses

See Also *simple_¬_in_tac*, *¬_in_conv*

Errors
```
28025 No applicable rule for this goal
```

SML
```
val ¬_IN_THEN : THM_TACTICAL;
```

Description This is a theorem tactical which applies a given theorem tactic to the result of transforming a theorem by moving a top level negation inside other predicate calculus connectives using the following rules:

$\neg\neg t$	$\rightarrow$	t
$\neg(t1 \land t2)$	$\rightarrow$	$\neg t1 \lor \neg t2$
$\neg(t1 \lor t2)$	$\rightarrow$	$\neg t1 \land \neg t2$
$\neg(t1 \Rightarrow t2)$	$\rightarrow$	$t1 \land \neg t2$
$\neg(t1 \Leftrightarrow t2)$	$\rightarrow$	$(t1 \land \neg t2) \lor (t2 \land \neg t1)$
$\neg\forall vs\bullet t$	$\rightarrow$	$\exists vs\bullet\neg t$
$\neg\exists vs\bullet t$	$\rightarrow$	$\forall vs\bullet\neg t$
$\neg\exists_1 vs\bullet t$	$\rightarrow$	$\forall vs\bullet\neg(t \land \forall vs'\bullet t[vs'] \Rightarrow vs' = vs)$
$\neg T$	$\rightarrow$	F
$\neg F$	$\rightarrow$	T

This function partially evaluates given only the theorem and theorem-tactical.

See Also *SIMPLE_¬_IN_THEN*

Errors
```
29010 No applicable rule for ?0
```

SML
```
val ¬_rewrite_canon : THM -> THM list
val ∀_rewrite_canon : THM -> THM list
```

Description These are some of the standard canonicalisation functions used for breaking theorems up into lists of equations for use in rewriting. They four perform the following transformations:

¬_rewrite_canon	$(\Gamma \vdash \neg(t1 \lor t2))$	$= (\Gamma \vdash \neg t1 \land \neg t2)$
¬_rewrite_canon	$(\Gamma \vdash \neg\exists vs \bullet t)$	$= (\Gamma \vdash \forall vs \bullet \neg t)$
¬_rewrite_canon	$(\Gamma \vdash \neg\neg t)$	$= (\Gamma \vdash t)$
¬_rewrite_canon	$(\Gamma \vdash \neg t)$	$= (\Gamma \vdash t \Leftrightarrow F)$
∀_rewrite_canon	$(\Gamma \vdash \forall vs \bullet t)$	$= \Gamma \vdash t$

See Also *simple_¬_rewrite_canon*, *simple_∀_rewrite_canon*.

Errors
```
26201 Failed as requested
```

The area given by the failure will be *fail_canon*.

SML
```
val ¬_T2 : TERM -> (THM -> TACTIC) -> (THM -> TACTIC) -> TACTIC;
```

Description A form of proof by contradiction using two theorem tactics to simplify the subgoals.

Note that *strip_tac* may be used to push a negation inside other logical connectives, which is often the best way of handling a negated goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ \neg t2}{\begin{array}{c} ttac1\ (t2 \vdash t2)\ \{\ \Gamma\ \}\ t1; \\ ttac2\ (t2 \vdash t2)\ \{\ \Gamma\ \}\ \neg\ t1 \end{array}} \quad \begin{array}{l} \neg\_T2 \\ ttac1\ \ ttac2 \\ \ulcorner t1 \urcorner \end{array}$$

Uses To prove a negated term by showing that assuming the term gives rise to a contradiction.

See Also *strip_tac*, *contr_tac*, *¬_tac*, *STRIP_CONCL_T*, *¬_in_conv*

Errors
```
28022 ?0 is not boolean
28023 Goal is not of the form ⌜¬ t⌝
```

SML

val **¬_tac** : *TERM* −> *TACTIC*;

Description A form of proof by contradiction as a tactic: $\neg t2$ holds if $t2 \vdash t1$ and $t2 \vdash \neg t1$ for some term $t1$.

Note that *strip_tac* may be used to push a negation inside other logical connectives, which is often the best way of handling a negated goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ \neg t2}{\{strip\ t2,\ \Gamma\ \}\ t1;\ \{strip\ t2,\ \Gamma\ \}\ \neg\ t1} \qquad \begin{array}{l}\neg\_tac\\ \ulcorner t1 \urcorner\end{array}$$

Uses To prove a negated term by showing that assuming the term gives rise to a contradiction.

See Also *strip_tac*, *contr_tac*, *¬_T*

Errors

28022 ?0 is not boolean
28023 Goal is not of the form ⌜¬ t⌝

SML

val **¬_T** : *TERM* −> (*THM* −> *TACTIC*) −> *TACTIC*;

Description A form of proof by contradiction using a theorem tactic to simplify the subgoals.

Note that *strip_tac* may be used to push a negation inside other logical connectives, which is often the best way of handling a negated goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ \neg t2}{\begin{array}{c}ttac\ (t2 \vdash t2)\ \{\ \Gamma\ \}\ t1;\\ ttac\ (t2 \vdash t2)\ \{\ \Gamma\ \}\ \neg\ t1\end{array}} \qquad \begin{array}{l}\neg\_T\\ ttac\\ \ulcorner t1 \urcorner\end{array}$$

Uses To prove a negated term by showing that assuming the term gives rise to a contradiction.

See Also *strip_tac*, *contr_tac*, *¬_tac*, *STRIP_CONCL_T*, *¬_in_conv*

Errors

28022 ?0 is not boolean
28023 Goal is not of the form ⌜¬ t⌝

SML
```
val ¬_¬_thm : THM
val ¬_∨_thm : THM
val ¬_∧_thm : THM
val ¬_⇒_thm : THM
val ¬_⇔_thm : THM
val ¬_if_thm : THM
val ¬_f_thm : THM

val ⇒_thm : THM
val ⇔_thm : THM
val if_thm : THM
```

Description These theorems are tautologies saved in the theory "misc" because they are frequently used in tactic and conversion programming.

The first seven theorems are De Morgan's laws for the various propositional connectives formulated so that they can be used to normalise a propositional term by moving all negations inside other connectives. *¬_t_thm* is also provided but is documented elsewhere.

The last three theorems give definitions for implication, bi-implication and conditional in terms of disjunction, conjunction and negation.

```
¬_¬_thm       ⊢ ∀a•¬ ¬a ⇔ a
¬_∨_thm       ⊢ ∀a b•¬ (a ∨ b) ⇔ (¬a ∧ ¬b)
¬_∧_thm       ⊢ ( ¬(a ∧ b) ⇔ (¬a ∨ ¬b)
¬_⇒_thm       ⊢ ∀a b•¬(a ⇒ b) ⇔ (a ∧ ¬b)
¬_⇔_thm       ⊢ ∀ a b • ¬ (a ⇔ b) ⇔ a ∧ ¬ b ∨ b ∧ ¬ a
¬_if_thm      ⊢ ∀ a b • ¬ (if a then T else T) ⇔ (if a then ¬ T else ¬ T)

¬_f_thm       ⊢ ¬F ⇔ T
⇒_thm         ⊢ ∀a b•(a ⇒ b) ⇔ (¬a ∨ b)
⇔_thm         ⊢ ∀a b•(a ⇔ b) ⇔ (a ⇒ b) ∧ (b ⇒ a)
if_thm        ⊢ ∀a b c•(if a then b else c) ⇔ (a ∧ b) ∨ (¬a ∧ c)
```

See Also *¬_t_thm*.

SML
```
val ⇒_tac : TACTIC;
```

Description Strip the antecedent of an implicative goal into the assumption list.

Tactic

$$\frac{\{\ \Gamma\ \}\ t1\ \Rightarrow\ t2}{\{strip\ t1,\ \Gamma\ \}\ t2}\qquad \Rightarrow\_tac$$

Errors
```
28051 Goal is not of form: { Γ } t1 ⇒ t2
```

SML
```
val ⇒_THEN : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical to apply a given theorem tactic to the result of eliminating ⇒ from a theorem of the form $\Gamma \vdash t1 \Rightarrow t2$.

$$\Rightarrow\_THEN\ thmtac\ (\Gamma \vdash t1 \Rightarrow t2) = thmtac\ (\Gamma \vdash \neg\ t1 \lor t2)$$

The function is partially evaluated with only the theorem tactic and theorem arguments.

Errors
```
28054 ?0 is not of the form: ‘Γ ⊢ t1 ⇒ t2‘
```

SML
```
val ⇒_thm_tac : THM -> TACTIC;
```

Description A tactic which uses a theorem whose conclusion is an implication, $t1 \Rightarrow t2$, to reduce a goal with conclusion $t2$ to $t1$.

Tactic

$$\frac{\{\ \Gamma\ \}\ t2}{\{\ \Gamma\ \}\ t1} \qquad \begin{array}{l}\Rightarrow\_thm\_tac \\ \Gamma 1 \vdash t1 \Rightarrow t2\end{array}$$

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Uses Mainly for use in tactic programming where the extra generality of *bc_thm_tac* and *bc_tac* is not required.

See Also *bc_thm_tac*, *bc_tac*.

Errors
```
29013 Conclusion of the goal is not ?0
```

SML
```
val ⇒_T : (THM -> TACTIC) -> TACTIC;
```

Description Reduce an implicative goal by passing the antecedent to a tactic generating function.

Tactic

$$\frac{\{\ \Gamma\ \}\ t1 \Rightarrow t2}{ttac\{\ t1,\ \Gamma\ \}\ t2} \qquad \begin{array}{l}\Rightarrow\_T \\ ttac\end{array}$$

Errors
```
28051 Goal is not of form: { Γ } t1 ⇒ t2
```

SML
```
val ∀_tac : TACTIC;
```

Description Reduce a universally quantified goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ \forall\ vs[x1,...] \bullet t[x1,...]}{\{\ \Gamma\ \}\ t[x1',...]} \qquad \forall\_tac$$

where $x1'$ is a variant name of $x1$, etc, different from any variable in Γ or t.

See Also *simple_∀_tac*

Errors
```
29001 Goal is not of the form: { Γ } ∀ vs • t[vs]
```

SML
```
val ∃_tac : TERM -> TACTIC ;
```

Description Provide a witness for an existential subgoal.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{ \Gamma \} \; \exists \; vs[x1,...] \bullet t2[x1,...]}{\{ \Gamma \} \; t2[t1',...]} \qquad \exists\_tac \; \ulcorner t \urcorner$$

where $vs[t\_1, ...]$ is t, type instantiated to have the same type as $vs[x1, ...]$, and broken up using *Fst* and *Snd* as necessary.

See Also *simple_∃_tac*

Errors

29002 Goal is not of the form: $\{ \Gamma \} \; \exists \; vs \bullet t2[vs]$
29008 Cannot match witness ?0 to varstruct ?1

SML
```
val ∃_THEN : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical which applies a given theorem tactic to the result of eliminating the outermost quantifier from a theorem of the form $\Gamma \vdash \exists vs \bullet t$.

$$\exists\_THEN \; thmtac \; (\Gamma \vdash \exists vs[x1,...] \bullet t) = thmtac \; (\Gamma \vdash t[x1'/x1,...])$$

where ⌜x1'⌝ is a variant of ⌜x1⌝ , etc, which does not appear in Γ or in the assumption or conclusion of the goal.

See Also *SIMPLE_∃_THEN*

Errors

29003 ?0 is not of the form: '$\Gamma \vdash \exists \; vs \bullet t$'

SML
```
val ∃₁_tac : TERM -> TACTIC;
```

Description Provide a witness for a goal with conclusion of the form $\exists_1 x \bullet t$.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Tactic

$$\frac{\{ \Gamma \} \; \exists_1 \; vs[x1,...] \bullet P[x1,...]}{\begin{array}{c}\{ \Gamma \} \; P[t1',...]; \\ \{ \Gamma \} \; \forall vs[x1',...] \bullet \\ P[x1',...] \Rightarrow vs[x1',...] = t'\end{array}} \qquad \exists_1\_tac1 \; \ulcorner t \urcorner$$

where x_i' is a variant of x_i which does not occur free in t, t' is equal to t type instantiated to the type of $vs[x1, ...]$, and $vs[t1', ...]$ equals t' (perhaps using *Fst* and *Snd*).

Errors

29004 Goal is not of the form: $\{ \Gamma \} \; \exists_1 \; vs \bullet t$
29008 Cannot match witness ?0 to varstruct ?1

SML
```
val ∃₁_THEN : (THM -> TACTIC) -> (THM -> TACTIC);
```

Description A theorem tactical which applies a given theorem tactic to the result of eliminating the outermost quantifier from a theorem of the form $\Gamma \vdash \exists\_1 vs \bullet t$.

```
∃₁_THEN thmtac (Γ ⊢ ∃₁ vs[x1,...]•t) =
        thmtac (Γ ⊢ t[x1′/x1,...] ∧
                ∀vs[x1″,...]•P[x1″,...] ⇒ vs[x1″,...] = vs[x1′,...])
```

where ⌜x1'⌝ and ⌜x1"⌝ are distinct variants of ⌜x1⌝, etc, which do not appear free in Γ or in the assumptions or conclusion of the goal.

Errors
```
29005 ?0 is not of the form: ‘Γ ⊢ ∃₁ vs • t‘
```

SML
```
val ϵ_tac : TERM -> TACTIC;
val ϵ_T : TERM -> (THM -> TACTIC) -> TACTIC;
```

Description Given a choice term, $\epsilon x \bullet t$ say, *ϵ_tac* sets $\exists x \bullet t$ as a lemma, and derives the new assumption $t[\epsilon x \bullet t / x]$ from it.

ϵ_T is the same as *ϵ_tac* except that it passes the new assumption to a tactic generating function.

Tactic

$$\frac{\{\ \Gamma\ \}\ t1}{\{\ \Gamma\ \}\ \exists x \bullet t;\ \{strip\ t[\epsilon x \bullet t/x],\ \Gamma\ \}\ t1} \qquad \begin{array}{l} \epsilon\_tac \\ \ulcorner \epsilon x \bullet t \urcorner \end{array}$$

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable. This occurs when the use of the choice function is in some sense irrelevant to the truth of the goal, e.g., $(\epsilon x \bullet T) = (\epsilon x \bullet T)$.

See Also *all_ϵ_tac*, *all_ϵ_T* (which are easier to use in most cases).

Errors
```
29050 ?0 is not of the form ⌜ϵx•p x⌝
```

7.4 Propositional Equational Reasoning

SML

signature **PropositionalEquality** = *sig*

Description This is the signature of a structure containing proof procedures for propositional calculus with equality.

SML

(* *Proof Context*: **prop_eq** *)

Description This is a complete proof context whose purpose is to supply a decision procedure for problems involving sets of equalities and the propositional calculus.

Contents The rewriting, theorem stripping and conclusion stripping components are as for the proof context *predicates* (q.v.). The automatic proof tactic is *prop_eq_prove_tac* (q.v.) The automatic proof conversion just tries to prove its argument, t say, using the automatic proof tactic and returns $t \Leftrightarrow T$ if it succeeds.

SML

(* *Proof Context*: **′prop_eq** *)

Description This is a component proof context whose purpose is to supply a decision procedure for problems involving sets of equalities and the propositional calculus.

Contents The automatic proof components are as for proof context *prop_eq*. Other components are blank.

SML

(* *Proof Context*: **prop_eq_pair** *)

Description This is a complete proof context whose main purpose is to supply a decision procedure for problems involving sets of equalities, the propositional calculus and pairing.

Contents The rewriting, theorem stripping and conclusion stripping components are as for the proof context *predicates* (q.v.) each augmented with conversion *pair_eq_conv* (q.v.) which effect the following transformations:

$$
\begin{array}{lll}
Fst(a,b) = x & \to & a = x \\
Snd(a,b) = y & \to & b = y \\
x = Fst(a,b) & \to & x = a \\
y = Snd(a,b) & \to & y = b \\
(a,b) = (c,d) & \to & a = c \land b = d \\
(a,b) = z & \to & a = Fst\ z \land b = Snd\ z \\
z = (a,b) & \to & Fst\ z = a \land Snd\ z = b \\
z = w & \to & Fst\ z = Fst\ w \land Snd\ z = Snd\ w
\end{array}
$$

The automatic proof tactic is *prop_eq_prove_tac* (q.v.). The automatic proof conversion just tries to prove its argument, t say, using the automatic proof tactic and returns $t \Leftrightarrow T$ if it succeeds.

SML

(* *Proof Context*: **′prop_eq_pair** *)

Description This is a component proof context whose purpose is to supply a decision procedure for problems involving sets of equalities, the propositional calculus and pairing.

Contents The rewriting, theorem stripping and conclusion stripping components contain only the *pair_eq_conv* conversion. The automatic proof components are as for *prop_eq_pair*. Other components are blank.

SML

```
val ASM_PROP_EQ_T : (THM list -> TACTIC) -> THM list -> TACTIC
val PROP_EQ_T : (THM list -> TACTIC) -> THM list -> TACTIC
```

Description These are theorem tacticals which process the argument theorems and (for *ASM_PROP_EQ_T*) the assumptions before calling the argument theorem tactic. A call of "*ASM_PROP_EQ_T thm_tac thms*" takes *thms* plus theorems representing any equations from the assumptions, these are cannonicalised by the rewriting canon of the current proof context, then processed by *prop_eq_rule* (q.v.) to form the arguments passed to function *thm_tac*. The order of the assumptions may be changed. Tactical *PROP_EQ_T* does not use the assumptions.

Uses With the rewriting tactics.

SML

```
val pair_eq_conv : CONV
```

Description This conversion transforms equations involving pairs and the constants *Fst* and *Snd* into new equations whose comparands have simpler types by using the first match found in the following rules:

$Fst(a,b) = x$	$\rightarrow$	$a = x$
$Snd(a,b) = y$	$\rightarrow$	$b = y$
$x = Fst(a,b)$	$\rightarrow$	$x = a$
$y = Snd(a,b)$	$\rightarrow$	$y = b$
$(a,b) = (c,d)$	$\rightarrow$	$a = c \wedge b = d$
$(a,b) = z$	$\rightarrow$	$a = Fst\ z \wedge b = Snd\ z$
$z = (a,b)$	$\rightarrow$	$Fst\ z = a \wedge Snd\ z = b$
$z = w$	$\rightarrow$	$Fst\ z = Fst\ w \wedge Snd\ z = Snd\ w$

Uses The conversion is intended for use in tactic and conversion programming. It is usefully applied before using *prop_eq_prove_tac* or *ASM_PROP_EQ_T* (q.v.). The normal interactive interface is via rewriting or stripping in the proof context *prop_eq_pair* (q.v.).

Errors

84001 ?0 is not an equation involving pairs

SML

```
val prop_eq_prove_tac : THM list -> TACTIC;
```

Description This tactic is suitable to be used as an automatic proof procedure in a proof context, it aims to solve problems which may be solved by reasoning in the propositional calculus with equality.

The tactic has the following steps:

1. It strips all of the assumptions, using the stripping functions of the current proof context, back into the assumptions. More precisely, '*DROP_ASMS_T (MAP_EVERY strip_asm_tac)*' is used.

2. It applies *contr_tac* to increase the stock of assumptions.

3. It splits all of the assumptions into two groups, those which are equations and those which are not.

4. Using the equation assumptions and the given theorems, a new set of theorems is produced using *prop_eq_rule* (q.v.) which equate all members of an equivalence classes to a common member of the class.

5. It rewrites all of the other assumptions with these new theorems and with the rewriting theorems of the current proof context.

6. It strips the rewritten assumptions and the equational assumptions from step 3 back into the goal.

SML

```
val prop_eq_rule : THM list -> THM list * THM list;
```

Description Given a list of theorems with conclusions of the form ⌜ $a_i = b_i$ ⌝ for various a_i and b_i this function produces a set of theorems that equate all members of each equivalence class determined by the equations to a common value. The equivalence classes are the sets of all a_i and b_i that are equated either directly or transitively, they comprise terms that are α-convertible rather than requiring strict equality. For each of the equivalence classes a set of theorems equating each term in the class to the "simplest" (see below) term in the class is generated. These new theorems have the simplest term as their right hand comparand, duplicated theorems and identity theorems are excluded. The first list in the result tuple contains the new theorems from all of the equivalence classes. The second list in the result tuple comprises all the argument theorems which were not equasions. The new theorems are intended to be used as arguments for a rewriting operation.

The choice of the "simplest" term is intended to give the most useful rewriting theorems and those which are least likely to loop. HOL constants are considered the most simple, variables next, then functional applications, with lamdba abstractions considered the most complex. A simple recursive counting function is used to traverse each term to evaluate its complexity. Function *term_order* (q.v.) is used when the counting function cannot decide.

Example

Applying this rule to a list of theorems with the following conclusions:

```
⊢ a1=b1    ⊢ a1=c1    ⊢ d1=c1    ⊢ z1=x1
⊢ b1=y1    ⊢ z1=w1    ⊢ w1=y1    ⊢ c1=y1
⊢ a2=b2    ⊢ a2=c2    ⊢ d2=c2    ⊢ z2=x2
⊢ b2=y2    ⊢ z2=w2    ⊢ w2=y2    ⊢ c2=y2
⊢ x ∧ y
```

will produce a list of theorems with the following conclusions as the first element of the result tuple:

```
⊢ x1=a1    ⊢ z1=a1    ⊢ w1=a1    ⊢ d1=a1
⊢ y1=a1    ⊢ b1=a1    ⊢ c1=a1
⊢ x2=a2    ⊢ z2=a2    ⊢ w2=a2    ⊢ d2=a2
⊢ y2=a2    ⊢ b2=a2    ⊢ c2=a2
```

plus the non equational theorems the second element of the result tuple.

7.5 Algebraic Normalisation

SML

signature **Normalisation** = *sig*

Description This is the signature of a structure containing conversions for monomial and polynomial term normalisation and related metalanguage functions.

SML

val **anf_conv** : *CONV*;
val **ANF_C** : *CONV −> CONV*;

Description *anf_conv* is a conversion which proves theorems of the form ⊢ *t1* = *t2* where *t1* is a term formed from atoms of type ℕ and *t2* is in what we may call additive normal form, i.e. it has the form: $t_1 + t_2 + ...$, where the t_i have the form $s_1 * s_2 * ...$where the s_i are atoms. Here, by atom we mean a term which is not of the form $t_1 + t_2 + ...$or $s_1 * s_2 * ....$

The summands t_i and, within them, the factors s_j are given in increasing order with respect to the ordering on terms given by the function *term_order*, q.v. Arithmetic computation is carried out on atoms to ensure that at most one of the summands is a numeric literal and that, within each summand, at most one factor is a numeric literal. Any literal appears at the beginning of its factor or summand and addition of *0* or multiplication by *1* is simplified out.

ANF_C conv is a conversion which acts like *anf_conv* but which applies *conv* to each atom as it is encountered (and normalises the result recursively). The argument conversion may signal that it does not wish to change a subterm, *t* say, either by failing or by returning *t* = *t*, the former approach is more efficient.

The conversions fail with error number 81032 if there are no changes to be made to the term.

Errors

81032 ?0 is not of type ⌜:ℕ⌝ or is already in additive normal form

SML

val **ASYM_C** : *CONV −> CONV*
val **GEN_ASYM_C** : *TERM ORDER −> CONV −> CONV*

Description These conversionals allow one to control the behaviour of a conversion by making it asymmetric with respect to an ordering relation on terms (in the sense that the resulting conversion will only prove theorems of the form *t1* = *t2* in which *t2* strictly precedes *t1* in the ordering.

ASYM_C c is a conversion which behaves like *c* on terms *t1* for which *c t1* is a theorem with conclusion *t1* = *t2* where *t2* (strictly) precedes *t1* in the standard ordering on terms given by *term_order* q.v. and fails on other terms.

GEN_ASYM_C is like *ASYM_C* but allows the ordering function used to be supplied as a parameter. The parameter is interpreted as an ordering relation on terms in the same sense as the ordering relations used by *sort*, q.v.

Errors

81010 The conversion did not decrease the order of the term
81011 On argument ?0 the conversion returned ?1 which is not an equation

SML

```
val cnf_conv : CONV;
```

Description This is a conversion which proves theorems of the form $\vdash t1 \Leftrightarrow t2$ where *t2* is in conjunctive normal form, i.e. either T or F or a conjunction of one or more disjunctions in which each disjunct is a propositional atom. Here, by atom we mean either a term whose principal connective is not a propositional calculus connective or the negation of such a term.

The conversion simplifies disjunctions and conjunctions as they are generated according to the following schema.

$$
\begin{array}{lll}
a \land T & \rightarrow & a \\
T \land a & \rightarrow & a \\
F \land a & \rightarrow & F \\
a \land F & \rightarrow & F \\
a \land a & \rightarrow & a \\
a \land \neg a & \rightarrow & F \\
 & & \\
a \lor T & \rightarrow & T \\
T \lor a & \rightarrow & T \\
F \lor a & \rightarrow & a \\
a \lor F & \rightarrow & a \\
a \lor a & \rightarrow & a \\
a \lor \neg a & \rightarrow & T \\
 & & \\
\neg T & \rightarrow & F \\
\neg F & \rightarrow & T
\end{array}
$$

Note, however, that more global simplifications are not done, e.g. there is no attempt to eliminate a conjunct all of whose constituent atoms are contained in another conjunct. Thus, the conversion will not automatically prove tautologies.

The conversion fails with error number 81030 if there are no changes to be made to the term.

See Also *strip_tac* and *taut_rule* which supply a more useful and efficient means for working with the propositional calculus in most cases.

Errors

81030 ?0 is not of type ⌜:BOOL⌝ or is already in conjunctive normal form

SML

```
val dnf_conv : CONV;
```

Description This is a conversion which proves theorems of the form $\vdash t1 \Leftrightarrow t2$ where $t2$ is in disjunctive normal form, i.e. either T or F or a disjunction of one or more conjunctions in which each conjunct is a propositional atom. Here, by atom we mean either a term whose principal connective is not a propositional calculus connective or the negation of such a term.

The conversion simplifies disjunctions and conjunctions as they are generated according to the following schema.

$$
\begin{array}{ll}
a \land T \rightarrow & a \\
T \land a \rightarrow & a \\
F \land a \rightarrow & F \\
a \land F \rightarrow & F \\
a \land a \rightarrow & a \\
a \land \neg a \rightarrow & F \\
 & \\
a \lor T \rightarrow & T \\
T \lor a \rightarrow & T \\
F \lor a \rightarrow & a \\
a \lor F \rightarrow & a \\
a \lor a \rightarrow & a \\
a \lor \neg a \rightarrow & T \\
 & \\
\neg T \rightarrow & F \\
\neg F \rightarrow & T
\end{array}
$$

Note, however, that more global simplifications are not done, e.g. there is no attempt to eliminate a disjunct all of whose constituent atoms are contained in another disjunct. Thus, the conversion will not automatically prove tautologies.

The conversion fails with error number 81031 if there are no changes to be made to the term.

See Also *strip_tac* and *taut_rule* which supply a more useful and efficient means for working with the propositional calculus in most cases.

Errors

81031 ?0 is not of type ⌜:BOOL⌝ or is already in disjunctive normal form

SML

```
val gen_term_order : (TERM -> (TERM * INTEGER)) -> TERM -> TERM -> int;
```

Description *gen_term_order* gives a means of creating orderings on terms. It is retained for backwards compatibility, *make_term_order* now being the recommended way of constructing term orderings.

In the call *gen_term_order special*, the idea is that whenever two terms, *tm1* and *tm2* say, are compared, *special* is applied to them to produce two pairs, (*tm1′*, *k1*) and (*tm2′*, *k2*) say. These pairs are then compared lexicographically (using the ordering recursively for the first components, in a similar way to *term_order*, q.v.). It is the caller's responsibility to provide an argument *special* which will ensure that this procedure terminates. A sufficient condition is only to use functions *special* with the property that for some disjoint sets of terms *X_1*, *X_2*, ..., we have that *special tm* = (*tm*, *0*) if *tm* ∉ *X_i* for any *i* and that *special tm* = (*x_i*, *f_i*(*tm*)) if *tm* ∈ *X_i*, where *x_i* is a fixed element of *X_i* and *f_i* is a fixed injection of *X_i* into the natural numbers.

See Also *make_term_order1* which is now the recommended way of constructing new term orderings.

SML

```
val make_term_order :
        (TERM ORDER -> TERM ORDER) list -> TERM ORDER;
```

Description *make_term_order* provides a systematic method for constructing term orderings. Its argument is a list of term order combinators: i.e., endofunctions on the type of term orderings.

The orderings *make_term_order* returns are derived from a base ordering on terms which works as follows:

1. Constants are ordered lexicographically by name (using *ascii_order*), then type (using *type_order*).
2. Variables are ordered lexicographically by name (using *ascii_order*), then type (using *type_order*).
3. Simple λ-abstractions are ordered lexicographically by recursion, bound variable first, then matrix.
4. Applications ordered lexicographically by recursion, function first, then operand.

If the above function ewre called *base*, then the ordering *make_term_order* [*f*, *g*, ..., *h*] acts as: *f*(*g*(...(*h*(*base*))...)) where each recursive call in *base* is a call on *make_term_order* [*f*, *g*, ..., *h*].

For example, the following defines an ordering on terms whcih makes the immediate successor of any term of type *BOOL* its immediate successor:

```
fun f t = (dest_¬ t, 1) handle Fail _ => (t, 0);

val ¬_order = make_term_order [fn r => induced_order(f, pair_order r int_order)];
```

SML

```
val poly_conv : TERM ORDER ->
        THM -> THM -> THM -> THM -> THM ->
        CONV -> CONV -> CONV -> CONV;
```

Description This conversion normalises terms constructed from atoms using two operators, both associative and commutative, the second of which, say op_* distributes over the other, say op_+. For clarity, we write the two operators with infix syntax although they need not actually be infix constants. Here, by "atom" we mean any term which is not of the form $t1\ op_+\ t2$ or $t1\ op_*\ t2$. The theorems computed by the conversion have the form $t = t_1\ op_+\ t_2\ op_+\ ...$, where the $t_i$are in non-decreasing order with respect to the ordering on terms given by the first parameter and have the form $s_1\ op_*\ s_2\ op_*\ ...$, where the s_i are atoms and are in non-decreasing order.

The associativity and commutativity of the operators and the distributivity are given as the five theorem parameters (which are also used to infer what the two operators are; n.b. the operators can be arbitrary terms, they need not be constants). The remaining parameters are conversions which are applied to each atom as it is encountered and to each subterm of the form $t_i\ op_+\ ...$ or $t_i\ op_*\ ...$ as it id created. In more detail the parameters are, in order, as follows:

1. A term ordering, such as *term_order*, q.v.
2. A theorem of the form $\vdash \forall x\ y \bullet x\ op_+\ y = y\ op_+\ x$.
3. A theorem of the form $\vdash \forall x\ y\ z \bullet (x\ op_+\ y)\ op_+\ z = x\ op_+\ y\ op_+\ z$.
4. A theorem of the form $\vdash \forall x\ y \bullet x\ op_*\ y = y\ op_*\ x$.
5. A theorem of the form $\vdash \forall x\ y\ z \bullet (x\ op_*\ y)\ op_*\ z = x\ op_*\ y\ op_*\ z$.
6. A theorem of the form $\vdash \forall x\ y\ z \bullet x\ op_*\ (y\ op_+\ z) = (x\ op_*\ y)\ op_+\ (x\ op_*\ z)$.
7. A conversion to be applied to any subterm of the form $t_i\ op_+\ ...$ whenever such a subterm is created. The result of the conversion will not be further normalised.
8. A conversion to be applied to any subterm of the form $t_i\ op_*\ ...$ whenever such a subterm is created. The result of the conversion will not be further normalised.
9. A conversion to be applied to any atom as it is encountered. If the conversion produces a non-atomic term, this is normalised recursively as it is produced.

The conversions supplied as parameters may signal that they do not wish to change a subterm, t say, either by failing or by returning $t = t$, the former approach is more efficient. The whole conversion fails with error number 81025 if there are no changes to be made to the term.

Errors

81023	*?0 does not have the form* $\vdash$ *t1 op1 (t2 op2 t3) = (t1 op1 t2) op2 (t1 op1 t3)*
81024	*?0 and ?1 do not have the forms* $\vdash$ *t1 op1 t2 = t2 op1 t1* *and* $\vdash$ *t1 op1 (t2 op2 t3) = (t1 op1 t2) op2 (t1 op1 t3)*
81025	*?0 is already sorted*

SML

```
val sort_conv : TERM ORDER ->
        THM -> THM -> CONV -> CONV -> CONV;
```

Description This conversion normalises a term constructed from atoms using an associative and commutative binary operator, *op* say. For clarity, we write two operator with infix syntax although it need not actually be an infix constant. Here, by "atom" we mean any term which is not of the form *t1 op t2*. The theorems computed by the conversion have the form $t = t_1\ op\ t_2\ op\ ...$, where the t_i are in non-decreasing order with respect to the ordering on terms given by the first parameter.

The associativity and commutativity of the operator are given as the two theorem parameters (which are also used to infer what *op* is; n.b. *op* can be an arbitrary term, it need not be a constant). The remaining parameters are conversions which are applied to each atom as it is encountered and to each subterm of the form $t = t_i\ op\ ...$ as it is created. In more detail the parameters are, in order, as follows:

1. A term ordering, such as *term_order*, q.v.
2. A theorem of the form $\vdash \forall x\ y \bullet t\ x\ y = t\ y\ x$.
3. A theorem of the form $\vdash \forall x\ y\ z \bullet (x\ op\ y)\ op\ z = x\ op\ y\ op\ z$.
4. A conversion to be applied to each subterm of the form: $t_i\ op\ ...$ whenever such a subterm is created. The result of the conversion will not be further normalised.
5. A conversion to be applied to each atom as it is encountered. If the conversion produces a non-atomic term, this is normalised recursively.

The conversions supplied as parameters may signal that they do not wish to change a subterm, *t* say, either by failing or by returning $t = t$, the former approach is more efficient. The whole conversion fails with error number 81025 if there are no changes to be made to the term.

Errors

```
81021 ?0 does not have the form ⊢ t1 op t2 = t2 op t1
81022 ?0 does not have the form ⊢ (t1 op t2) op t3 = t1 op (t2 op t3)
81025 ?0 is already sorted
81029 Internal error: unexpected error in term normalisation package
```

SML

```
val term_order : TERM -> TERM -> int;
```

Description *term_order* gives an ordering relation on HOL terms. The ordering relation follows the same conventions as those used by the sorting function *sort*, namely, *term_order t1 t2* is negative if *t1* precedes *t2*, *0* if *t1* and *t2* are equivalent and positive if *t2* precedes *t1*. The ordering used is, with some exceptions, that all constants precede all variables which precede all abstractions which precede all applications. Lexicographic ordering on the immediate constituents gives the ordering within each of these four classes (using alphabetic ordering of strings, *type_order* or *term_order* recursively to order the constituents as appropriate). The exceptions are *(i)* that any term of the form $\neg t$ comes immediately after *t*, *(ii)* that the numeric literals *0*, *1*, ... are taken in numeric rather than alphabetic order and come before all other terms, and *(iii)* that terms of the form $i * x$ where *i* is a numeric literal are ordered so that the terms x, $0{*}x$, $1{*}x$, $2{*}x$, ... are consecutive.

See Also *gen_term_order1* which is the recommended way of constructing new term orderings.

SML

```
val type_order : TYPE -> TYPE -> int;
```

Description *type_order* gives a useful ordering relation HOL types. The ordering relation follows the same conventions as those used by the sorting function *sort*, namely, *type_order t1 t2* is negative if *t1* precedes *t2*, *0* if *t1* and *t2* are equivalent and positive if *t2* precedes *t1*. The ordering used is essentially that type variables are ordered by the alphabetic ordering of their names and precede all compound types which are ordered by the lexicographic ordering on their immediate constituents (using the alphabetic ordering for the type constructor names and the type ordering recursively for its operands).

7.6 First Order Resolution

SML

signature **Resolution** = *sig*

Description This is the signature of a structure providing Resolution facilities to ICL HOL.

SML

(* *resolution_diagnostics* − *boolean flag declared by new_flag* *)

Description This is by default false, but if set true then the resolution mechanism will report the generation of new, unsubsumed theorems, and whether these subsume pre-existing theorems.

Uses Provide the designer of the resolution functions access to detailed diagnostics. Not intended for use by others. May be withdrawn.

SML

```
type BASIC_RES_TYPE
        (* TERM * bool * (TERM * (TERM -> THM -> THM))list
           * TYPE list * THM * TERM list * TYPE list * int
           * FRAG_PRIORITY
        *);
type RES_DB_TYPE (* = BASIC_RES_TYPE list * BASIC_RES_TYPE list *
        BASIC_RES_TYPE list * THM list *);
```

Description These are type abbreviation for the basic resolution tool based on *prim_res_rule*. The arguments to *BASIC_RES_TYPE* are:

1. The term is a subterm of the theorem argument(5), reached through outer universal quantifications and all propositional connectives.
2. The bool is false if and only if the subterm occurs "negatively" in the conclusion of the theorem.
3. This list states how to specialise the given term to some other value in a theorem already specialised by the preceding entries in the list, and appropriately type instantiated.
4. The type list is the instantiable type variables of the subterm.
5. The theorem is the source of the fragment.
6. The term list is the term variables that may not be used in unifying the fragment
7. The next type list is the type variables that may not be used in unifying the fragment
8. The integer indicates the "generation", i.e. the number of resolutions involved in creating the fragment (initial theorems are at 0).
9. This argument indicates the priority given to taking this fragment from the *toprocess* list to use next.

The arguments to *RES_DB_TYPE*:

1. Items yet to be checked against (*against*).
2. Items checked against, but to be rechecked against new items to check with (*done*).
3. Items to check with (*toprocess*).
4. Theorems used to derive current items (*dbdata*).

SML

```
val BASIC_RESOLUTION_T : int -> THM list -> (THM -> TACTIC) ->
        (THM -> TACTIC) -> TACTIC;
```

Description *BASIC_RESOLUTION_T limit thms thmtac1 thmtac2* (*seqasms*, *conc*) will first apply *thmtac1* to the negated goal, probably adding it into the assumption list in some manner. The assumptions derived from this will become the set of support, the pre-existing assumptions and the input *thms* will be the rest of the theorems. These theorems will be resolved against each other until only theorems with default generation past *limit* can be derived, or until ...⊢ *F* is derived, or until no further resolution can be done. An assumption's or input theorem's generation is *0*, and a theorem that is the result of resolution has a default generation of *1* plus the sum of the generations of the resolved theorems. Its actual generation will be the minimum of its default generation, and the generations of any previous theorems it subsumes. In any resolution attempt at least one of the two theorems will be assumed fragments from the stripped goal, or be derived from an earlier resolution in the evaluation. Duplicates and pure specialisations will be discarded.

Resolution will be attempted on subterms reached through outer universal quantification, and propositional connectives, by specialising the outer quantifications, and by type instantiation on the input theorems *thms* where necessary and possible.

The resulting list of theorems will have all the *thms* removed, all the theorems derived from stripping and negating the goal, and all the old assumptions removed. *MAP_EVERY thmtac* is then applied to the new theorems, and then to the goal. As a special case, ...⊢ *F* is checked for, before any further processing. If present it will be used to prove the goal.

Uses On its own, or in combination with some canonicalisation of the input theorems.

Errors

67003 The limit, ?0, must be a positive integer
67004 No resolution occurred

SML
```
val BASIC_RESOLUTION_T1 : int -> THM list -> (THM -> TACTIC) ->
        TACTIC;
```

Description *BASIC_RESOLUTION_T1 limit thms thmtac* (*seqasms*, *conc*) will take the theorems gained by *asm_rule*'ing the assumptions and *thms* as inputs. These theorems will be resolved against each other until only theorems with default generation past *limit* can be derived, or until ...⊢ *F* is derived, or until no further resolution can be done. An assumption's or input theorem's generation is *0*, and a theorem that is the result of resolution has a default generation of *1* plus the sum of the generations of the resolved theorems. Its actual generation will be the minimum of its default generation, and the generations of any previous theorems it subsumes. In any resolution attempt at least one of the two theorems will be from the original goals assumptions, or be derived from an earlier resolution in the evaluation. Duplicates and pure specialisations will be discarded.

Resolution will be attempted on subterms reached through outer universal quantification, and propositional connectives, by specialising the outer quantifications, and by type instantiation on the input theorems *thms* where necessary and possible.

The resulting list of theorems will have all the *thms* removed, and all the old assumptions removed. *MAP_EVERY thmtac* is then applied to the new theorems, and then to the goal. As a special case, ...⊢ *F* is checked for, before any further processing. If present it will be used to prove the goal.

Uses On its own, or in combination with some canonicalisation of the input theorems.

Errors
```
67003 The limit, ?0, must be a positive integer
67004 No resolution occurred
```

SML
```
val basic_resolve_rule: TERM -> THM -> THM -> THM;
```

Description *basic_resolve_rule subterm pos neg* attempts to resolve two theorems that have a common subterm, *subterm*, occurring "positively" in *pos* and "negatively" in *neg*.

Rule

$$\frac{\Gamma \vdash P\ [subterm] \qquad \Delta \vdash N\ [subterm]}{simplify\ (\Gamma, \Delta \vdash P[F] \lor N[T])} \quad basic\_resolve\_rule\ subterm$$

Where *simplify* carries out the simplifications in the predicate calculus where an argument is the constant ⌜*T*⌝ or ⌜*F*⌝, plus a few others.

Errors
```
3031  ?0 is not of type ⌜:BOOL⌝
67009 ?0 is not a subterm of ?1
```

SML
```
val basic_res_extract : RES_DB_TYPE -> THM list;
```

Description This is the extraction function for the basic resolution tool based on *prim_res_rule*. It does no more than return the fourth item of the *RES_DB_TYPE* tuple.

SML

val **basic_res_next_to_process** : *BASIC_RES_TYPE list* −>
BASIC_RES_TYPE list;

Description This takes as the next fragment to process the first fragment which comes from a theorem that subsumed some pre-existing one, and failing that the next one on the list of fragments.

SML

val **basic_res_post** :
(*THM* −> *THM* −> *int*) −>
(*THM list* ∗ *int*) ∗ *RES_DB_TYPE* −>
(*RES_DB_TYPE* ∗ *bool*);

Description This is the post processor for the basic resolution tool based on *prim_res_rule*. The results will be split into their respective conjuncts (if any). Then *basic_res_post subsum* ((*res*, *gen*), *data*) will test each member of *res*, checking for the conclusion *T* or *F*, and then against each member of the theorem list of *data*. In checking one theorem against another it will use *subsum* - discarding the new theorem if the result is *1*, and discarding (with tidying up of *data*) the original if the result is *2*, or keeping both (except for discards from further tests) if the result is *0*, or any other value bar *1* and *2*. *gen* is the default "generation" of the new theorems, except that the fragments for each new theorem will have the minimum generation number of this default generation, and the generation of any theorem in *data* it subsumes.

SML

val **basic_res_pre** : *THM list* −> *THM list* −> *RES_DB_TYPE*;

Description This is the preprocessor for the basic resolution tool based on *prim_res_rule*. The first argument is the set of support theorems, the second argument is the rest of the input theorems. Each theorem will be fragmented, and each fragment added to the appropriate list (i.e. to the third list of the result if in the set of support, and the first list if otherwise). The final theorem list part of the result, *dbdata*, is just the appending of the first list of theorems to the second.

SML

val **basic_res_resolver** : *Unification.SUBS* −> *int* −>
BASIC_RES_TYPE −> *BASIC_RES_TYPE* −> *THM list* ∗ *int*;

Description This is the resolver for the basic resolution tool based on *prim_res_rule*. Resolution seeks to find sufficient term specialisation and type instantiation on both terms to make one of the two term fragments the negation of the other, using *term_unify*. The resolution will not be attempted if the result would involve more resolutions than the "generations" limit. If this can be done then the two original theorems are specialised and instantiated in the same manner and the term fragment cancelled by *basic_resolve_rule*, and the result returned as a singleton list, paired with the default generation of the result. Prior to being returned, any allowed universal quantification will be added back in. In the basic resolution tool the generality of a list of theorems is unnecessary.

The *SUBS* argument is a "scratchpad" for the type unifier. The function keeps track of the number of resolutions used to create the result.

Errors

67001 Neither argument is in the set of support
67002 Cannot resolve the two arguments
67008 term_unify succeeded on ?0 and ?1 but failed to resolve ?2 and ?3

Message is a variant on 67002, included for diagnostic purposes. It will be removed in a more stable product.

SML
```
val basic_res_rule : int -> THM list -> THM list ->
        THM list;
```

Description *basic_res_rule limit sos rest* will resolve the theorems in the set of support and the rest against each other until only theorems with default generation past *limit* can be derived, or until ...⊢ F is derived, or until no further resolution can be done. A input theorem's generation is *0*, and a theorem that is the result of resolution has a default generation of *1* plus the sum of the generations of the resolved theorems. Its actual generation will be the minimum of its default generation, and the generations of any previous theorems it subsumes. In any resolution attempt at least one of the two theorems will belong to the set of support, or be derived from an earlier resolution in the evaluation. Resolution will be attempted on subterms reached through outer universal quantification, and propositional connectives, by specialising the outer quantifications, and by type instantiation where necessary and allowed. Duplicates and pure specialisations in the resulting list will be discarded.

If any of the input theorems have ⌜F⌝ as a conclusion then that theorem is returned as a singleton list.

Uses On its own, or in combination with some canonicalisation of the input theorems.

Errors
```
67003 The limit, ?0, must be a positive integer
67004 No resolution occurred
```

SML
```
val basic_res_subsumption : THM -> THM -> int;
```

Description This returns *1* if the conclusion of the first theorem equals the second's, or is a less general form than the second (i.e. could be produced only by specialising and type instantiating the second theorem). It returns *2* if the second theorem's conclusion is a less general form than the first, and otherwise returns *0*.

SML
```
val basic_res_tac1 : int -> THM list -> TACTIC;
```

Description *basic_res_tac1 limit thms* (*seqasms*, *conc*) will take the theorems gained by *asm-_rule*'ing the assumptions and *thms* as inputs. These theorems will be resolved against each other until only theorems with default generation past *limit* can be derived, or until ...⊢ *F* is derived, or until no further resolution can be done. An assumption's or input theorem's generation is *0*, and a theorem that is the result of resolution has a default generation of *1* plus the sum of the generations of the resolved theorems. Its actual generation will be the minimum of its default generation, and the generations of any previous theorems it subsumes. In any resolution attempt at least one of the two theorems will be from the original goals assumptions, or be derived from an earlier resolution in the evaluation. Duplicates and pure specialisations will be discarded.

Resolution will be attempted on subterms reached through outer universal quantification, and propositional connectives, by specialising the outer quantifications, and by type instantiation on the input theorems *thms* where necessary and possible.

The resulting list of theorems will have all the *thms* removed, and all the old assumptions removed. *MAP_EVERY strip_asm_tac* is then applied to the new theorems, and then to the goal. As a special case, ...⊢ *F* is checked for, before any further processing. If present it will be used to prove the goal.

Uses On its own, or in combination with some canonicalisation of the input theorems.

Errors

67003 The limit, ?0, must be a positive integer
67004 No resolution occurred

SML
```
val basic_res_tac2 : int -> THM list -> TACTIC;
```

Description *basic_res_tac2 limit thms* (*seqasms*, *conc*) will first strip the negated goal into the assumption list. This uses *strip_tac*, except that the negation is pushed through all the outer universals. The assumptions derived from this will become the set of support, the pre-existing assumptions and the input *thms* will be the rest of the theorems. These theorems will be resolved against each other until only theorems with default generation past *limit* can be derived, or until ...⊢ *F* is derived, or until no further resolution can be done. An assumption's or input theorem's generation is *0*, and a theorem that is the result of resolution has a default generation of *1* plus the sum of the generations of the resolved theorems. Its actual generation will be the minimum of its default generation, and the generations of any previous theorems it subsumes. In any resolution attempt at least one of the two theorems will be assumed fragments from the stripped goal, or be derived from an earlier resolution in upon those fragments. Duplicates and pure specialisations will be discarded.

Resolution will be attempted on subterms reached through outer universal quantification, and propositional connectives, by specialising the outer quantifications, and by type instantiation on the input theorems *thms* where necessary and possible.

The tactic will fail unless the resulting list of theorems contains ...⊢ *F*. If present it will be used to prove the goal.

Errors

67003 The limit, ?0, must be a positive integer
67004 No resolution occurred
67014 Failed to prove goal

SML

```
val basic_res_tac3 : int -> THM list -> TACTIC;
```

Description *basic_res_tac3 limit thms* (*seqasms*, *conc*) will take the theorems gained by *asm_rule*'ing the assumptions and *thms* as inputs. These theorems will be resolved against each other until only theorems with default generation past *limit* can be derived, or until ...⊢ F is derived, or until no further resolution can be done. An assumption's or input theorem's generation is *0*, and a theorem that is the result of resolution has a default generation of *1* plus the sum of the generations of the resolved theorems. Its actual generation will be the minimum of its default generation, and the generations of any previous theorems it subsumes. In any resolution attempt at least one of the two theorems will be from the original goals assumptions, or be derived from an earlier resolution in the evaluation. Duplicates and pure specialisations will be discarded.

Resolution will be attempted on subterms reached through outer universal quantification, and propositional connectives, by specialising the outer quantifications, and by type instantiation on the input theorems *thms* where necessary and possible.

The tactic will fail unless the resulting list of theorems contains ...⊢ F. If present it will be used to prove the goal.

Errors

67003	*The limit, ?0, must be a positive integer*
67004	*No resolution occurred*
67014	*Failed to prove goal*

SML

```
val basic_res_tac4 : int -> int list -> int list ->
        THM list -> THM list -> TACTIC;
```

Description *basic_res_tac4 limit sos rest sos_thms rest_thms* (*seqasms*, *conc*) will take the theorems gained by *asm_rule*'ing the numbered assumptions and *thms* as inputs. The "set of support" theorems with be those assumptions noted in the *sos* and those theorems in *sos_thms*, and "the rest" will be those assumptions noted in the *rest*, as well as *rest_thms*. These theorems will be resolved against each other until only theorems with default generation past *limit* can be derived, or until ...⊢ F is derived, or until no further resolution can be done. An assumption's or input theorem's generation is *0*, and a theorem that is the result of resolution has a default generation of *1* plus the sum of the generations of the resolved theorems. Its actual generation will be the minimum of its default generation, and the generations of any previous theorems it subsumes. In any resolution attempt at least one of the two theorems will be from the set of support, or be derived from an earlier resolution in the evaluation. Duplicates and pure specialisations will be discarded.

Resolution will be attempted on subterms reached through outer universal quantification, and propositional connectives, by specialising the outer quantifications, and by type instantiation on the input theorems *thms* where necessary and possible.

The resulting list of theorems will have all the *thms* removed, and all the old assumptions removed. *MAP_EVERY strip_asm_tac* is then applied to the new theorems, and then to the goal. As a special case, ...⊢ F is checked for, before any further processing. If present it will be used to prove the goal.

Uses On its own, or in combination with some canonicalisation of the input theorems.

Errors

67003	*The limit, ?0, must be a positive integer*
67004	*No resolution occurred*
9303	*the index ?0 is out of range*

SML

```
val basic_res_tac : int -> THM list -> TACTIC;
```

Description *basic_res_tac limit thms* (*seqasms*, *conc*) will first strip the negated goal into the assumption list. This uses *strip_tac*, except that the negation is pushed through all the outer universals. The assumptions derived from this will become the set of support, the pre-existing assumptions and the input *thms* will be the rest of the theorems. These theorems will be resolved against each other until only theorems with default generation past *limit* can be derived, or until ...⊢ F is derived, or until no further resolution can be done. An assumption's or input theorem's generation is *0*, and a theorem that is the result of resolution has a default generation of *1* plus the sum of the generations of the resolved theorems. Its actual generation will be the minimum of its default generation, and the generations of any previous theorems it subsumes. In any resolution attempt at least one of the two theorems will be assumed fragments from the stripped goal, or be derived from an earlier resolution in upon those fragments. Duplicates and pure specialisations will be discarded.

Resolution will be attempted on subterms reached through outer universal quantification, and propositional connectives, by specialising the outer quantifications, and by type instantiation on the input theorems *thms* where necessary and possible.

The resulting list of theorems will have all the *thms* removed, all the theorems derived from stripping and negating the goal removed, and all the old assumptions removed. *MAP_EVERY strip_asm_tac* is then applied to the new theorems, and then to the goal. As a special case, ...⊢ F is checked for, before any further processing. If present it will be used to prove the goal.

Uses On its own, or in combination with some canonicalisation of the input theorems.

Errors

67003 The limit, ?0, must be a positive integer
67004 No resolution occurred

SML

```
val prim_res_rule :
        (THM list -> THM list -> ('a list * 'a list * 'a list * 'b)) -> (* preprocessor *)
        ('a -> 'a -> 'c) -> (* the resolver function *)
        (('c * ('a list * 'a list * 'a list * 'b)) ->
                (('a list * 'a list * 'a list * 'b) * bool)) -> (* postprocessor *)
        ('a list -> 'a list) -> (* next item to process *)
        ('a list * 'a list * 'a list * 'b -> THM list) -> (* extract results *)
        THM list -> (* input set of support theorems *)
        THM list -> (* input other theorems *)
        THM list;   (* final outcome *)
```

Description *prim_res_rule prep reso postp next extract limit sos rest* works as follows:

- If any of the input theorems have ⌜*F*⌝ as a conclusion then that theorem is returned as a singleton list.
- Evaluate *prep sos rest*, and set (*against*, *tried*, *toprocess*, *dbdata*) to this.
- Attempt resolutions, choosing the head of *toprocess* against the head of *against*. Commonly, the head of *toprocess* should be the first fragment from the set of support, *against* is all the non-set of support fragments, plus the head of *toprocess*, and *tried* is empty.
- The resolver will usually return a list of theorems, and perhaps some further data. When a resolution attempt returns a list of theorems, *res*, (resolution failures should not occur, just []), evaluate *postp* (*res*, (*against*, *tried*, *toprocess*, *dbdata*)) to extract a new (*against*, *tried*, *toprocess*, *dbdata*), and *halt*. It is up to the postprocessor to move the head of *against* either to *tried* or just thrown away.
- If *halt* is true (e.g. have proved ...⊢*F*), or the *toprocess* list is empty then return as a result of the call *extract* (*against*, *tried*, *toprocess*, *dbdata*).
- If *halt* is false, then continue with the new data. If *against* is [] then the head of *toprocess* is dropped, and the new list of things to process generated by *next* (*tl toprocess*), the new head of this cons'd to *done* and *against* is set to *done* reversed, and then *done* set to [].

Errors

67004 No resolution occurred
67010 Postprocessor corrupted processing

SML

```
val term_unify : Unification.SUBS -> (TYPE list) -> (TERM list) ->
        (TERM * TERM list * TYPE list) *
        (TERM * TERM list * TYPE list) ->
        ((TYPE * TYPE) list * (TERM * TERM) list) *
        ((TYPE * TYPE) list * (TERM * TERM) list);
```

Description This is a method of unifying two subterms in the context of limitations on both type instantiation and term specialisation. The *SUBS* argument is a "scratchpad" for the type unification function, based on *Unification.unify*. The initial type list is a list of type variables to avoid in generating new names, and the initial term list a list of term variables to likewise avoid. The other two input arguments are each a tuple of: a term to unify, a list of variables in the term that may be specialised, and a list of types for which instantiation is allowed. If the two terms can be unified then the function returns two tuples, referring to each of the two input tuples. Each tuple is a list of type instantiations and a list of term specialisations, which pair the original before type instantiation, and the result, type instantiated.

Errors

```
3007   ?0 is not a term variable
3019   ?0 is not a type variable
67005  Cannot unify ?0 and ?1
67006  Cannot unify ?0 and ?1 as cannot specialise ?2
67012  Cannot unify ?0 and ?1 as would cause a loop
```

As as errors of *Unification.unify*.

7.7 Proof Contexts

SML

```
signature ProofContext = sig
```

Description This provides the basic tools for handling equational and proof contexts. To keep them short, the names in the structure are heavily abbreviated. The abbreviations used are:

pc(s)	proof context(s)
rw	rewriting
cs	constant specification
∃	existential theorem prover
pr	*prove_tac* and related tools
sg	goal stripping
st	theorem stripping
cd	clausal definition
vs	variable structure
ad	application data
net	discrimination net
eqn_cxt	equational context
nd	dictionary of discrimination nets (and sources)
canon(s)	theorem canonicalisation
mmp	matching mp rule
eqm	equation matcher

SML

```
(* proof context key "initial" *)
```

Description This is the initial proof context, formed with empty lists and other default values. It thus has no default rewriting or stripping theorems. The rewriting canonicalisation is the identity. The automated existence prover fails on any input. The matching modus ponens rule is *Nil*.

SML

```
type EQN_CXT;
```

Description This is the type of equational contexts. An equational context is a list of conversions, each paired with term index. It represents a statement of how to rewrite a term to result in an equational theorem, guided by the outermost form of the term to be rewritten, which is matched against the term index of each conversion. It is used to create a single conversion via *eqn_cxt_conv* (q.v.).

A theorem may be converted into a member of an equational context by *thm_eqn_cxt*. A pre-existing conversion may be converted by determining the term index that matches at least all terms that the conversion must work on (see *net_enter* for details), and pair it with the conversion.

```
type EQN_CXT = (TERM * CONV) list;
```

Note that equational contexts can be merged by appending. An equational context may be transformed into a conversion discrimination net by *make_net* or *list_net_enter*(q.v.).

SML
```
val asm_prove_tac : THM list -> TACTIC;
```

Description This tactic is an automatic proof procedure appropriate to the current proof context.

At the point of applying this tactic to its theorems it will access the current setting of proof context field *pr_tac*, apply it to the theorem list immediately, and then to the goal when available (i.e. the result is partially evaluated with only the list of theorems).

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{current\_ad\_pr\_tac\ ()\ thms\ (\{\ \Gamma\ \},\ t)} \quad \begin{array}{l} asm\_prove\_tac \\ thms \end{array}$$

See Also *PC_T1* to defer accessing the proof context until application to the goal; *prove_tac* for the form that does not react to the presence of assumptions.

Errors

51021 The current proof context was created in theory ?0 at a point now either not in scope, deleted or modified

and as the proof context setting.

SML
```
val asm_prove_∃_tac : TACTIC;
```

Description This tactic is an automatic proof procedure for existential proofs, appropriate to the current proof context.

At the point of applying this tactic to a goal it will access the current setting of proof context field *prove_∃*, apply it to the goal using *conv_tac*.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\begin{array}{c} conv\_tac\ (current\_ad\_cs\_\exists\_conv\ ()) \\ (\{\ \Gamma\ \},\ t) \end{array}} \quad \begin{array}{l} asm\_prove\_\exists\_tac \\ thms \end{array}$$

See Also *prove_∃_tac* that does not react to any assumptions that are present.

Errors

51021 The current proof context was created in theory ?0 at a point now either not in scope, deleted or modified

and as the proof context setting.

SML
```
val commit_pc : string -> unit;
```

Description This commits a record of the proof context database, preventing further change, and allowing it to be used in the creation of further records. The context must be loadable at the point of committing (i.e. was created at a point now in scope), and after committal the proof context can only be loaded at a point when the point of committal is in scope, rather than the point of its initial creation (i.e. doing *new_pc*).

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed

SML
```
val current_ad_mmp_rule : unit -> (THM -> THM -> THM) OPT;
```

Description This function returns the application data of the current proof context for the matching modus ponens rule as used by tools such as *forward_chain_rule*.

See Also *set_mmp_rule* for user data.

Errors
```
51021 The current proof context was created in theory ?0 at a
      point now either not in scope, deleted or modified
```

SML
```
val current_ad_pr_conv : unit -> (THM list -> CONV) OPT;
```

Description These functions returns the application data of the current proof context to the proof contexts for *prove_conv*.

See Also *set_pr_conv* for user data.

Errors
```
51021 The current proof context was created in theory ?0 at a
      point now either not in scope, deleted or modified
```

SML
```
val current_ad_pr_tac : unit -> (THM list -> TACTIC) OPT;
```

Description This function returns the application data of the current proof context for *prove_tac*.

See Also *set_pr_tac* for user data.

Errors
```
51021 The current proof context was created in theory ?0 at a
      point now either not in scope, deleted or modified
```

SML
```
val current_ad_rw_eqm_rule : unit -> (THM -> TERM * CONV) OPT;
```

Description This function returns the application data of the current proof context for the equation matcher as used by the rewriting tools.

See Also *set_rw_eqm_rule* for user data.

Errors
```
51021 The current proof context was created in theory ?0 at a
      point now either not in scope, deleted or modified
```

SML

```
val delete_pc_fields : string list -> string -> unit;
```

Description *delete_pc_fields fields key* empties (sets to the value of proof context "initial") the named fields, *fields* of the proof context with key *key*. If any field is divided into subfields, this deletion includes deleting the subfields of the field gained from merging in other proof contexts, as well as the proof context's "own" subfield.

Valid field names are:

```
"rw_eqn_cxt","rw_canons","st_eqn_cxt","sc_eqn_cxt",
"cs_∃_convs","∃_cd_thms","∃_vs_thms","pr_tac","pr_conv",
"nd_entries", "mmp_rule"
```

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
51019 There is no field called ?0

SML

```
val delete_pc : string -> unit;
```

Description This deletes a record from the proof context database. The record with key "initial" may not be deleted.

Errors

51010 There is no proof context with key ?0
51012 Initial proof context may not be deleted

SML

```
val eqn_cxt_conv : EQN_CXT -> CONV;
```

Description This function creates a single conversion from an equational context. This is done via *make_net* and *net_lookup*(q.v). There is a *CHANGED_C* wrapped around each conversion in the equational context.

Errors

51005 Equational context gave no conversions that succeeded for ?0

SML
```
val EXTEND_PC_C1 : string -> ('a -> CONV) -> 'a -> CONV;
val EXTEND_PCS_C1 : string list -> ('a -> CONV) -> 'a -> CONV;
```

Description *EXTEND_PC_C context conv arg* will apply conversion *conv arg* in the proof context obtained by merging the proof context with key *context* into the current proof context. The named context is used as it is at the point of applying the rule to the argument. The *pr_tac*, *pr_conv* and *mpp_rule* fields are taken from the named proof context. This is done via pushing and popping on the proof context stack.

EXTEND_PCS_C1 takes a list of proof contexts instead, merged as if by, e.g. *push_extend_pcs*.

See Also *PC_C*

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
```

and as the errors of the conversion. The previous proof context is restored, even if the conversion fails.

SML
```
val EXTEND_PC_C : string -> CONV -> CONV;
val EXTEND_PCS_C : string list -> CONV -> CONV;
```

Description *EXTEND_PC_C context conv* will apply conversion *conv* to a term in the proof context obtained by merging the proof context with key *context* into the current proof context. The named context is used as it is at the point of applying the conversion to a term. The *pr_tac*, *pr_conv* and *mpp_rule* fields are taken from the named proof context. This is done via pushing and popping on the proof context stack.

EXTEND_PCS_C takes a list of proof contexts instead, merged as if by, e.g. *push_extend_pcs*

Note that when using this functions that the standard rewriting functions (obvious candidates for this function) access the current proof context at the point of being given their theorem list argument: see *EXTEND_PC_C1* for a method of avoiding this.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
```

and as the errors of the conversion. The previous proof context is restored, even if the conversion fails.

SML
```
val extend_pc_rule1 : string -> ('a -> 'b -> THM) -> 'a -> 'b -> THM;
val extend_pcs_rule1 : string list -> ('a -> 'b -> THM) -> 'a -> 'b -> THM;
```

Description *extend_pc_rule1 context rule arg1 arg2* will apply rule *rule arg1* to *arg2* in the proof context obtained by merging the proof context with key *context* into the current proof context. The named context is used as it is at the point of applying the rule to the argument. The *pr_tac*, *pr_conv* and *mpp_rule* fields are taken from the named proof context. This is done via pushing and popping on the proof context stack.

extend_pcs_rule1 takes a list of proof contexts instead, merged as if by, e.g. *push_extend_pcs*.

See Also *pc_rule*

Errors

51010	*There is no proof context with key ?0*
51014	*Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified*

and as the errors of the rule. The previous proof context is restored, even if the rule fails.

SML
```
val extend_pc_rule : string -> ('a -> THM) -> ('a -> THM);
val extend_pcs_rule : string list -> ('a -> THM) -> ('a -> THM);
```

Description *extend pc_rule context rule* will apply rule *rule* to its argument in the proof context obtained by merging the proof context with key *context* into the current proof context. The named context is used as it is at the point of applying the rule to the argument. The *pr_tac*, *pr_conv* and *mpp_rule* fields are taken from the named proof context. This is done via pushing and popping on the proof context stack.

extend_pcs_rule takes a list of proof contexts instead, merged as if by, e.g. *extend_merge_pcs*

Note that when using this functions that the standard rewriting functions (obvious candidates for this function) access the current proof context at the point of being given their theorem list argument: see *extend_pc_rule1* for a method of avoiding this.

Errors

51010	*There is no proof context with key ?0*
51014	*Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified*

and as the errors of the rule. The previous proof context is restored, even if the rule fails.

SML
```
val EXTEND_PC_T1 : string -> ('a -> TACTIC) -> 'a -> TACTIC;
val EXTEND_PCS_T1 : string list -> ('a -> TACTIC) -> 'a -> TACTIC;
```

Description *EXTEND_PC_T1 context tac arg* will apply tactic *tac arg* to a goal, and evaluate the proof, in the proof context obtained by merging the proof context with key *context* into the current proof context. The named context is used as it is at the point of applying the rule to the argument. The *pr_tac*, *pr_conv* and *mpp_rule* fields are taken from the named proof context. This is done via pushing and popping on the proof context stack.

EXTEND_PCS_T1 takes a list of proof contexts instead, merged as if by, e.g. *push_extend_pcs*.

See Also *PC_T*

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
```

and as the errors of the tactic. The previous proof context is restored, even if the tactic application or proof fails.

SML
```
val EXTEND_PC_T : string -> TACTIC -> TACTIC;
val EXTEND_PCS_T : string list -> TACTIC -> TACTIC;
```

Description *EXTEND_PC_T context tac* will apply tactic *tac* to a goal, and evaluate its proof, in the proof context obtained by merging the proof context with key *context* into the current proof context. The named context is used as it is at the point of applying the tactic to a goal. The *pr_tac*, *pr_conv* and *mpp_rule* fields are taken from the named proof context. This is done via pushing and popping on the proof context stack.

EXTEND_PCS_T_T takes a list of proof contexts instead, merged as if by, e.g. *push_extend_pcs*

Note that when using this functions that the standard rewriting functions (obvious candidates for this function) access the current proof context at the point of being given their theorem list argument: see *EXTEND_PC_T1* for a method of avoiding this.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
```

and as the errors of the tactic. The previous proof context is restored, even if the tactic application or proof fails.

SML
```
val force_delete_theory : string -> unit;
```

Description *force_delete_theory thy* attempts to delete theory *thy* and all its descendants. If *thy* is in scope, then the function will change the current theory to the first theory that it can in the list returned by *get_parents thy*; (there may be none, in which case the function fails). It will then determine whether *thy* and its descendants can all be deleted: in particular it checks that none of them are locked (see *lock_theory*) or are a read-only ancestor.

The function indicates:

- whether the current theory has been deleted, and if so states the new current theory,
- the list of theories that have been deleted (unless this is just the requested theory, and is also not the current theory).

Further, all proof contexts created in now deleted theories will also be deleted (but the current proof context will remain unchanged).

Errors
```
51002 Cannot open any of the parent theories, ?0, of the named theory, ?1
51003 Will not be able to delete theories ?0, so no deletions made
51004 Unexpectedly unable to delete any of ?0
51006 Cannot open the parent theory, ?0, of the named theory, ?1
51007 Will not be able to delete theory ?0, so no deletions made
51008 Named theory, ?0, has no parents
```

Error 51004 will be raised by *error* rather than *fail*, as it shouldn't happen.

SML
```
val get_current_pc : unit -> (string list * string);
```

Description Returns the key(s) of the entries from which the current proof context was copied, and the theory in which the single proof context was created. If the theory has since been placed out of scope, deleted or if the definition level becomes deleted, e.g. because an axiom or definition has been deleted, then this will output

```
(["context name"],"theory name (out of scope, deleted, or modified)")
```

Note that the key may no longer access a proof context in the database identical to the current proof context.

Merged proof contexts upon the stack (from *push_merge_pcs* and *set_merge_pcs*) will have the list of names of the constituent proof contexts, singleton contexts will have singleton lists.

See Also *get_stack_pcs*

SML
```
val get_pcs : unit -> (string * string) list;
```

Description This lists the names of the proof contexts held in the proof context database, and the theory that was current at their time of creation. If the theory has since been deleted or if the definition level becomes deleted, e.g. because an axiom or definition has been deleted, then this will output ("*context name*","*theory name (out of scope, deleted, or modified)*")

See Also *get_stack_pcs*, *get_current_pc*.

SML

```
val get_stack_pcs : unit -> (string list * string) list;
```

Description This lists the keys of the proof contexts held in the proof context stack, and the theory that was current at their time of creation. If a proof context is pushed onto the stack by, e.g. *push_pc*, the "keys" will be the singleton list of the name of the source proof context. If a proof context is pushed onto the stack by, e.g. *push_merge_pcs*, the "keys" will be the list of the names of the source proof contexts. If the theory has since been deleted or if the definition level becomes deleted, e.g. because an axiom or definition has been deleted, then this will output

```
(["context name"],"theory name (out of scope, deleted, or modified)")
```

The head of the list returned is the current proof context, as also displayed by *get_current_pc*.

SML

```
val merge_pcs : string list -> string -> unit;
```

Description *merge_pcs keys tokey* takes a list of committed proof contexts named by *keys*, and merges their fields into proof context *tokey*'s fields, discarding duplicates. For each field that has subfields the lists of subfields from each proof context are appended, discarding subfields with duplicate keys, and if a field is not divided into subfields, then the proof contexts fields are appended, discarding duplicates. The *pr_conv*, *pr_tac* and *mmp_rule* fields take the value of the last proof context in the list that has the field set.

Failure to extract any proof context for merging will result in the proof context *tokey* being unchanged.

See Also *merge_pc_fields*, *delete_pc_fields*

Errors

```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
51017 Proof context ?0 has not been committed
```

SML

```
val merge_pc_fields : {context:string,fields:string list}list -> string -> unit;
```

Description *merge_pc_fields fields tokey* merges the fields noted for each committed proof context in *fields* into proof context *tokey*'s fields, discarding duplicates. Merging for each field that has subfields the lists of subfields is appending the proof contexts fields, discarding subfields with duplicate keys, and if a field is not divided into subfields, then the proof contexts fields are appended, discarding duplicates. Each of the *pr_conv*, *pr_tac* and *mmp_rule* fields take the value from the last proof context whose list of field names includes that field and which has the field set.

Failure to extract any proof context for merging will result in the proof context *tokey* being unchanged.

Valid field names are:

```
"rw_eqn_cxt","rw_canons","st_eqn_cxt","sc_eqn_cxt",
"cs_∃_convs","∃_cd_thms","∃_vs_thms","pr_tac","pr_conv",
"nd_entries", "mmp_rule"
```

See Also *delete_pc_fields* and *merge_pcs*, which used together in a particular order can give the same functionality as this function.

Errors

```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
51017 Proof context ?0 has not been committed
51019 There is no field called ?0
```

SML

```
val new_pc : string -> unit;
```

Description *new_pc new* creates a new record in the proof context database, with key *new*. The fields of the proof context are set to default values. A note will be made of the current theory, and its current definition level at the time of creation, and an error will be raised if an attempt is made to push the new proof context (see *push_pc*) when that theory is not in scope, or when the definition level has been recorded as deleted. The definition level will be recorded as deleted if, e.g., some definition or axiom that was in scope in the original theory has since been deleted.

One responsibility of the creator of a proof context is to ensure that the theorems used within, or created by, the new context are also in scope: this is not automatically checked.

Errors

```
51011 There is already a proof context with key ?0
```

SML
```
val PC_C1 : string -> ('a -> CONV) -> 'a -> CONV;
val MERGE_PCS_C1 : string list -> ('a -> CONV) -> 'a -> CONV;
```

Description *PC_C context conv arg* will apply conversion *conv arg* in the proof context with key *context*, using the named context as it is at the point of applying the conversion to a term. This is done via pushing and popping on the proof context stack.

MERGE_PCS_C1 takes a list of proof contexts instead, merged as if by, e.g. *push_merge_pcs*.

See Also *PC_C*

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified

and as the errors of the conversion. The previous proof context is restored, even if the conversion fails.

SML
```
val PC_C : string -> CONV -> CONV;
val MERGE_PCS_C : string list -> CONV -> CONV;
```

Description *PC_C context conv* will apply conversion *conv* to a term in the proof context with key *context*, using the named context as it is at the point of applying the conversion to a term. This is done via pushing and popping on the proof context stack.

MERGE_PCS_C takes a list of proof contexts instead, merged as if by, e.g. *push_merge_pcs*

Note that when using this functions that the standard rewriting conversions (obvious candidates for this function) access the current proof context at the point of being given their theorem list argument: see *PC_C1* for a method of avoiding this.

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified

and as the errors of the conversion. The previous proof context is restored, even if the conversion fails.

SML
```
val pc_rule1 : string -> ('a -> 'b -> THM) -> 'a -> 'b -> THM;
val merge_pcs_rule1 : string list -> ('a -> 'b -> THM) -> 'a -> 'b -> THM;
```

Description *pc_rule context rule arg1 arg2* will apply rule *rule arg1* to *arg2* in the proof context with key *context*, using the named context as it is at the point of applying the rule to argument *arg2*. This is done via pushing and popping on the proof context stack.

merge_pcs_rule1 takes a list of proof contexts instead, merged as if by, e.g. *push_merge_pcs*.

See Also *pc_rule*

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified

and as the errors of the rule. The previous proof context is restored, even if the rule fails.

SML
```
val pc_rule : string -> ('a -> THM) -> ('a -> THM);
val merge_pcs_rule : string list -> ('a -> THM) -> ('a -> THM);
```

Description *pc_rule context rule* will apply rule *rule* to its argument in the proof context with key *context*, using the named context as it is at the point of applying the rule to the argument. This is done via pushing and popping on the proof context stack.

merge_pcs_rule takes a list of proof contexts instead, merged as if by, e.g. *push_merge_pcs*

Note that when using this functions that the standard rewriting functions (obvious candidates for this function) access the current proof context at the point of being given their theorem list argument: see *pc_rule1* for a method of avoiding this.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
```

and as the errors of the rule. The previous proof context is restored, even if the rule fails.

SML
```
val PC_T1 : string -> ('a -> TACTIC) -> 'a -> TACTIC;
val MERGE_PCS_T1 : string list -> ('a -> TACTIC) -> 'a -> TACTIC;
```

Description *PC_T1 context tac arg* will apply tactic *tac arg* to a goal, and evaluate the proof, in the proof context with key *context*, using at both times the named context as it is at the point of applying the tactic to a goal. This is done via pushing and popping on the proof context stack.

MERGE_PCS_T1 takes a list of proof contexts instead, merged as if by, e.g. *push_merge_pcs*.

See Also *PC_T*

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
```

and as the errors of the tactic. The previous proof context is restored, even if the tactic application or proof fails.

SML
```
val PC_T : string -> TACTIC -> TACTIC;
val MERGE_PCS_T : string list -> TACTIC -> TACTIC;
```

Description *PC_T context tac* will apply tactic *tac* to a goal, and evaluate its proof, in the proof context with key *context*, using at both times the named context as it is at the point of applying the tactic to a goal. This is done via pushing and popping on the proof context stack.

PCS_MERGE_T takes a list of proof contexts instead, merged as if by, e.g. *push_merge_pcs*

Note that when using this functions that the standard rewriting functions (obvious candidates for this function) access the current proof context at the point of being given their theorem list argument: see *PC_T1* for a method of avoiding this.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
```

and as the errors of the tactic. The previous proof context is restored, even if the tactic application or proof fails.

SML

val **pending_push_merge_pcs** : *string list −> unit −> unit*;
val **pending_push_extend_pcs** : *string list −> unit −> unit*;

Description *pending_push_merge_pcs* takes a snapshot of the result of merging the named proof contexts, and returns a function that, when applied to () stacks the previous proof context, and and sets the current proof context of the system to this snapshot.

pending_push_extend_pcs takes a snapshot of the result of merging the named proof contexts with the current proof context and then behaves just like *pending_push_merge_pcs*.

Merged proof contexts upon the stack will have *current_ad_names* giving the list of names of the constituent proof contexts, singleton contexts will have singleton lists. The proof contexts used need not have been committed. The *pr_conv*, *pr_tac* and *mmp_rule* fields take the value of the last proof context in the list that has the field set.

The proof context must be in scope both at the time of the snapshot, and at the time of pushing on the stack.

This provides a method of being independent of changes to uncommitted proof contexts, or proof context deletions.

See Also *push_merge_pc*

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified
51020 Must be at least one key in list

SML

val **pending_push_pc** : *string −> unit −> unit*;
val **pending_push_extend_pc** : *string −> unit −> unit*;

Description *pending_push_pc* takes a snapshot of the named proof context, and returns a function that, when applied to () : *unit* stacks the previous "current" proof context, and sets the current proof context of the system to this snapshot.

pending_push_extend_pc takes a snapshot of the result of merging the named proof context with the current proof context and then behaves just like *pending_push_merge_pc*.

The proof context must be in scope both at the time of the snapshot, and at the time of pushing on the stack.

This provides a method of being independent of changes to uncommitted proof contexts, or proof context deletions.

See Also *push_pc*

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified

SML

```
val pending_reset_pc_database : unit -> unit -> unit;
```

Description This function, applied to () takes a snapshot of the proof context database, and returns a function that, if applied to () will restore the proof context database to the snapshot.

This function is particularly useful in initialising child databases, and in conjunction with *pending-_reset_pc_stack* and *pending_reset_pc_evaluators*.

Note that a named proof context on the proof context stack is never taken as more than an echo of the item with that name (if any) of proof context database, and this function in particular, though not alone, is responsible for the possible differences.

SML

```
val pending_reset_pc_stack : unit -> unit -> unit;
```

Description This function, applied to () takes a snapshot of the proof context stack, and returns a function that, if applied to () will restore the proof context stack to the snapshot.

Uses This function is particularly useful in initialising child databases, and in conjunction with *pending_reset_pc_database* and *pending_reset_pc_evaluators*.

SML

```
val pending_reset_pc_evaluators : unit -> unit -> unit;
```

Description This function, applied to () takes a snapshot of the proof context evaluators (e.g. the one set by *pp′set_eval_ad_∃_vs_thms* , and returns a function that, if applied to () will restore the proof context evaluators to the snapshot.

Uses This function is particularly useful in initialising child databases, and in conjunction with *pending_reset_pc_database*, and *pending_reset_pc_stack*

SML

```
val pop_pc : unit -> unit;
```

Description This function unstacks the top of the proof context stack, and sets the current proof context of the system to it. There will always be a current proof context, though it may be the trivial "initial" proof context.

This function may make an out of scope proof context the current proof context.

See Also *push_pc*, *set_pc*, *push_merge_pcs*, *set_merge_pcs*

Errors

```
51001 The proof context stack is empty
```

SML

```
val pp′set_eval_ad_rw_net : (EQN_CXT -> CONV NET) -> unit;
val current_ad_rw_net : unit -> CONV NET;
```

Description These functions provide the interface to the initial conversion net for rewriting (see e.g. *rewrite_tac*) held in the application data of a proof context. The first sets the evaluator, the second extracts the field in the current proof context.

See Also *set_rw_eqn_cxt* for the associated user data.

Errors

```
51021 The current proof context was created in theory ?0 at a
      point now either not in scope, deleted or modified
```

SML
```
val pp'set_eval_ad_rw_canon : ((THM -> THM list) list -> (THM -> THM list))
        -> unit;
val current_ad_rw_canon : unit -> THM -> THM list;
```

Description These functions provide the interface to the canonicalisation function applied to rewriting theorems (see e.g. *rewrite_tac*) held in the application data of a proof context. The proof context is accessed after providing the theorem. The first sets the evaluator, the second extracts the field in the current proof context.

See Also *set_rw_canons* for the associated user data.

Errors
```
51021  The current proof context was created in theory ?0 at a
       point now either not in scope, deleted or modified
```

SML
```
val pp'set_eval_ad_st_conv : (EQN_CXT -> CONV) -> unit;
val current_ad_st_conv : unit -> CONV;
```

Description These functions provide the interface to the conversion for stripping theorems into the assumption list (see e.g. *strip_tac*) held in the application data of a proof context. The proof context is accessed before provision of a term. The first sets the evaluator, the second extracts the field in the current proof context.

See Also *set_st_conv* for the associated user data.

Errors
```
51021  The current proof context was created in theory ?0 at a
       point now either not in scope, deleted or modified
```

SML
```
val pp'set_eval_ad_sc_conv : (EQN_CXT -> CONV) -> unit;
val current_ad_sc_conv : unit -> CONV;
```

Description These functions provide the interface to the conversion for stripping goal conclusions (see e.g. *strip_tac*) held in the application data of a proof context. The proof context is accessed before provision of a term. The first sets the evaluator, the second extracts the field in the current proof context.

See Also *set_sg_conv* for the associated user data.

SML

```
val pp'set_eval_ad_nd_net :
        (string -> (TERM * (TERM -> THM)) list ->
                (TERM -> THM) NET) -> unit;
val current_ad_nd_net : string -> (TERM -> THM) NET;
```

Description These functions provide the interface to the additional dictionary of discrimination nets held in the application data of a list of proof contexts.

The application data is generated by taking, for each key in at least one of the dictionaries in the appropriate subfields of the proof context, the appended lists of all the entries for that key in any of the subfields of the proof context. To this is applied the evaluator set by *pp'set_eval_ad_nd_net* first applied to the dictionary key. The result is used as an entry, using the same dictionary key, in the resulting dictionary of nets. The default evaluator will just use *make_net* on each list of sources.

current_ad_nd_net key returns the net indexed by the key *key* in the current proof context. If no entry exists it returns the empty net *empty_net*. Note that the returned net can be viewed as something of type *EQN_CXT*, and made into a conversion by *eqn_cxt_conv*.

Uses For extending the proof context mechanisms. Though available to the end user, and indeed intended for use by the sophisticated user, the proof context mechanisms (as opposed to proof contexts) should be extended under ICL direction.

See Also *set_nd_entry* for the associated user data.

SML

```
val pp'set_eval_ad_cs_∃_convs : (CONV list -> CONV)
        -> unit;
val current_ad_cs_∃_conv : unit -> CONV;
```

Description These functions provide the interface to the existence prover for constant specifications (see *const_spec*) held in the application data of a proof context. The proof context is accessed before provision of a term. The first sets the evaluator, the second extracts the field in the current proof context.

See Also *set_cs_∃_rule* for the associated user data.

Errors

```
51015 No automated existence prover in the current proof context succeeds
51021 The current proof context was created in theory ?0 at a
        point now either not in scope, deleted or modified
```

SML

```
val pp'set_eval_ad_∃_cd_thms : (THM list ->
        (TERM list * int list * TYPE * (TERM list)list * THM) list) -> unit;
val current_ad_∃_cd_thms : unit ->
        (TERM list * int list * TYPE * (TERM list)list * THM) list;
```

Description These functions provide the interface to the clausal definition theorem information for the existence prover *prove_∃_conv*. See *evaluate_∃_cd_thms* for details upon the form of the information. The first sets the evaluator, the second extracts the field in the current proof context.

See Also *set_∃_cd_thms* for the associated user data.

Errors

```
51021 The current proof context was created in theory ?0 at a
        point now either not in scope, deleted or modified
```

SML
```
val pp'set_eval_ad_∃_vs_thms : ((string * (TERM list * THM)) list ->
        (string * (TERM list * THM)) list) -> unit;
val current_ad_∃_vs_thms : unit ->
        (string * (TERM list * THM)) list;
```

Description These functions provide the interface to the application data variable structure information for the existence prover *prove_∃_conv*. The first sets the evaluator, the second extracts the field in the current proof context.

See Also *set_∃_vs_thms* for user data.

Errors

51021 The current proof context was created in theory ?0 at a point now either not in scope, deleted or modified

SML
```
val prove_conv : THM list -> CONV;
```

Description This conversion is an automatic proof procedure appropriate to the current proof context.

At the point of applying this conversion to its theorems it will access the current setting of proof context field *pr_conv*, applying the result to the theorem list immediately, and then to the term when available (i.e. the result is partially evaluated with only the list of theorems).

Conversion

$$\frac{}{current\_ad\_pr\_conv\ ()\ thms\ \ulcorner t \urcorner} \quad \begin{array}{l} prove\_conv \\ thms\ \ulcorner t \urcorner \end{array}$$

See Also *PC_C1* to defer accessing the proof context until application to the term.

Errors

51021 The current proof context was created in theory ?0 at a point now either not in scope, deleted or modified

and as the proof context setting.

SML
```
val prove_rule : THM list -> TERM -> THM;
```

Description This rule is an automatic proof procedure appropriate to the current proof context.

At the point of applying this rule to its theorem list it will access the current setting of proof context field *pr_conv*, apply it to the theorem list immediately, and then to the term when available (i.e. the result is partially evaluated with only the list of theorems), and then, if the resulting theorem is '⊢*term'* ⇔ *T*' (with no assumptions) where *term* is α-convertible to *term'*, then apply *⇔_t_elim*, and otherwise fail.

Rule

$$\frac{}{\vdash tm} \quad \begin{array}{l} prove\_rule \\ thms\ \ulcorner tm \urcorner \end{array}$$

See Also *pc_rule1* to defer accessing the proof context until application to the term.

Errors

51021 The current proof context was created in theory ?0 at a point now either not in scope, deleted or modified

51022 Result of applying conversion to ?0, which was ?1, not of form: '⊢ input ⇔ T'

and as the proof context setting.

SML

val **prove_∃_conv** : *CONV*;

Description This conversion is an automatic proof procedure for existential proofs, appropriate to the current proof context.

At the point of applying this conversion to a term it will access the current setting of proof context field *cs_∃_conv*, apply it to the theorem list, and then to the term.

The resulting theorem is not checked as having its L.H.S. being the input term.

Conversion

$$\frac{}{current\_ad\_cs\_\exists\_conv\ ()\ \ulcorner t \urcorner} \quad \begin{array}{l} prove\_\exists\_conv \\ \ulcorner t \urcorner \end{array}$$

Errors

51021 The current proof context was created in theory ?0 at a point now either not in scope, deleted or modified

and as the proof context setting.

SML

val **prove_∃_rule** : *TERM −> THM*;

Description This rule is an automatic proof procedure for existential proofs, appropriate to the current proof context.

At the point of applying this rule to a term *term* it will access the current setting of proof context field *cs_∃_conv*, apply it to the term, and then, if the resulting theorem is '$\vdash term' \Leftrightarrow T$' (with no assumptions) where *term* is α-convertible to $term'$, then apply $\Leftrightarrow$_*t_elim*, and otherwise fail.

Rule

$$\frac{}{\vdash tm} \quad \begin{array}{l} prove\_\exists\_rule \\ \ulcorner tm \urcorner \end{array}$$

Errors

51021 The current proof context was created in theory ?0 at a point now either not in scope, deleted or modified

51022 Result of applying conversion to ?0, which was ?1, not of form: '⊢ input ⇔ T'

and as the proof context setting.

SML
```
val push_extend_pcs : string list -> unit;
val set_extend_pcs : string list -> unit;
```

Description *push_extend_pcs* stacks the previous "current" proof context, and and then merges the proof contexts with the given keys into the current proof context. *set_extend_pcs* merges the proof contexts with the given keys into the previous current proof context without changing the stack.

Merged proof contexts upon the stack will have *current_ad_names* giving the list of names of the constituent proof contexts, singleton contexts will have singleton lists. The proof contexts used need not have been committed.

The *pr_conv*, *pr_tac* and *mmp_rule* fields take the value of the last proof context in the list that has the field set.

The current proof context is accessed by the functions prefixed *current_ad_*, and by *get_current-_pc*.

See Also *pop_pc*, *push_merge_pcs*, *set_merge_pcs*

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51020 Must be at least one key in list
```

SML
```
val push_extend_pc : string -> unit;
val set_extend_pc : string -> unit;
```

Description *push_extend_pc* stacks the previous "current" proof context, and and then merges the proof context with the given key into the current proof context. *set_extend_pcs* merges the proof context with the given key into the current proof context without changing the stack.

Merged proof contexts upon the stack will have *current_ad_names* giving the list of names of the constituent proof contexts. The proof context used need not have been committed.

The *pr_conv*, *pr_tac* and *mpp_rule* fields take the value from the named proof context.

The current proof context is accessed by the functions prefixed *current_ad_*, and by *get_current-_pc*.

See Also *pop_pc*, *push_pc*, *set_pc*

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51020 Must be at least one key in list
```

SML
```
val push_merge_pcs : string list -> unit;
val set_merge_pcs : string list -> unit;
```

Description *push_merge_pcs* stacks the previous "current" proof context, and and sets the current proof context of the system to the merge of the proof contexts with the given keys. *set_merge_pcs* discards the previous "current" proof context, and and sets the current proof context of the system to the merge of the proof contexts with the given keys. Merged proof contexts upon the stack will have *current_ad_names* giving the list of names of the constituent proof contexts, singleton contexts will have singleton lists. The proof contexts used need not have been committed.

The *pr_conv*, *pr_tac* and *mmp_rule* fields take the value of the last proof context in the list that has the field set.

The current proof context is accessed by the functions prefixed *current_ad_*, and by *get_current-_pc*.

See Also *pop_pc*, *push_pc*, *set_pc*

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51020 Must be at least one key in list
```

SML
```
val push_pc : string -> unit;
val set_pc : string -> unit;
```

Description *push_pc* stacks the previous "current" proof context, and and sets the current proof context of the system to the proof context with the given key. *set_pc* discards the previous "current" proof context, and and sets the current proof context of the system to the proof context with the given key.

The current proof context is accessed by the functions prefixed *current_ad_*, and by *get_current-_pc*.

See Also *pending_push_pc*, *pop_pc*, *push_merge_pcs*, *set_merge_pcs*

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
```

SML
```
val set_cs_∃_convs : (CONV list) ->
        string -> unit;
val get_cs_∃_convs : string ->
        (((CONV list) * string) list);
```

Description These functions provide the interface to the existence provers for constant specifications (see *const_spec*) held in the user data of a proof context. Under the initial evaluator, the existence proving conversion supplied by *current_cs_∃_conv* will have each of the conversions tried, in the reverse order of their entry, being applied to the RHS of the result of the previous successful application, or the initial term to which the conversion was applied, until the RHS is ⌜T⌝, or no conversions remain.

Example
```
If get_cs_∃_convs of the current proof context returns

[([conv1, conv2],"pc1"),([conv3, conv4],"pc2")]

Then current_ad_cs_∃_conv  will return

conv4 AND_OR_C conv3 AND_OR_C conv2 AND_OR_C conv1
```

"setting" overwrites the subfield whose key is the proof context's name, "getting" returns the entire field (which pairs data with proof context names).

Errors
```
51010  There is no proof context with key ?0
51014  Proof context ?0 was created in theory ?1 at a
       point now either not in scope, deleted or modified
51016  Proof context ?0 has been committed
```

SML
```
val set_mmp_rule : (THM -> THM -> THM) -> string -> unit;
val get_mmp_rule : string -> (THM -> THM -> THM) OPT;
```

Description These functions provide the interface to the proof contexts for the matching modus ponens rule as used by tools such as *forward_chain_rule*. Note that setting overwrites all previous data in this field, including from merged in proof contexts. Merged proof contexts take their value for this field from the last proof context in the list that has the field set.

Errors
```
51010  There is no proof context with key ?0
51014  Proof context ?0 was created in theory ?1 at a
       point now either not in scope, deleted or modified
51016  Proof context ?0 has been committed
```

SML

```
val set_nd_entry : string -> (TERM * (TERM -> THM))list -> string -> unit;
val get_nd_entry : string -> string ->
        ((TERM * (TERM -> THM))list * string) list;
```

Description These functions provide the interface to the additional dictionary of sources for discrimination nets held in the user data of a proof context. The dictionary is actually a list of subfields of the proof context, indexed by source proof context name, each subfield being a dictionary in its own right. You "set" a single dictionary entry of the subfield indexed by the proof context's name (creating a new entry if necessary). You "get" the dictionaries for all the subfields.

set_nd_entry dict_key entry pc_name overwrites (or creates, if necessary) the proof context's name's subfield dictionary entry whose key is *dict_key* in the proof context *pc_name* with the value *entry*.

get_nd_entry dict_key pc_name returns the dictionary entries whose keys are *dict_key* from each of the subfields in the proof context *pc_name*, paired with the source proof context name, or an empty list if the entry is not present in the dictionaries of any of the subfields of that proof context.

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed

SML

```
val set_pr_conv : (THM list -> CONV) -> string -> unit;
val get_pr_conv : string -> (THM list -> CONV);
val get_pr_conv1 : string -> (THM list -> CONV) OPT;
```

Description These functions provide the interface to the proof contexts for *prove_conv*. Note that setting overwrites all previous data in this field, including from merged in proof contexts. If the field has not been set, *get_pr_conv* returns a function mapping any list of theorems to *fail_conv* and *get_pr_conv1* returns *Nil*. Merged proof contexts take their value for this field from the last proof context in the list that has the field set.

Note that when using these functions that the standard rewriting functions (obvious candidates for inclusion in automatic proof) access the current proof context at the point of being given their theorem list argument.

Errors

51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed

SML
```
val set_pr_tac : (THM list -> TACTIC) -> string -> unit;
val get_pr_tac : string -> (THM list -> TACTIC);
val get_pr_tac1 : string -> (THM list -> TACTIC) OPT;
```

Description These functions provide the interface to the proof contexts for *prove_tac*. Note that setting overwrites all previous data in this field, including from merged in proof contexts. If the field has not been set, *get_pr_tac* returns a function mapping any list of theorems to *fail_tac* and *get_pr_tac1* returns *Nil*. Merged proof contexts take their value for this field from the last proof context in the list that has this field set.

When *asm_prove_tac* is applied to its theorem list argument the system will evaluate this by applying the value set by *set_pr_tac* for the current proof context to that argument. The provided values for *set_pr_tac* can interpret their theorem list arguments as they wish (e.g. as a set of rewrite theorems, or as theorems to resolve against) - no interpretation is forced upon this argument.

Note that when using these functions that the standard rewriting functions (obvious candidates for inclusion in automatic proof) access the current proof context at the point of being given their theorem list argument.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
```

SML
```
val set_rw_canons : (THM -> THM list) list ->
        string -> unit;
val get_rw_canons : string -> ((THM -> THM list) list * string) list;
```

Description These functions provide the interface to the individual canonicalisation functions used to create the canonicalisation function applied to rewriting theorems (see e.g. *rewrite_tac*) held in the user data of a proof context.

"setting" overwrites the subfield whose key is the proof context's name, "getting" returns the entire field (which pairs data with proof context names).

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
```

SML
```
val set_rw_eqm_rule : (THM -> TERM * CONV) -> string -> unit;
val get_rw_eqm_rule : string -> (THM -> TERM * CONV) OPT;
```

Description These functions provide the interface to the proof contexts for the equation matcher as used by the rewriting tools. Note that setting overwrites all previous data in this field, including from merged in proof contexts. Merged proof contexts take their value for this field from the last proof context in the list that has the field set.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
```

SML
```
val set_rw_eqn_cxt : EQN_CXT -> string -> unit;
val get_rw_eqn_cxt : string -> (EQN_CXT * string) list;
val add_rw_thms : THM list -> string -> unit;
```

Description These functions provide the interface to the equational context for rewriting (see e.g. *rewrite_tac*) held in the user data of a proof context. "setting" overwrites the subfield whose key is the proof context's name, "getting" returns the entire field (which pairs data with proof context names). "adding" processes its theorems by first canonicalising according to the current proof context's canonicalisation function, and then with *thm_eqn_cxt* and then adds them into the subfield whose key is the proof context's name.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
```

SML
```
val set_sc_eqn_cxt : EQN_CXT -> string -> unit;
val get_sc_eqn_cxt : string -> (EQN_CXT * string) list;
val add_sc_thms : THM list -> string -> unit;
```

Description These functions provide the interface to the equational context for stripping goal conclusions (see e.g. *strip_tac*) held in the user data of a proof context. "setting" overwrites the subfield whose key is the proof context's name, "getting" returns the entire field (which pairs data with proof context names). "adding" processes its theorems by first canonicalising according to the current proof context's canonicalisation function, and then with *thm_eqn_cxt* and then adds them into the subfield whose key is the proof context's name.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
```

SML
```
val set_st_eqn_cxt : EQN_CXT -> string -> unit;
val get_st_eqn_cxt : string -> (EQN_CXT * string)list;
val add_st_thms : THM list -> string -> unit;
```

Description These functions provide the interface to the equational context for stripping theorems into the assumption list (see e.g. *strip_tac*) held in the user data of a proof context. "setting" overwrites the subfield whose key is the proof context's name, "getting" returns the entire field (which pairs data with proof context names). "adding" processes its theorems by first canonicalising according to the current proof context's canonicalisation function, and then with *thm_eqn_cxt* and then adds them into the subfield whose key is the proof context's name.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
```

SML
```
val set_∃_cd_thms : THM list -> string -> unit;
val get_∃_cd_thms : string -> THM list;
val add_∃_cd_thms : THM list -> string -> unit;
```

Description These functions provide the interface to the unevaluated clausal definition theorems held for the existence prover *prove_∃_conv*. There are no subfields to this field, so "setting" overwrites the field with the proof context's name, "getting" returns the field. "adding" unions its theorem list with the proof contexts field.

See Also See *evaluate_∃_cd_thms* for details upon the form of the theorems.

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
```

SML

```
val set_∃_vs_thms : (string * (TERM list * THM)) list
        -> string -> unit;
val get_∃_vs_thms : string -> (string * (TERM list * THM)) list;
```

Description These functions provide the interface to the variable structure information for the existence prover *prove_∃_conv*. An individual entry in the list gives a method of handling an extended variable structure. It consists of the name of the constructor; a list of functions that extract each field of the constructor, and a theorem that states how the extraction functions extract from a data construction, and that the data constructor may be applied to the extracted values to regain the original value. For instance, for pairs the information is:

```
        (",",
        ([⌜Fst⌝,⌜Snd⌝],
        ‘⊢ ∀ x y p •
                Fst (x, y) = x ∧ Snd (x, y) = y ∧
                (Fst p, Snd p) = p)‘))
```

There are no subfields to this field, so "setting" overwrites the field with the proof context's name, "getting" returns the field.

Errors

```
51010  There is no proof context with key ?0
51014  Proof context ?0 was created in theory ?1 at a
       point now either not in scope, deleted or modified
51016  Proof context ?0 has been committed
```

SML

```
val simple_ho_thm_eqn_cxt : THM -> (TERM * CONV);
```

Description This function is an equation matcher for use by the rewriting tools that uses higher-order matching. It transforms an equational theorem into a representation of a higher-order rewrite rule in a form suitable for inclusion in an an equational context (*EQN_CXT* q.v.)

```
thm_eqn_cxt ‘Γ ⊢ ∀ x1 ... • LHS = RHS‘ →
        (LHS′, simple_eq_match_conv1 ‘Γ ⊢ ∀ x1 ... • LHS = RHS‘)
```

Here the pattern term LHS' is derived from *LHS* by replacing linear patterns (see *simple_ho_match*) by variables of the same type.

The universal quantifiers must be over simple variables (not patterns) and the higher-order matching is done using *simple_ho_match*.

See Also *cthm_eqn_cxt* which canonicalises the theorem before transformation.

Errors

```
7095   ?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u = v′ where ⌜xi⌝ are variables
```

SML

val **thm_eqn_cxt** : *THM* −> (*TERM* ∗ *CONV*);

Description This function is a simple form of equation matcher for use by the rewriting tools. It transforms an equational theorem into a representation of a first-order rewrite rule in a form suitable for inclusion in an an equational context (*EQN_CXT* q.v.)

thm_eqn_cxt ‘Γ ⊢ ∀ x1 ... • LHS = RHS‘ →
        (*LHS*, *simple_eq_match_conv1* ‘Γ ⊢ ∀ x1 ... • LHS = RHS‘)

The universal quantifiers must be over simple variables (not patterns).

See Also *cthm_eqn_cxt* which canonicalises the theorem before transformation.

Errors

7095 ?0 is not of the form ‘Γ ⊢ ∀ x1 ... xn • u = v′ where ⌜xi⌝ are variables

SML

signature **ProofContexts1** = *sig*

Description This signature gives access to two functions used in supplying the first group of proof contexts. Proof contexts themselves have no entry in the signature, however the contexts provided are:

Component	Complete
′simple_abstractions	*predicates*
′paired_abstractions	*predicates1*
′propositions	*basic_hol*
′fun_ext	*basic_hol1*
′pair	*sets_ext*
′pair1	*hol*
′ℕ	*hol1*
′ℕ_lit	
′list	
′char	
′sum	
′one	
′combin	
′sets_alg	
′sets_ext	
′basic_prove_∃_conv	

SML

(∗ *Proof Context*: ′**basic_prove_∃_conv** ∗)

Description A component proof context that adds the function *basic_prove_∃_conv* as an automatic existence prover.

Contents Automatic proof procedures are respectively “always fail tactic”, “always fail conversion”, and *basic_prove_∃_conv*.

Usage Notes Requires theory “basic_hol”, intended to be combined into the merge of any component proof contexts that do not have their own special existence prover. It should usually be the first in the list of proof contexts to be merged together, so that other proof contexts may introduce pre-processors, and then the final default prover is invoked. This is because the standard application of the list of existence prover conversions is defined to be to apply them in a cumulative manner, in reverse order.

SML

|(∗ *Proof Context*: ′**simple_abstractions** ∗)

Description A component proof context for handling only simple abstractions in stripping and canonicalisation.

Contents Rewriting:

Stripping theorems:

|*simple_¬_in_conv*

Stripping conclusions:

|*simple_¬_in_conv*

Rewriting canonicalisation:

|*simple_∀_rewrite_canon*, *simple_¬_rewrite_canon*

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Not to be used with proof context "′paired_abstractions" as their "domains" overlap. It requires theory *basic_hol*.

SML

|(∗ *Proof Context*: ′**paired_abstractions** ∗)

Description A component proof context for handling simple and paired abstractions in stripping and canonicalisation.

Contents Rewriting:

|*β_conv*

Stripping theorems:

|*¬_in_conv*, *∃₁_conv*,
|*∀_uncurry_conv*, *∃_uncurry_conv*

Stripping conclusions:

|*¬_in_conv*, *∀_uncurry_conv*

Rewriting canonicalisation:

|*∀_rewrite_canon*, *¬_rewrite_canon*

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Not to be used with proof context "′simple_abstractions", as their "domains" overlap. It requires theory *basic_hol*.

SML

(∗ *Proof Context*: ′**propositions** ∗)

Description A component proof context for reasoning about propositions.

Contents Rewriting:

$eq\_rewrite\_thm$, $\Leftrightarrow\_rewrite\_thm$, $\neg\_rewrite\_thm$,
$\wedge\_rewrite\_thm$, $\vee\_rewrite\_thm$, $\Rightarrow\_rewrite\_thm$,
$if\_rewrite\_thm$, $\forall\_rewrite\_thm$, $\exists\_rewrite\_thm$,
$\beta\_rewrite\_thm$, $simple\_\beta\_conv$

Stripping theorems:

$\Rightarrow\_thm$, $\Leftrightarrow\_thm$, $simple\_\exists_1\_conv$,
‘⊢ ∀ x • ((x = x) ⇔ T)‘,
‘⊢ ∀ x • (¬(x = x) ⇔ F)‘,
‘⊢ ∀ a $t1$ $t2$• (if a $then$ $t1$ $else$ $t2$) ⇔ (a ⇒ $t1$) ∧ (¬ a ⇒ $t2$)‘

Note these are intended to be used with ($simple\_$) $\neg\_in\_conv$ from “′paired_abstractions” or “′simple_abstractions”, which covers the cases of an outermost ¬ for each operator.

Stripping conclusions:

$\Leftrightarrow\_thm$,
‘⊢ ∀ x • ((x = x) ⇔ T)‘,
‘⊢ ∀ x • (¬(x = x) ⇔ F)‘,
‘⊢ ∀ a $t1$ $t2$• (if a $then$ $t1$ $else$ $t2$) ⇔ (a ⇒ $t1$) ∧ (¬ a ⇒ $t2$)‘
‘⊢ ∀a b•(a ∨ ¬b) ⇔ (b ⇒ a)‘
‘⊢ ∀ a b• ¬ a ∨ b ⇔ a ⇒ b‘
‘⊢ ∀ a b• a ∨ b ⇔ ¬ a ⇒ b‘

Note that the above are intended to be used in combination with ($simple\_$) $\neg\_in\_conv$ from “′paired_abstractions” or “′simple_abstractions”, which covers the cases of an outermost ¬ for each operator.

Rewriting canonicalisation:

$\wedge\_rewrite\_canon$, $f\_rewrite\_canon$

Automatic proof procedures are respectively $taut\_tac$, $taut\_conv$ and $basic\_prove\_\exists\_conv$.

Usage Notes Usually used in conjunction with “′paired_abstractions” or “′simple_abstractions”, requires theory $basic\_hol$.

SML
|(∗ *Proof Context*: **′fun_ext** ∗)

Description A component proof context for adding reasoning using functional extensionality.

Contents Rewriting:

|*ext_thm*

Stripping theorems:

|*ext_thm*

Stripping conclusions:

|*ext_thm*

Rewriting canonicalisation:

Automatic proof procedures are, respectively, *taut_tac*, *taut_conv* and *basic_prove_∃_conv*.

Usage Notes Normally used in conjunction with "′propositions", requires theory *basic_hol*.

SML
|(∗ *Proof Context*: **predicates** ∗)

Description A "mild" complete proof context for reasoning about the predicate calculus, including paired abstractions.

Contents Proof contexts "′basic_prove_∃_conv", "′paired_abstractions" and "′propositions".

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and *basic_prove_∃_conv* (merged in from the proof context of the same name).

Usage Notes Requires theory *basic_hol*.

SML
|(∗ *Proof Context*: **predicates1** ∗)

Description An "aggressive" complete proof context for reasoning about the predicate calculus, including paired abstractions and functional extensionality.

Contents Proof contexts "′basic_prove_∃_conv", "′paired_abstractions", "′propositions" and "′fun_ext".

Automatic proof procedures are, respectively, *basic_prove_tac*, *basic_prove_conv* and *basic_prove_∃_conv* (merged in from the proof context of the same name).

Usage Notes Requires theory *basic_hol*.

SML

(∗ *Proof Context*: ′**pair** ∗)

Description A "mild" component proof context for theory *pair*.

Contents Rewriting (selected from *pair_clauses*):

$$`\vdash \forall\ x\ y\ a\ b\ p\ fu\ fc \bullet Fst\ (x,\ y) = x \land Snd\ (x,\ y) = y \land ((a,\ b) = (x,\ y) \Leftrightarrow a = x \land b = y) \land (Fst\ p,\ Snd\ p) = p \land Curry\ fc\ x\ y = fc\ (x,\ y) \land Uncurry\ fu\ (x,\ y) = fu\ x\ y \land Uncurry\ fu\ p = fu\ (Fst\ p)\ (Snd\ p)`$$

Stripping theorems:

$$`\vdash \forall\ a\ b\ x\ y \bullet ((a,\ b) = (x,\ y) \Leftrightarrow a = x \land b = y)`$$

Stripping conclusions:

$$`\vdash \forall\ a\ b\ x\ y \bullet ((a,\ b) = (x,\ y) \Leftrightarrow a = x \land b = y)`$$

Existential variable structures:

$$`\vdash \forall\ x\ y\ p \bullet Fst\ (x,\ y) = x \land Snd\ (x,\ y) = y \land (Fst\ p,\ Snd\ p) = p`$$

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Requires theory *basic_hol*.

SML

(∗ *Proof Context*: ′**pair1** ∗)

Description An "aggressive" component proof context for theory *pair*.

Contents Rewriting:

$$⊢ \forall a\ b\ p \bullet ((a, b) = p \Leftrightarrow a = Fst\ p \wedge b = Snd\ p) \wedge (p = (a, b) \Leftrightarrow Fst\ p = a \wedge Snd\ p = b)$$

Stripping theorems (selected from *pair_clauses*):

$$⊢ \forall a\ b\ p \bullet ((a, b) = p \Leftrightarrow a = Fst\ p \wedge b = Snd\ p) \wedge (p = (a, b) \Leftrightarrow Fst\ p = a \wedge Snd\ p = b)$$

Stripping conclusions:

$$⊢ \forall a\ b\ p \bullet ((a, b) = p \Leftrightarrow a = Fst\ p \wedge b = Snd\ p) \wedge (p = (a, b) \Leftrightarrow Fst\ p = a \wedge Snd\ p = b)$$

Existential variable structures:

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Requires theory *basic_hol*, expected to be used in combination with "′pair".

SML

(∗ *Proof Context*: ′ℕ ∗)

Description A "mild" component proof context for theory ℕ.

Contents Rewriting:

≥_def, *greater_def*, *plus_clauses*, *times_clauses*,
≤_clauses, *less_clauses*, *minus_clauses*

Stripping theorems:

≥_def, *greater_def*, *plus_clauses*, *times_clauses*,
≤_clauses, *less_clauses*, *minus_clauses*,
and all boolean equations also pushed through $¬$

Stripping conclusions:

≥_def, *greater_def*, *plus_clauses*, *times_clauses*,
≤_clauses, *less_clauses*, *minus_clauses*,
and all boolean equations also pushed through $¬$

Existential clausal definition theorems:

prim_rec_thm

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Requires theory *basic_hol*.

SML

(* *Proof Context*: ′ℕ_**lit** *)

Description A component proof context for theory ℕ, that will, e.g., evaluate any arithmetic expression involving only numeric literals and certain arithmetic operators, namely +, ∗, −, *Div*, *Mod*, ≤, <, >, ≥, *and* =.

Contents Rewriting:

plus_conv, *times_conv*, *minus_conv*, *div_conv*,
mod_conv, ≤_*conv*, *less_conv*, *greater_conv*,
≥_*conv*, ℕ_*eq_conv*

Stripping theorems:

≤_*conv*, *less_conv*, *greater_conv*,
≥_*conv*, ℕ_*eq_conv*

Stripping conclusions:

≤_*conv*, *less_conv*, *greater_conv*,
≥_*conv*, ℕ_*eq_conv*

Existential clausal definition theorems:

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Requires theory *basic_hol*, expected to be used with proof context "′ℕ". It is separated from it as spotting the application of the conversions is time consuming, and may be known to be irrelevant.

SML

```
(* Proof Context: 'list *)
```

Description A component proof context for the theory *list*.

Contents Rewriting:

```
list_clauses
```

Stripping theorems:

```
‘ ⊢ ∀ x1 x2 list1 list2
    • ¬ Cons x1 list1 = []
      ∧ ¬ [] = Cons x1 list1
      ∧ (Cons x1 list1 = Cons x2 list2 ⇔ x1 = x2 ∧ list1 = list2)‘
```

Stripping conclusions:

```
‘ ⊢ ∀ x1 x2 list1 list2
    • ¬ Cons x1 list1 = []
      ∧ ¬ [] = Cons x1 list1
      ∧ (Cons x1 list1 = Cons x2 list2 ⇔ x1 = x2 ∧ list1 = list2)‘
```

Existential clausal definition theorems:

```
list_prim_rec_thm
```

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Requires theory *list*.

SML

```
(* Proof Context: 'char *)
```

Description A component proof context for theory *char*, for reasoning about character and string literals.

Contents Rewriting:

```
char_eq_conv, string_eq_conv
```

Stripping theorems:

```
char_eq_conv, string_eq_conv
```

Stripping conclusions:

```
char_eq_conv, string_eq_conv
```

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and an existence prover preprocessor that rewrites with ⊢ "" = [] which assists using list's primitive induction on strings.

Usage Notes Requires theory *basic_hol*.

SML
|(∗ *Proof Context*: **basic_hol** ∗)

Description A "mild" complete proof context for the ancestors of theory *basic_hol*.

Contents Proof contexts "predicates", "′pair, "′ℕ", "′list", and "′char". Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and *basic_prove_∃_conv* (merged in from the proof context of the same name).

Usage Notes Requires theory *basic_hol*.

SML
|(∗ *Proof Context*: **basic_hol1** ∗)

Description An "aggressive" complete proof context for the ancestors of theory *basic_hol*.

Contents Proof contexts "predicates1", "′pair", "′pair1", "′ℕ", "′ℕ_lit", "′list", and "′char".

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and *basic_prove-_∃_conv* (merged in from the proof context of the same name).

Usage Notes Requires theory *basic_hol*.

SML
|(∗ *Proof Context*: **′mmp1** ∗)

Description A component proof context with the matching modus ponens rule set to *⇒_match_mp_rule1*. All other fields are empty.

Usage Notes This makes forward chaining work as in releases prior to 2.9.1 (so that bound variables that are not constrained by the pattern matching are specialised to themselves).

SML
|(∗ *Proof Context*: **′mmp2** ∗)

Description A component proof context with the matching modus ponens rule set to *⇒_match_mp_rule2*. All other fields are empty.

Usage Notes Use this to ensure the default behaviour in forward chaining (so that bound variables that are not constrained by the pattern matching are specialised with new names as necessary to avoid variable capture).

SML
|(∗ *Proof Context*: **′sho_rw** ∗)

Description A component proof context with the equation matching rule set to *simple_higher_order_thm_eqn_cxt*. All other fields are empty.

Usage Notes With this proof context, rewriting treats the rewriting theorems as higher order rewrite rules. For example, rewriting with the theorem *prenex_clauses* (q.v.) will convert a term into prenex normal form.

SML

(∗ *Proof Context*: ′**sum** ∗)

Description A "mild" component proof context for theory *sum*.

Contents Rewriting:

$$
\begin{array}{l}
` \vdash \forall\ x1\ x2\ y1\ y2\ z \\
\quad \bullet (InL\ x1 = InL\ x2 \Leftrightarrow x1 = x2) \\
\quad \land (InR\ y1 = InR\ y2 \Leftrightarrow y1 = y2) \\
\quad \land \neg\ InL\ x1 = InR\ y1 \\
\quad \land \neg\ InR\ y1 = InL\ x1 \\
\quad \land OutL\ (InL\ x1) = x1 \\
\quad \land OutR\ (InR\ y1) = y1 ` \\
\quad \land IsL(InL\ x1) \land IsR(InR\ y1) \\
\quad \land \neg\ IsL(InR\ y1) \land \neg\ IsR(InL\ x1)
\end{array}
$$

Stripping theorems:

$$
\begin{array}{l}
` \vdash \forall\ x1\ x2\ y1\ y2\ z \\
\quad \bullet (InL\ x1 = InL\ x2 \Leftrightarrow x1 = x2) \\
\quad \land (InR\ y1 = InR\ y2 \Leftrightarrow y1 = y2) \\
\quad \land \neg\ InL\ x1 = InR\ y1 \\
\quad \land \neg\ InR\ y1 = InL\ x1 ` \\
\quad \land IsL(InL\ x1) \land IsR(InR\ y1) \\
\quad \land \neg\ IsL(InR\ y1) \land \neg\ IsR(InL\ x1)
\end{array}
$$

Stripping conclusions:

$$
\begin{array}{l}
` \vdash \forall\ x1\ x2\ y1\ y2\ z \\
\quad \bullet (InL\ x1 = InL\ x2 \Leftrightarrow x1 = x2) \\
\quad \land (InR\ y1 = InR\ y2 \Leftrightarrow y1 = y2) \\
\quad \land \neg\ InL\ x1 = InR\ y1 \\
\quad \land \neg\ InR\ y1 = InL\ x1 ` \\
\quad \land IsL(InL\ x1) \land IsR(InR\ y1) \\
\quad \land \neg\ IsL(InR\ y1) \land \neg\ IsR(InL\ x1)
\end{array}
$$

Existential clausal definition theorems:

$$` \vdash \forall\ f\ g \bullet \exists_1\ h \bullet (\forall\ x \bullet\ h\ (InL\ x) = f\ x) \land (\forall\ x \bullet\ h\ (InR\ x) = g\ x) `$$

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Requires theory *sum*.

SML

```
(∗ Proof Context: 'one ∗)
```

Description A component proof context for theory *one*

Contents Rewriting (these both have the problem that their discrimination net entry will match anything):

one_def, *one_fns_thm*

Stripping theorems:

$\text{‘} \vdash \forall\ x\ y : ONE \bullet (x = y) \Leftrightarrow T\text{‘}$
$\text{‘} \vdash \forall\ x\ y : {'a} \rightarrow ONE \bullet (x = y) \Leftrightarrow T\text{‘}$
and through ¬

Stripping conclusions:

$\text{‘} \vdash \forall\ x\ y : ONE \bullet (x = y) \Leftrightarrow T\text{‘}$
$\text{‘} \vdash \forall\ x\ y : {'a} \rightarrow ONE \bullet (x = y) \Leftrightarrow T\text{‘}$
and through ¬

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Requires theory *one*. As when entered into the rewriting net the rewriting theorems will match any term presented to the net, this proof context will slow down rewriting.

SML

```
(∗ Proof Context: 'combin ∗)
```

Description A component proof context for theory *combin*

Contents Rewriting:

comb_i_def, *comb_k_def*, *o_def*, *o_i_thm*

Stripping theorems:

Stripping conclusions:

Automatic proof procedures are, respectively, *basic_prove_tac*, *basic_prove_conv* and no existence prover.

Usage Notes Requires theory *combin*.

SML

(* *Proof Context*: ′**sets_alg** *)

Description A "mild" component proof context for theory *set*.

Contents Rewriting:

$\in$_*comp_conv*, $\in$_*enum_set_conv*, *complement_clauses*,
$\cup$_*clauses*, $\cap$_*clauses*, *set_dif_clauses*, $\ominus$_*clauses*,
$\subseteq$_*clauses*, $\subset$_*clauses*, $\bigcup$_*clauses*,
$\bigcap$_*clauses*, $\mathbb{P}$_*clauses*
‘ ⊢ ∀ x y
• ¬ x ∈ {}
∧ x ∈ *Universe*
∧ (x ∈ {y} ⇔ x = y)‘

Stripping theorems:

$\in$_*comp_conv*, $\in$_*enum_set_conv*, $\in$_*in_clauses*
$\subseteq$_*clauses*, $\subset$_*clauses*
plus these all pushed in through ¬

Stripping conclusions:

$\in$_*comp_conv*, $\in$_*enum_set_conv*, $\in$_*in_clauses*
$\subseteq$_*clauses*, $\subset$_*clauses*
plus these all pushed in through ¬

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and the existence prover preprocessor:

TOP_MAP_C (*all_∃_uncurry_conv AND_OR_C sets_simple_∃_conv*)

. The preprocessor causes set membership (∈) to be treated as function application in some cases.

Usage Notes Should not be used with proof context "′sets_ext", requires theory *sets*.

SML

(∗ *Proof Context*: **′sets_ext** ∗)

Description A component proof context for theory *set*, "aggressively" using the extensionality of sets.

Contents Rewriting:

∈_comp_conv, *∈_enum_set_conv*, *∈_in_clauses*, *sets_ext_clauses*

Stripping theorems:

∈_comp_conv, *∈_enum_set_conv*, *∈_in_clauses*, *sets_ext_clauses*
plus these all pushed in through ¬

Stripping conclusions:

∈_comp_conv, *∈_enum_set_conv*, *∈_in_clauses*, *sets_ext_clauses*
plus these all pushed in through ¬

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and the existence prover preprocessor:

TOP_MAP_C (*all_∃_uncurry_conv AND_OR_C sets_simple_∃_conv*)

The preprocessor causes set membership (∈) to be treated as function application in some cases.

Usage Notes Should not be used with proof context "′sets_alg", requires theory *sets*.

SML

(∗ *Proof Context*: **sets_ext** ∗)

Description A complete proof context for reasoning about sets within the predicate calculus, "aggressively" using the extensionality of sets.

Contents Proof contexts "′sets_ext" and "predicates".

Usage Notes Requires theory *sets*. The proof context "′sets_ext1" offers a much more useful treatment of sets of pairs.

SML
```
(* Proof Context: 'sets_ext1 *)
```

Description A component proof context for theory *set*, including sets of pairs, "aggressively" using the extensionality of sets.

Contents Rewriting:

| *∈_comp_conv*, *∈_enum_set_conv*, *∈_in_clauses*,
| *sets_eq_conv*, *⊆_conv*, *⊂_conv*

Stripping theorems:

| *∈_comp_conv*, *∈_enum_set_conv*, *∈_in_clauses*,
| *sets_eq_conv*, *⊆_conv*, *⊂_conv*
| *plus these all pushed in through ¬*

Stripping conclusions:

| *∈_comp_conv*, *∈_enum_set_conv*, *∈_in_clauses*,
| *sets_eq_conv*, *⊆_conv*, *⊂_conv*
| *plus these all pushed in through ¬*

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and the existence prover preprocessor:

| *TOP_MAP_C (all_∃_uncurry_conv AND_OR_C sets_simple_∃_conv)*

The preprocessor causes set membership (∈) to be treated as function application in some cases.

Usage Notes Should not be used with proof context "'sets_alg", requires theory *sets*.

SML
```
(* Proof Context: sets_ext *)
```

Description A complete proof context for reasoning about sets within the predicate calculus, "aggressively" using the extensionality of sets.

Contents Proof contexts "'sets_ext" and "predicates".

Usage Notes Requires theory *sets*.

SML
```
(* Proof Context: sets_ext1 *)
```

Description A complete proof context for reasoning about sets, including sets of pairs, within the predicate calculus, "aggressively" using the extensionality of sets.

Contents Proof contexts "'sets_ext1" and "predicates".

Usage Notes Requires theory *sets*. The proof context "sets_ext1" offers a much more useful treatment of sets of pairs.

SML
```
(* Proof Context: hol *)
```

Description A "mild" complete proof context for the ancestors of theory *hol*

Contents Proof contexts "basic_hol", "'sum", "'combin", and "'sets_alg".

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and *basic_prove-_∃_conv* (merged in from the proof context of the same name).

Usage Notes Requires theory *hol*.

SML
```
(∗ Proof Context: hol1 ∗)
```

Description An "aggressive" complete proof context for the ancestors of theory *hol*.

Contents Proof contexts "basic_hol1", "'one", "'sum", "'combin", and "'sets_ext".

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and *basic_prove_∃_conv* (merged in from the proof context of the same name).

Usage Notes Requires theory *hol*. The proof context *hol2* offers a more useful treatment of sets of pairs.

SML
```
(∗ Proof Context: hol2 ∗)
```

Description An "aggressive" complete proof context for the ancestors of theory *hol*.

Contents Proof contexts "basic_hol1", "'one", "'sum", "'combin", and "'sets_ext1".

Automatic proof procedures are respectively *basic_prove_tac*, *basic_prove_conv* and *basic_prove_∃_conv* (merged in from the proof context of the same name).

Usage Notes Requires theory *hol*.

SML
```
val basic_prove_conv : THM list -> CONV;
```

Description This is the conversion used for the automatic proof conversion (*pr_tac* field) of most supplied proof contexts, and is a reasonable, general-purpose, automatic proof conversion. It will either prove the theorem with the given conclusion, or fail.

In summary it will:

1. Set the term as the goal of the subgoal package.
2. Attempt to rewrite the term with the current default rewrite rules and given theorems.
3. Repeatedly apply *strip_tac* to the goal.
4. Try *all_var_elim_asm_tac* to do variable elimination.
5. Attempt to prove the resulting goals with resolution for up to 3 resolution steps, with goal's negated conclusion as a resolvant that must be used, and the assumptions as possible other resolvants. This has no effect on any resulting goal if it is unsolved.
6. Attempt to prove the resulting goals with resolution for up to 3 resolution steps amongst just the assumptions. This has no effect on any resulting goal if it is unsolved.
7. If the proof is successful, return ⊢ *term* ⇔ *T* and otherwise fail.

Note that in the stripping step may result in more than one subgoal, and thus the plural "resulting goals".

Under the current interface to resolution this equivalent to:

```
fun basic_prove_conv thms tm =
⇔_t_intro (
tac_proof(([],tm),
        TRY_T (rewrite_tac thms)  THEN
        REPEAT strip_tac THEN_TRY
        (basic_res_tac2 3 [⊢ ∀ x • x = x]
         ORELSE_T basic_res_tac3 3 [⊢ ∀ x • x = x]))
)
```

In the implementation however, partial evaluation with just the theorems is allowed.

Errors
```
76001  Could not prove theorem with conclusion ?0
```

SML

```
val basic_prove_tac : THM list -> TACTIC;
```

Description This is the tactic used for the automated proof tactic (the *pr_tac* field) of most supplied proof contexts, and is a reasonable, general-purpose, automatic proof tactic.

In summary it will:

1. Try *all_var_elim_asm_tac* to do variable elimination.
2. Extract the assumption list, rewrite each extracted assumption with the current default rewrite rules and given theorems, and strip the results back into the assumption list.
3. Attempt to rewrite the resulting goal's conclusions with the current default rewrite rules and given theorems.
4. Again try *all_var_elim_asm_tac* to do variable elimination.
5. Repeatedly apply *strip_tac* to the conclusions of the resulting goals.
6. Attempt to prove each of the resulting goals with resolution for up to 3 resolution steps, with goal's negated conclusion as a resolvant that must be used, and the assumptions as possible other resolvants. This has no effect on any resulting goal if it is unsolved.
7. Attempt to prove each of the resulting goals with resolution for up to 3 resolution steps amongst just the assumptions. This has no effect on any resulting goal if it is unsolved.

Note that either stripping step may result in more than one subgoal, and thus the plural "resulting goals".

Under the current interface to resolution this is

```
fun basic_prove_tac thms =
TRY_T all_var_elim_asm_tac THEN
DROP_ASMS_T (MAP_EVERY (strip_asm_tac o rewrite_rule thms) o rev) THEN
(TRY_T (rewrite_tac thms)) THEN
TRY_T all_var_elim_asm_tac THEN
REPEAT strip_tac THEN_TRY
(basic_res_tac2 3 [⊢ ∀ x • x = x]
 ORELSE_T basic_res_tac3 3 [⊢ ∀ x • x = x])
```

SUPPORT FOR Z

8.1 Syntactic Manipulations

In the following descriptions of derived term constructors for Z it has been convenient to describe the effects of constructors using Z language quotations. In doing so quotations have sometimes been used which would not in fact be acceptable to the Z parser. The most frequent example of these is in quoting the declaration part of variable binding constructs in Z. The Z parser will not accept such declarations in isolation from the variable binding construct of which they form a part, but the most readable description of the effect of the constructor is obtained if we describe this as if the parser did accept such declarations in isolation.

In practice the best way of obtaining the term corresponding to the declaration part of such a construct is to parse a horizontal schema containing the required declaration part, and then take it apart using the appropriate destructor.

SML

```
signature ZTypesAndTerms = sig
```

Description The Z Abstract Machine functions are packaged into this signature.

SML

```
datatype BDZ
        =       BdzOk       of Z_TERM
        |       BdzNotZ     of int
        |       BdzFail     of {
                            BdzFCode  : int,
                            BdzFCompc : int,
                            BdzFArgc  : int
                            }
        ;
```

Description The return value from function *basic_dest_z_term*. The *BdzFail* constructor gives information primarily intended for use by the Z pretty printer.

See Also Function *basic_dest_z_term*.

SML

```
datatype  Z_TERM =
      ZDec of TERM list * TERM                  | ZSchemaDec of TERM * string
    | ZDecl of TERM list
    | ZEq of TERM * TERM                        | Z∈ of TERM * TERM
    | ZTrue                                     | ZFalse
    | Z¬ of TERM                                | Z¬s of TERM
    | Z∧ of TERM * TERM                         | Z∧s of TERM * TERM
    | Z∨ of TERM * TERM                         | Z∨s of TERM * TERM
    | Z⇒ of TERM * TERM                         | Z⇒s of TERM * TERM
    | Z⇔ of TERM * TERM                         | Z⇔s of TERM * TERM
    | Z∃ of TERM * TERM * TERM                  | Z∃s of TERM * TERM * TERM
    | Z∃1 of TERM * TERM * TERM                 | Z∃1s of TERM * TERM * TERM
    | Z∀ of TERM * TERM * TERM                  | Z∀s of TERM * TERM * TERM
    | ZSchemaPred of TERM * string
    | ZLVar of string * TYPE * TERM list        | ZGVar of string * TYPE * TERM list
    | ZInt of string                            | ZString of string
    | ZFloat of TERM * TERM * TERM              | Z⟨⟩ of TYPE * TERM list
    | ZSetd of TYPE * TERM list                 | ZSeta of TERM * TERM * TERM
    | Zℙ of TERM
    | ZTuple of TERM list
    | ZBinding of (string * TERM) list
    | Z× of TERM list
    | Zθ of TERM * string
    | ZSels of TERM * string
    | ZSelt of TERM * int                       | Zμ of TERM * TERM * TERM
    | ZApp of TERM * TERM                       | Zλ of TERM * TERM * TERM
    | ZLet of (string * TERM) list * TERM
    | ZHSchema of TERM * TERM
    | ZDecors of TERM * string                  | ZPres of TERM
    | Z↾s of TERM * TERM                        | ZHides of TERM * string list
    | ZΔs of TERM                               | ZΞs of TERM
    | Z⨾s of TERM * TERM
    | ZRenames of TERM * (string * string) list
    ;
```

Description This datatype corresponds to a version of the abstract syntax of Z in which recursion has been removed and the distinction between declarations, predicates and terms ignored. It is used by the generalised mapping functions *mk_z_TERM*, *is_z_term* and *dest_z_term* (q.v.).

SML

```
datatype  Z_TYPE =  ZGivenType of string
            |       ZVarType of string
            |       ZPowerType of TYPE
            |       ZTupleType of TYPE list
            |       ZSchemaType of (string * TYPE) list;
```

Description This datatype is a representation of the abstract syntax of Z types. It is used by the generalised mapping functions *mk_z_TYPE*, *is_z_type* and *dest_z_type* (q.v.). The operand of *ZGivenType* is the HOL name of the type.

SML
```
val basic_dest_z_term : TERM * TERM list -> BDZ;
```

Description Function *basic_dest_z_term* does the work of destroying a term to yield its Z structure. The arguments are in the result of applying *strip_app* to a term.

A call of '*basic_dest_z_term*(*strip_app zt*)' will attempt to destroy the Z term *zt*, if successful (i.e., *zt* is a valid Z term) then *BdzOk* is returned with the appropriate *Z_TERM* value. If *zt* is not a valid Z term then one of the other *BDZ* constructors is returned, these include an error code indicating what was wrong with the term. A *BdzFail* is returned when the term is similar to a Z term (i.e., it has a known constructor but the wrong number of arguments). In this case the *BdzFCompc* and *BdzFArgc* fields tell how many component lists and arguments (respectively) are allowed in a well formed Z term. A *BdzNotZ* is returned when the term is not recognisable as a Z term. In cases where insufficient component lists or arguments are given to a known constructor either *BdzFail* or *BdzNotZ* may be returned.

All of the error codes of function *dest_z_term* may be returned by this function.

See Also Functions: *dest_z_term* and *strip_app*; and, datatype *BDZ*.

SML
```
val dest_z_name1 : string -> string * string OPT;
val dest_z_name2 : string -> string OPT -> string list list * string OPT;
```

Description Supplying *dest_z_name2* with the result of *dest_z_name1* gives the same overall result as *dest_z_name* q.v. These functions allow the destruction of the component names and projection part to be deferred for efficiency, in case they are not required.

Errors
```
47000 ?0 is not a Z constant name
```

SML
```
val dest_z_name : string -> string * string list list * string OPT;
```

Description Analyses the names of Z semantic constants, returning the basic name and lists of embedded component names. If the name is a projection, then the projection part is also returned.

Errors
```
47000 ?0 is not a Z constant name
```

SML
```
val dest_z_term : TERM -> Z_TERM;
```

Description Converts a HOL term, which represents a valid Z term, to the appropriate *Z_TERM*.

See Also *dest_z_term1* which makes a more careful check, especially of schema constructs.

Errors
```
47900 ?0 is not a Z term
47901 ?0 is not a Z package
47910 ?0 is not a Z simple declaration
47911 ?0 is not a Z schema declaration
47912 ?0 is not a Z declaration
47920 ?0 is not a Z existential quantification
47921 ?0 is not a Z unique existential quantification
47922 ?0 is not a Z universal quantification
47923 ?0 is not a Z schema as a predicate
47110 ?0 is not a Z sequence display
47120 ?0 is not a Z set display
47130 ?0 is not a Z set comprehension
47170 ?0 is not a Z θ term
47190 ?0 is not a Z function application
47200 ?0 is not a Z λ abstraction
47936 ?0 is not a Z definite description
47937 ?0 is not a Z let expression
47940 ?0 is not a Z schema
47941 ?0 is not a Z schema existential quantification
47942 ?0 is not a Z schema unique existential quantification
47943 ?0 is not a Z schema universal quantification
```

SML
```
val dest_z_type : TYPE -> Z_TYPE;
```

Description Converts a HOL type, which represents a valid Z type, to the appropriate *Z_TYPE*.

Errors
```
47800 ?0 is not a Z type
```

SML
```
val gvar_subst : TERM -> (TERM * TERM) list;
```

Description Given an arbitrary term, *t*, *gvar_subst* creates a substitution mapping those free variables of *t* (in the HOL sense) which have the same names as Z global variables (i.e. HOL constants) in the current scope to the appropriate instances of those global variables (with generic instantiation using 𝕌 as necessary). The resulting substitution may then be used with *subst*, q.v., to “bind” the term into the current scope.

SML
```
val is_z_term : TERM -> bool;
```

Description Tests if a given HOL term is valid Z in its top level structure.

Uses Recursively in well-formedness checks.

See Also *is_z_term1* for a more complete check of top level structure, *is_z* for a full traversal of the terms structure.

SML
```
val is_z_type : TYPE -> bool;
```

Description Tests if a given HOL type represents a valid Z type.

Uses Recursively in well-formedness checks.

SML
```
val mk_dollar_quoted_string : string -> string;
val dest_dollar_quoted_string : string -> string;
val is_dollar_quoted_string : string -> bool;
```

Description The Z parser allows an arbitrary ML character string to be used to form an identifier. These functions implement the encoding used to embed an arbitrary ML string in the name of a Z variable:

Example
```
mk_dollar_quoted_string"<ext-name>" = "$\"<ext-name>\""
dest_dollar_quoted_string"$\"<ext-name>\"" = "<ext-name>"
is_dollar_quoted_string"$\"<ext-name>\"" = true
is_dollar_quoted_string"\"<ext-name>\"" = false
```

Errors
```
47001 ?0 is not a valid dollar-quoted string
```

SML
```
val mk_u : TYPE -> TERM;
val is_u : TERM -> bool;
val dest_u : TERM -> TYPE;
```

Description These functions create, test for, and destroy terms of the form $\mathbb{U}[Totality]$ which are used by the Z type inferrer to stand for elided generic actual parameters. The type parameter to *mk_u* and the result of *dest_u* is the type of the $\mathbb{U}$-term in question.

Errors
```
47950 ?0 is not of the form ⌜𝕌[Totality]⌝
47951 ?0 is not an instance of ⌜:'a SET⌝
```

SML
```
val mk_z_app : TERM * TERM -> TERM;
val is_z_app : TERM -> bool;
val dest_z_app : TERM -> TERM * TERM;
```

Description Z function application. The first argument must be a set of pairs, the second must have the same type as the first elements of the pairs.

Definition
```
mk_z_app(ⓩf⌝,ⓩa⌝) = ⓩ(f a)⌝
```

Errors
```
47190 ?0 is not a Z function application
47191 ?0 has the wrong type to be a Z function
47192 ?0 has the wrong type to be applied to ?1
```

SML

```
val mk_z_binding : (string * TERM) list -> TERM;
val is_z_binding : TERM -> bool;
val dest_z_binding : TERM -> (string * TERM) list;
```

Description The binding constructor.

Definition

mk_z_binding [("n_1",⌜Z t_1⌝),...,("n_n",⌜Z t_n⌝)] = ⌜Z (n_1≙t_1,...,n_n≙t_n)⌝

Errors

47151 ?0 is not a Z binding
47152 Cannot bind more than one value to ?0

SML

```
val mk_z_decl : TERM list -> TERM;
val is_z_decl : TERM -> bool;
val dest_z_decl : TERM -> TERM list;
```

Description Constructor, discriminator and destructor functions for the declaration part of a schema text. Its arguments must be made using *mk_z_dec* or *mk_z_schema_dec*.

Definition

mk_z_decl [⌜Z t1⌝,...,⌜Z tn⌝] = ⌜Z t1;...;tn⌝

Errors

47912 ?0 is not a Z declaration
3012 ?0 and ?1 do not have the same types

SML

```
val mk_z_decor_s : TERM * string -> TERM;
val is_z_decor_s : TERM -> bool;
val dest_z_decor_s : TERM -> TERM * string;
```

Description Constructor, discriminator and destructor functions for systematic decoration of schemas. The first argument must be a schema, the second a decoration.

Example

$mk\_z\_decor_s$(⌜Z [a,b,c:X | a = b]⌝,"′") = ⌜Z [a′,b′,c′:X | a′ = b′]⌝

Errors

47340 ?0 is not a Z decorated schema

SML

val **mk_z_dec** : *TERM list* ∗ *TERM* −> *TERM*;
val **is_z_dec** : *TERM* −> *bool*;
val **dest_z_dec** : *TERM* −> *TERM list* ∗ *TERM*;

Description Makes a simple declaration of one or more variables of the same type for use in the declaration part of a schema text.

Definition

$mk\_z\_dec([⌜_Z v_1⌝,...,⌜_Z v_n⌝],⌜_Z S⌝) = ⌜_Z v_1,...v_n : S⌝$

Where the v_i and the members of S must have the same type.

Uses May only be used to make arguments for *mk_z_decl*.

Errors

47060 ?0 is not a Z set
3012 ?0 and ?1 do not have the same types
3017 An empty list argument is not allowed
47061 ?0 is not a Z simple declaration

SML

val **mk_z_eq** : *TERM* ∗ *TERM* −> *TERM*;
val **is_z_eq** : *TERM* −> *bool*;
val **dest_z_eq** : *TERM* −> *TERM* ∗ *TERM*;

Description Equality. For the moment this is the same as HOL equality, but this is likely to change in the future. Both arguments must be of the same type.

Definition

$mk\_z\_eq(⌜_Z a⌝,⌜_Z b⌝) = ⌜_Z (a = b)⌝$

Errors

3012 ?0 and ?1 do not have the same types
47220 ?0 is not a Z equality

SML

val **mk_z_false** : *TERM*;
val **is_z_false** : *TERM* −> *bool*;

Description The Z constant *false*. It is the same as the HOL constant *F*.

SML

val **mk_z_float** : *TERM* ∗ *TERM* ∗ *TERM* −> *TERM*;
val **is_z_float** : *TERM* −> *bool*;
val **dest_z_float** : *TERM* −> *TERM* ∗ *TERM* ∗ *TERM*;

Description Constructor, discriminator and destructor functions for floating point literals. The argument is a triple of terms of type ℤ giving the mantissa, the number of digits after the decimal point and the exponent in that order, i.e., the triple (x, p, e) represents the real number $x \times 10^{e-p}$.

Errors

47107 ?0 is not a Z floating point literal
47108 ?0 does not have type ℤ

SML
```
val mk_z_given_type : string -> TYPE;
val is_z_given_type : TYPE -> bool;
val dest_z_given_type : TYPE -> string
```

Description These are the constructor, discriminator and destructor functions for the types of given sets. The type names used by these functions are the HOL names.

Errors
```
47010 ?0 is not a Z given type
```

SML
```
val mk_z_gvar : string * TYPE * TERM list -> TERM;
val is_z_gvar : TERM -> bool;
val dest_z_gvar : TERM -> string * TYPE * TERM list;
```

Description Constructor, discriminator and destructor functions for global variables. If the third argument is the empty list, this function is the same as the HOL *mk_const* function, otherwise a generic constant is created, the third argument being the generic actual parameters.

Errors
```
47100 ?0 is not a Z global variable
```

SML
```
val mk_z_hide_s : TERM * string list -> TERM;
val is_z_hide_s : TERM -> bool;
val dest_z_hide_s : TERM -> TERM * string list;
```

Description The schema hiding constructor. The first argument must be a schema, the second is a list of components to be hidden.

Definition
```
mk_z_hide_s(ⓩS⌝,["c1",..."cn"]) = ⓩS \ (c1,...,cn)⌝
```

Errors
```
47420 ?0 is not a Z schema hiding
```

SML
```
val mk_z_h_schema : TERM * TERM -> TERM;
val is_z_h_schema : TERM -> bool;
val dest_z_h_schema : TERM -> TERM * TERM;
```

Description The schema constructor. The first argument is a declaration constructed using *mk_z_decl*, the second is a predicate.

Definition
```
mk_z_h_schema(ⓩd⌝,ⓩp⌝) = ⓩ[d | p]⌝
```

Errors
```
47940 ?0 is not a Z schema
```

SML
```
val mk_z_int : string -> TERM;
val is_z_int : TERM -> bool;
val dest_z_int : TERM -> string;
```

Description Constructor, discriminator and destructor functions for integer literals. The argument should be a numeral, the result is the corresponding positive integer.

Errors
```
47105 ?0 is not a Z integer
```

SML

```
val mk_z_let : (string * TERM) list * TERM -> TERM;
val is_z_let : TERM -> bool;
val dest_z_let : TERM -> (string * TERM) list * TERM;
```

Description The let-term constructor. The arguments are list of pairs, each comprising a local variable name and a defining term for that local variable, and a term giving the body of the let-expression.

Definition

$mk\_z\_let([("v", ⌜_Z dt⌝), ...], ⌜_Z b⌝) = ⌜_Z let\ v \mathrel{\widehat=} dt; ... \bullet t⌝$

Errors

47211 ?0 is not a Z let term

SML

```
val mk_z_lvar : string * TYPE * TERM list -> TERM;
val is_z_lvar : TERM -> bool;
val dest_z_lvar : TERM -> string * TYPE * TERM list;
```

Description Constructor, discriminator and destructor functions for local variables. If the third argument is the empty list, this function is the same as the HOL *mk_var* function, otherwise a generic variable is created, the third argument being the generic actual parameters.

Errors

47090 ?0 is not a Z local variable

SML

```
val mk_z_power_type : TYPE -> TYPE;
val is_z_power_type : TYPE -> bool;
val dest_z_power_type : TYPE -> TYPE;
```

Description Set type constructor.

Definition

$mk\_z\_power\_type\ ty = \mathbb{P}\ ty$

Errors

47030 ?0 is not a Z set type

SML

```
val mk_z_pres : TERM -> TERM;
val is_z_pres : TERM -> bool;
val dest_z_pres : TERM -> TERM;
```

Description The schema precondition constructor. The argument must be a schema.

Definition

$mk\_z\_pre_s\ ⌜_Z S⌝ = ⌜_Z pre\ S⌝$

Errors

47350 ?0 is not a Z schema precondition

SML
```
val mk_z_rename_s : TERM * (string * string)list -> TERM;
val is_z_rename_s : TERM -> bool;
val dest_z_rename_s : TERM -> TERM * (string * string)list;
```

Description The schema renaming construct. Its argument must be a schema.

Definition
```
mk_z_rename_s (⌜Z S⌝,[("x1","y1"),...]) =
        ⌜Z S[x1/y1,...]⌝
```

Errors
```
47461 ?0 is not a Z schema renaming
47462 Cannot rename ?0 to more than one name
47463 Cannot rename more than one name to ?0
```

SML
```
val mk_z_schema_dec : TERM * string -> TERM;
val is_z_schema_dec : TERM -> bool;
val dest_z_schema_dec : TERM -> TERM * string;
```

Description Constructor, discriminator and destructor functions for the components of a schema (the first argument), systematically decorated with the second argument.

Uses May only be used to make arguments for *mk_z_decl*.

Errors
```
47940 ?0 is not a Z schema
47071 ?0 is not a Z schema as a declaration
```

SML
```
val mk_z_schema_pred : TERM * string -> TERM;
val is_z_schema_pred : TERM -> bool;
val dest_z_schema_pred : TERM -> TERM * string;
```

Description The schema as predicate constructor. The first argument must be a schema, the second is an optional decoration.

Errors
```
47940 ?0 is not a Z schema
47320 ?0 is not a Z schema as a predicate expression
```

SML
```
val mk_z_schema_type : (string * TYPE) list -> TYPE;
val is_z_schema_type : TYPE -> bool;
val dest_z_schema_type : TYPE -> (string * TYPE) list;
```

Description Binding type constructor.

Definition
```
mk_z_schema_type [(c1,ty1),...,(cn,tyn)] = [c1:ty1 ; ... ; cn:tyn]
```

Errors
```
47050 ?0 is not a Z binding type
```

SML

```
val mk_z_sel_s : TERM * string -> TERM;
val is_z_sel_s : TERM -> bool;
val dest_z_sel_s : TERM -> TERM * string;
```

Description Selection of a component from a binding. The type of the first argument must be a binding and the second argument must be a component of that type.

Definition

$mk\_z\_sel(⌜_Z S⌝,"c") = ⌜_Z S.c⌝$

Errors

47180 ?0 is not a Z selection

SML

```
val mk_z_sel_t : TERM * int -> TERM;
val is_z_sel_t : TERM -> bool;
val dest_z_sel_t : TERM -> TERM * int;
```

Description Selection of a component from a tuple. The type of the first argument must be a tuple and the second argument must be a component in that tuple.

Definition

$mk\_z\_sel_t(⌜_Z Tup⌝, i) = ⌜_Z Tup.i⌝$

Errors

47185 ?0 is not a Z tuple selection

SML

```
val mk_z_seta : TERM * TERM * TERM -> TERM;
val is_z_seta : TERM -> bool;
val dest_z_seta : TERM -> TERM * TERM * TERM;
```

Description Constructor, discriminator and destructor functions for set comprehension. The three arguments represent the declaration, predicate and body parts of the set comprehension and so must have the appropriate types. In particular, the first argument must be made using *mk_z_decl*.

Definition

$mk\_z\_seta(⌜_Z d⌝,⌜_Z p⌝,⌜_Z v⌝) = ⌜_Z \{d \mid p \bullet v\}⌝$

Errors

47130 ?0 is not a Z set comprehension

SML

```
val mk_z_setd : TYPE * TERM list -> TERM;
val is_z_setd : TERM -> bool;
val dest_z_setd : TERM -> TYPE * TERM list;
```

Description Constructor, discriminator and destructor functions for finite set displays. The result is the set made from the terms in the second argument, each of whose types must be the same as the first argument.

Definition

$mk\_z\_setd(ty,[⌜_Z t_1⌝,...,⌜_Z t_n⌝]) = ⌜_Z \{t_1,...,t_n\}⌝$

Where the t_i all have type ty.

Errors

47120 ?0 is not a Z set display

SML
```
val mk_z_string : string -> TERM;
val is_z_string : TERM -> bool;
val dest_z_string : TERM -> string;
```

Description Constructor, discriminator and destructor functions for string literals. The argument should be a string, the result is the corresponding string quotation.

Errors
```
47106 ?0 is not a Z string
```

SML
```
val mk_z_term : Z_TERM -> TERM;
```

Description Given any *Z_TERM*, *mk_z_TERM* calls the appropriate abstract machine *mk_* function.

SML
```
val mk_z_true : TERM;
val is_z_true : TERM -> bool;
```

Description The Z constant *true*. It is the same as the HOL constant *T*.

SML
```
val mk_z_tuple_type : TYPE list -> TYPE;
val is_z_tuple_type : TYPE -> bool;
val dest_z_tuple_type : TYPE -> TYPE list;
```

Description Cartesian product type constructor.

Definition
```
mk_z_tuple_type [ty1,...,tyn] = ty1 × ... × tyn
```

Errors
```
47040 ?0 is not a Z tuple type
```

SML
```
val mk_z_tuple : TERM list -> TERM;
val is_z_tuple : TERM -> bool;
val dest_z_tuple : TERM -> TERM list;
```

Description The tuple constructor.

Definition
```
mk_z_tuple [⌜Z t1⌝,...,⌜Z tn⌝] = ⌜Z (t1,...,tn)⌝
```

Errors
```
47150 ?0 is not a Z tuple
```

SML
```
val mk_z_type : Z_TYPE -> TYPE;
```

Description Given any *Z_TYPE*, *mk_z_type* calls the appropriate abstract machine *mk_* function.

SML
```
val mk_z_var_type : string -> TYPE;
val is_z_var_type : TYPE -> bool;
val dest_z_var_type : TYPE -> string;
```

Description The type of generic parameters.

Errors
```
47020 ?0 is not a Z type variable
```

SML

val **mk_z_$\mathbf{\Delta_s}$** : *TERM* −> *TERM*;
val **is_z_$\mathbf{\Delta_s}$** : *TERM* −> *bool*;
val **dest_z_$\mathbf{\Delta_s}$** : *TERM* −> *TERM*;

Description The delta constructor. Its argument must be a schema.

Definition

$mk\_z\_\Delta_s$ ⌜$_Z S$⌝ = ⌜$_Z \Delta S$⌝

Errors

47460 ?0 is not a Z Δ

SML

val **mk_z_∈** : *TERM* ∗ *TERM* −> *TERM*;
val **is_z_∈** : *TERM* −> *bool*;
val **dest_z_∈** : *TERM* −> *TERM* ∗ *TERM*;

Description Set membership. The second argument must be a set, whose members have the same type as the first argument.

Definition

$mk\_z\_\in$(⌜$_Z a$⌝,⌜$_Z b$⌝) = ⌜$_Z (a \in b)$⌝

Errors

47230 ?0 is not a Z set membership

SML

val **mk_z_$\mathbf{\Xi_s}$** : *TERM* −> *TERM*;
val **is_z_$\mathbf{\Xi_s}$** : *TERM* −> *bool*;
val **dest_z_$\mathbf{\Xi_s}$** : *TERM* −> *TERM*;

Description The xi constructor. Its argument must be a schema.

Definition

$mk\_z\_\Xi_s$ ⌜$_Z S$⌝ = ⌜$_Z \Xi S$⌝

Errors

47470 ?0 is not a Z Ξ

SML

val **mk_z_$\Leftrightarrow_\mathbf{s}$** : *TERM* ∗ *TERM* −> *TERM*;
val **is_z_$\Leftrightarrow_\mathbf{s}$** : *TERM* −> *bool*;
val **dest_z_$\Leftrightarrow_\mathbf{s}$** : *TERM* −> *TERM* ∗ *TERM*;

Description The schema equivalence constructor. Both arguments must be schemas.

Definition

$mk\_z\_\Leftrightarrow_s$(⌜$_Z R$⌝,⌜$_Z S$⌝) = ⌜$_Z R \Leftrightarrow S$⌝

Errors

47400 ?0 is not a Z schema if and only if

SML

```
val mk_z_⇔ : TERM * TERM -> TERM;
val is_z_⇔ : TERM -> bool;
val dest_z_⇔ : TERM -> TERM * TERM;
```

Description If and only if; the same as HOL ⇔. Its argument must be *bool* type.

Errors

```
3015  ?1 is not of type ⌜:BOOL⌝
3031  ?0 is not of type ⌜:BOOL⌝
47280 ?0 is not a Z if and only if
```

SML

```
val mk_z_⟨⟩ : TYPE * TERM list -> TERM;
val is_z_⟨⟩ : TERM -> bool;
val dest_z_⟨⟩ : TERM -> TYPE * TERM list;
```

Description Constructor, discriminator and destructor functions for finite sequences. The result is the sequence made from the terms in the second argument, each of whose types must be the same as the first argument.

Definition

```
mk_z_⟨⟩(ty,[ⓩt1⌝,...,ⓩtn⌝]) = ⓩ⟨t1,...,tn⟩⌝
```

Where the t_i all have type *ty*.

Errors

```
47110 ?0 is not a Z sequence display
```

SML

```
val mk_z_∧s : TERM * TERM -> TERM;
val is_z_∧s : TERM -> bool;
val dest_z_∧s : TERM -> TERM * TERM;
```

Description The schema conjunction constructor. Both arguments must be schemas.

Definition

```
mk_z_∧s(ⓩR⌝,ⓩS⌝) = ⓩR ∧ S⌝
```

Errors

```
47370 ?0 is not a Z schema conjunction
```

SML

```
val mk_z_∧ : TERM * TERM -> TERM;
val is_z_∧ : TERM -> bool;
val dest_z_∧ : TERM -> TERM * TERM;
```

Description Conjunction; the same as HOL ∧. Its arguments must be *bool* type.

Errors

```
3015  ?1 is not of type ⌜:BOOL⌝
3031  ?0 is not of type ⌜:BOOL⌝
47250 ?0 is not a Z conjunction
```

SML
```
val mk_z_∨s : TERM * TERM -> TERM;
val is_z_∨s : TERM -> bool;
val dest_z_∨s : TERM -> TERM * TERM;
```

Description The schema disjunction constructor. Both arguments must be schemas.

Definition
$mk\_z\_\lor_s(⌜_Z R⌝,⌜_Z S⌝) = ⌜_Z R \lor S⌝$

Errors
47380 ?0 is not a Z schema disjunction

SML
```
val mk_z_∨ : TERM * TERM -> TERM;
val is_z_∨ : TERM -> bool;
val dest_z_∨ : TERM -> TERM * TERM;
```

Description Disjunction; the same as HOL ∨. Its arguments must be *bool* type.

Errors
3015 ?1 is not of type ⌜:BOOL⌝
3031 ?0 is not of type ⌜:BOOL⌝
47260 ?0 is not a Z disjunction

SML
```
val mk_z_¬s : TERM -> TERM;
val is_z_¬s : TERM -> bool;
val dest_z_¬s : TERM -> TERM;
```

Description The schema negation constructor. The argument must be a schema.

Definition
$mk\_z\_\lnot_s\ ⌜_Z S⌝ = ⌜_Z \lnot S⌝$

Errors
47360 ?0 is not a Z schema negation

SML
```
val mk_z_¬ : TERM -> TERM;
val is_z_¬ : TERM -> bool;
val dest_z_¬ : TERM -> TERM;
```

Errors
3031 ?0 is not of type ⌜:BOOL⌝
47240 ?0 is not a Z negation

Description Negation; the same as HOL ¬. Its argument must be *bool* type.

SML
```
val mk_z_⇒s : TERM * TERM -> TERM;
val is_z_⇒s : TERM -> bool;
val dest_z_⇒s : TERM -> TERM * TERM;
```

Description The schema implication constructor. Both arguments must be schemas.

Definition
$mk\_z\_\Rightarrow_s(⌜_Z R⌝,⌜_Z S⌝) = ⌜_Z R \Rightarrow S⌝$

Errors
47390 ?0 is not a Z schema implication

SML
```
val mk_z_⇒ : TERM * TERM -> TERM;
val is_z_⇒ : TERM -> bool;
val dest_z_⇒ : TERM -> TERM * TERM;
```

Description Implication; the same as HOL ⇒. Its arguments must be *bool* type.

Errors
```
3015  ?1 is not of type ⌜:BOOL⌝
3031  ?0 is not of type ⌜:BOOL⌝
47270 ?0 is not a Z implication
```

SML
```
val mk_z_∀s : TERM * TERM * TERM -> TERM;
val is_z_∀s : TERM -> bool;
val dest_z_∀s : TERM -> TERM * TERM * TERM;
```

Description The schema universal quantifier constructor. The arguments must be a declaration (constructed using *mk_z_decl*), a predicate and a schema.

Definition
```
mk_z_∀s(ⓩd⌝,ⓩp⌝,ⓩS⌝) = ⓩ∀d | p • S⌝
```

Errors
```
47450 ?0 is not a Z schema universal
```

SML
```
val mk_z_∀ : TERM * TERM * TERM -> TERM;
val is_z_∀ : TERM -> bool;
val dest_z_∀ : TERM -> TERM * TERM * TERM;
```

Description Constructor, discriminator and destructor functions for universal quantification. Its arguments must be a declaration (constructed with *mk_z_decl*) and two predicates.

Definition
```
mk_z_∀(ⓩd⌝,ⓩp⌝,ⓩv⌝) = ⓩ∀d|p•v⌝
```

Errors
```
47912 ?0 is not a Z declaration
47310 ?0 is not a Z universal quantification
```

SML
```
val mk_z_∃1s : TERM * TERM * TERM -> TERM;
val is_z_∃1s : TERM -> bool;
val dest_z_∃1s : TERM -> TERM * TERM * TERM;
```

Description The schema unique existential quantifier constructor. The arguments must be a declaration (constructed using *mk_z_decl*), a predicate and a schema.

Definition
```
mk_z_∃1s(ⓩd⌝,ⓩp⌝,ⓩS⌝) = ⓩ∃1 d | p • S⌝
```

Errors
```
47440 ?0 is not a Z schema unique existential
```

SML
```
val mk_z_∃1 : TERM * TERM * TERM -> TERM;
val is_z_∃1 : TERM -> bool;
val dest_z_∃1 : TERM -> TERM * TERM * TERM;
```

Description Constructor, discriminator and destructor functions for unique existential quantification. Its arguments must be a declaration (constructed with *mk_z_decl*) and two predicates.

Definition
```
mk_z_∃1(⌜Z d⌝,⌜Z p⌝,⌜Z v⌝) = ⌜Z ∃1 d|p•v⌝
```

Errors
```
47912 ?0 is not a Z declaration
47300 ?0 is not a Z unique existential quantification
```

SML
```
val mk_z_∃s : TERM * TERM * TERM -> TERM;
val is_z_∃s : TERM -> bool;
val dest_z_∃s : TERM -> TERM * TERM * TERM;
```

Description The schema existential quantifier constructor. The arguments must be a declaration (constructed using *mk_z_decl*), a predicate and a schema.

Definition
```
mk_z_∃s(⌜Z d⌝,⌜Z p⌝,⌜Z S⌝) = ⌜Z ∃d | p • S⌝
```

Errors
```
47430 ?0 is not a Z schema existential
```

SML
```
val mk_z_∃ : TERM * TERM * TERM -> TERM;
val is_z_∃ : TERM -> bool;
val dest_z_∃ : TERM -> TERM * TERM * TERM;
```

Description Constructor, discriminator and destructor functions for existential quantification. Its arguments must be a declaration (constructed with *mk_z_decl*) and two predicates.

Definition
```
mk_z_∃(⌜Z d⌝,⌜Z p⌝,⌜Z v⌝) = ⌜Z ∃d|p•v⌝
```

Errors
```
47912 ?0 is not a Z declaration
47290 ?0 is not a Z existential quantification
```

SML
```
val mk_z_× : TERM list -> TERM;
val is_z_× : TERM -> bool;
val dest_z_× : TERM -> TERM list;
```

Description The cartesian product constructor.

Definition
```
mk_z_× [⌜Z t1⌝,...,⌜Z tn⌝] = ⌜Z (t1 × ... × tn)⌝
```

Errors
```
47160 ?0 is not a Z cartesian product
```

SML
```
val mk_z_⨾s : TERM * TERM -> TERM;
val is_z_⨾s : TERM -> bool;
val dest_z_⨾s : TERM -> TERM * TERM;
```

Description The sequential composition constructor. Its arguments must both be schemas.

Definition

$mk\_z\_{\fatsemi}s(⌜_Z R⌝,⌜_Z S⌝) = ⌜_Z R \fatsemi S⌝$

Errors

47480 ?0 is not a Z schema composition

SML
```
val mk_z_θ : TERM * string -> TERM;
val is_z_θ : TERM -> bool;
val dest_z_θ : TERM -> TERM * string;
```

Description The theta term constructor. The first argument must be a schema, the second is an optional decoration.

Definition

$mk\_z\_\theta(⌜_Z S⌝,"'") = ⌜_Z \theta S'⌝$

Errors

47170 ?0 is not a Z θ term

SML
```
val mk_z_λ : TERM * TERM * TERM -> TERM;
val is_z_λ : TERM -> bool;
val dest_z_λ : TERM -> TERM * TERM * TERM;
```

Description The lambda constructor. The arguments are a declaration (constructed using *mk_z_decl* q.v.), a predicate and the body of the abstraction.

Definition

$mk\_z\_\lambda(⌜_Z d⌝,⌜_Z p⌝,⌜_Z v⌝) = ⌜_Z \lambda d | p \bullet v⌝$

Errors

47200 ?0 is not a Z λ abstraction
47201 ?0, ?1 and ?2 are inconsistent in Z

SML
```
val mk_z_μ : TERM * TERM * TERM -> TERM;
val is_z_μ : TERM -> bool;
val dest_z_μ : TERM -> TERM * TERM * TERM;
```

Description The definite description constructor. The arguments are a declaration (constructed using *mk_z_decl* q.v.), a predicate and the body of the definite description.

Definition

$mk\_z\_\mu(⌜_Z d⌝,⌜_Z p⌝,⌜_Z v⌝) = ⌜_Z \mu d | p \bullet v⌝$

Errors

47210 ?0 is not a Z μ term

SML

val **mk_z_ℙ** : *TERM −> TERM*;
val **is_z_ℙ** : *TERM −> bool*;
val **dest_z_ℙ** : *TERM −> TERM*;

Description The powerset constructor.

Definition

$mk\_z\_\mathbb{P}\ t = \mathbb{P}\ t$

Errors

47140 ?0 is not a Z powerset

SML

val **mk_z_↾s** : *TERM ∗ TERM −> TERM*;
val **is_z_↾s** : *TERM −> bool*;
val **dest_z_↾s** : *TERM −> TERM ∗ TERM*;

Description The schema projection constructor. Both arguments must be schemas.

Definition

$mk\_z\_\restriction_s(⌜_Z R⌝,⌜_Z S⌝) = ⌜_Z R \restriction S⌝$

Errors

47410 ?0 is not a Z schema projection

8.2 Reasoning about Predicates

SML
signature **ZPredicateCalculus** = *sig*

Description This provides a set of rules of inference, conversions and tactics sufficient for reasoning about the Z predicate calculus in ProofPower. This structure declares the theory *z_language_ps*, which is also used by structures *ZSetTheory* and *ZSchemaCalculus*.

SML
(∗ *Proof Context*: **z_predicates** ∗)

Description A complete proof context for handling the requirements of the Z predicates of the Z language (as opposed to the mathematical tool-kit). It is composed of proof contexts "′z_predicates" and "′z_decl".

Usage Notes It requires theory *z_language_ps*. It is not intended to be mixed with HOL proof contexts.

SML

(∗ *Proof Context*: ′**z_predicates** ∗)

Description A component proof context for handling the requirements of the Z predicates of the Z language (as opposed to the mathematical tool-kit). It remains purely within the Z language, and thus lacks the features found in proof context "′z_decl" which are necessary for a complete treatment of Z predicates. (which may be found in proof context "z_predicates").

Predicates treated by this proof context are constructs formed from:

$=, \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \mathbb{U}, \forall\ D \mid P \bullet V, \exists\ D \mid P \bullet V, \exists_1\ D \mid P \bullet V$

This proof context further handles membership of constructs purely contructed from 𝕌, generic formals, and Z paragraph markers. The language predicate ∈ is treated with the set constructs that it expresses membership of. Schemas (and especially schema references) as predicates are treated by "z_schemas", except that this proof context will replace an ill-formed "schema as predicate" expression with an explicit membership.

Contents Rewriting:

⇔_rewrite_thm, *¬_rewrite_thm*, *eq_rewrite_thm*, *∧_rewrite_thm*, *∨_rewrite_thm*,
∀_rewrite_thm, *⇒_rewrite_thm*, *z_schema_pred_conv1*,
z_∈_u_conv, ‘⊢ ℙ 𝕌 = 𝕌‘, ‘⊢ (𝕌 × 𝕌 × ...) = 𝕌‘
‘⊢ {*lab1* : 𝕌; *lab2* : 𝕌; *lab3*,*lab4* : 𝕌; ... } = 𝕌‘
z_∀_inv_conv, *z_∃_inv_conv*, *simplifications as z_para_pred_canon*

Stripping theorems:

z_¬_in_conv, *z_¬_gen_pred_conv*, *z_∃_elim_conv*, *z_∃₁_conv*,
z_schema_pred_conv1, *z_schema_pred_conv1 pushed in* ¬,
z_∈_u_conv, *z_∈_u_conv pushed in* ¬, *z_∀_inv_conv*,
⇒_thm, *⇔_thm*, *∀_rewrite_thm*, *eq_rewrite_thm*,
simplifications as z_para_pred_canon

Note that we do not break apart a Z ∀ into HOL quantifiers during theorem stripping.

Stripping conclusions:

z_∀_elim_conv, *z_¬_in_conv*, *z_¬_gen_pred_conv*, *z_∈_u_conv*,
z_∈_u_conv pushed in ¬, *z_∃_inv_conv*, *⇔_thm*, *eq_rewrite_thm*,
‘⊢ ∀*a b*•(*a* ∨ ¬*b*) ⇔ (*b* ⇒ *a*)‘, ‘⊢ ∀ *a b*• ¬ *a* ∨ *b* ⇔ *a* ⇒ *b*‘
‘⊢ ∀ *a b*• *a* ∨ *b* ⇔ ¬ *a* ⇒ *b*‘, *z_schema_pred_conv1*, *z_schema_pred_conv1 pushed in* ¬,
simplifications as z_para_pred_canon

Note that we do not break apart a Z ∃ into HOL quantifiers during conclusion stripping.

Rewriting canonicalisation:

∀_rewrite_canon, *z_¬_rewrite_canon*, *∧_rewrite_canon*,
f_rewrite_canon, *z_∀_rewrite_canon*, *z_para_pred_canon*, *z_⇒_rewrite_canon*

Notice in particular the use of the HOL *∀_rewrite_canon*.

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and the list

z_∃_elim_conv2, *ALL_SIMPLE_∃_C* "*simplifications as z_para_pred_canon*",
basic_prove_∃_conv

The existence prover can also handle 1-tuples, 2-tuples, etc, up to 16-tuples. as arguments.

Usage Notes It requires theory *z_language_ps*. It is not intended to be mixed with HOL proof contexts. Use with proof context "′z_decl" to handle declarations properly.

SML

(∗ *Proof Context*: **′z_decl** ∗)

Description A component proof context for handling the requirements of converting Z declarations into their implicit predicates, kept separate from "′z_predicates" due to it introducing a small portion of Z library set theoretic reasoning.

The requirement is met by appropriate treatment of:

set_display ⊆ *set_expression*

during stripping.

Contents

Rewriting:

Stripping theorems:

z_setd_⊆_conv,
and this pushed in through ¬.

Stripping conclusions:

z_setd_⊆_conv,
and this pushed in through ¬.

Notice how this proof context does not use *z_setd_⊆_conv* for rewriting, but leaves such an effect to the proof context concerned with extensional reasoning about the Z library.

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence provers.

Usage Notes It requires theory *z_language_ps*. It is not intended to be mixed with HOL proof contexts. Used with proof context "′z_predicates".

SML

(∗ *Proof Context*: **′z_fc** ∗)

Description A component proof context giving a faster but less general automatic proof capability than the one supplied in most other proof contexts for Z. The automatic proof procedures in the proof context are *z_fc_prove_tac*, *z_fc_prove_conv*. All other fields are blank.

Usage Notes It requires theory *z_language_ps*.

Note that the way proof contexts are merged by *push_merge_pcs* is such that to get the faster automatic proof procedures, one should put *′z_fc* at the end of the list of proof contexts to be merged. For example, to work in the Z predicate calculus with the faster automatic proof procedures, one might use

```
push_merge_pcs["z_predicates", "'z_fc"];
```

SML
```
(* check_is_z : boolean flag *)

val set_check_is_z : bool -> bool;
```

Description This flag, if true (the default), will cause all Z inference rules and tactics that claim to remain in the Z language to check any terms they change (i.e. assumptions and conclusions) for remaining within the Z language. If any fail then the informational message 41004 is used to output text to the user. If the flag is false, no such checks are made. The checks are computationally expensive, and the results may be excessively verbose if terms are not all Z.

The function sets the flag to a specified value and retursn the original value.

Errors
41004 The following subterms in the result are not in the Z language: ?0

SML
```
val all_z_∀_intro : THM -> THM;
```

Description This will Z universally quantify all free variables in the conclusion of a theorem, that do not occur in the assumptions. The declaration part will state the variables are of type ⓩ𝕌⌝, and the predicate part will just be *true*. If no variables can be introduced then the original theorem will be returned.

SML
```
val check_is_z_thm : string -> THM -> THM;
val check_is_z_goal : string -> GOAL -> GOAL;
val check_is_z_term : string -> TERM -> TERM;
val CHECK_IS_Z_T : string -> TACTIC -> TACTIC;
val check_is_z_conv_result : string -> THM -> THM;
```

Description For *check_is_z_thm*, if flag *check_is_z* is true then the conclusion and assumptions of the provided theorem are checked for being within the Z language (except for outermost HOL universal quantification), and informational message 41005 used if not. The string argument is used as the name of the calling function in the error message. If the flag is false then there is no effect. In either case the theorem is passed through unchanged.

check_is_z_goal and *check_is_z_term* are analogous. *CHECK_IS_Z_T* checks each of the subgoals a tactic requests.

check_is_z_conv_result checks that the RHS of the resulting equational theorem, and any assumptions are within the Z language. This allows the RHS side of the equation to have outer HOL universal quantification, and the LHS not to be Z (e.g. in an Z introduction conversion) without complaint.

Errors
41005 In the result of ?0 the following subterms are not in the Z language: ?1

SML
```
val dest_z_term1 : TERM -> Z_TERM;
```

Description This function acts as *dest_z_term* on terms (i.e. expressions and predicates) in the Z language, but makes additional checks. This is in contrast to *dest_z_term* whose intended purpose is categorisation and destruction of Z terms with minimal overhead.

The function does not recursively check the constituents of the outermost Z syntactic construction. For example, it does not check that the constituents of a Z *decl* are individually in the syntactic category *dec*.

Errors
```
41002 Not within the Z language due to subterm ?0
```

SML
```
val is_z_term1 : TERM -> bool;
```

Description Tests if a given HOL term is valid Z in its top level structure.

Uses Recursively in well-formedness checks.

See Also *is_z_term* for a less complete check of top level structure, *is_z* for a full traversal of the terms structure.

SML
```
val is_z : TERM -> bool;
val is_all_z_type : TYPE -> bool;
```

Description If the term (i.e. expression or predicate) or type given is in the range of the Z mapping for a term or type respectively then these functions will return true. They will otherwise return false, unless the only form of incorrectness is that the constituents of a Z syntactic construction are not as required. For example, it does not check that the constituents of a Z *decl* are individually in the syntactic category *dec*.

The test traverses the provided object by using *full_dest_z_term* (and *dest_z_type* for constituent-types) - the test is passed if the entire term can be broken into non-type and non-term parts (i.e. primitives such as strings or integers). Otherwise it will fail with the given error message.

Note that a term is a subterm of itself for these purposes.

See Also *is_z_term* and *is_z_term1*.

Errors
```
41002 Not within the Z language due to subterm ?0
41003 Not within the Z language due to containing type ?0
```

SML
```
val not_z_subterms : TERM -> TERM list;
```

Description This function will return a list (perhaps empty) of all the subterms that prevent a term (i.e. expression or predicate) being within the Z language (by the checks of *is_z*, q.v.), starting with the rightmost subterm that is not Z. The subterms given will be maximal in the sense that subterms of those given will not be included in the list.

SML
```
val set_u_simp_eqn_cxt : EQN_CXT -> string -> unit;
val get_u_simp_eqn_cxt : string -> (EQN_CXT * string)list;
```

Description *set_u_simp_eqn_cxt ec pc_name*; sets the "icl′u_simp" entry of the dictionary of nets field of the proof context called "pc_name" to the equational context *ec*. This means that when this named proof context has been made the current proof context (probably merged with others) it will be "aware" of the equational contexts potential 𝕌 simplifications.

For example, to make the current proof context aware of the 𝕌 simplifications of the (in scope) theory "thy" one would do:

```
new_pc "thy_u_simp_pc";
set_u_simp_eqn_cxt (theory_u_simp_eqn_cxt "thy") "thy_u_simp_pc";
push_merge_pcs ("thy_u_simp_pc" :: other_desired_proof_contexts);
```

One could later update information about the theory (e.g. because new definitions have been added) by:

```
set_u_simp_eqn_cxt (theory_u_simp_eqn_cxt "thy") "thy_u_simp_pc";
set_merge_pcs ("thy_u_simp_pc" :: other_desired_proof_contexts);
```

set_u_simp_eqn_cxt ex pc_name; extracts the 𝕌 simplification subfields of the named proof context. These subfields are each an equational context paired with its original source proof context name.

See Also *u_simp_eqn_cxt*, *theory_u_simp_eqn_cxt*

Errors
```
51010 There is no proof context with key ?0
51014 Proof context ?0 was created in theory ?1 at a
      point now either not in scope, deleted or modified
51016 Proof context ?0 has been committed
```

SML
```
val theory_u_simp_eqn_cxt : string -> EQN_CXT;
```

Description *theory_u_simp_eqn_cxt theory_name* takes the named theory and checks it for theorems, definitions and axioms that could be used for creating proof context entries used by *z_∈_u_conv*.

A theorem is checked by canonicalising it, and accepting those resulting theorems that are equations between an expression that is not 𝕌, and 𝕌. Those that can be so used are processed by *thm_eqn_cxt* and then added to the equational context being generated.

Uses This function is primarily intended for the automatic extraction and processing of the given set and free type definitions of a theory, when building a proof context for a particular theory.

Note that equational contexts can be joined using list append, @.

See Also *u_simp_eqn_cxt*, *set_u_simp_eqn_cxt*

Errors As the failures of *get_defn*.

SML
```
val u_simp_eqn_cxt : THM list -> EQN_CXT;
```

Description *u_simp_eqn_cxt thms* takes each member of *thms*, and checks and then processes it for use in creating proof context entries used by *z_∈_u_conv*.

The check is that each theorem is canonicalised with the current proof context's canonicalisation function. For each resulting theorem, if it is a universally quantified equation of sets then it is processed by *thm_eqn_cxt* and added into the created equational context. If it is not equation of sets the theorem is ignored.

Uses This function is primarily intended to aid the construction of proof contexts containing 𝕌 simplification material.

Note that equational contexts can be joined using list append, @.

See Also *theory_u_simp_eqn_cxt*, *set_u_simp_eqn_cxt*

SML

```
val z_basic_prove_conv : THM list -> CONV;
```

Description This is the conversion used for the automatic proof conversion (*pr_tac* field) of most supplied proof contexts, and is a reasonable, general-purpose, automatic proof conversion. It will either prove the theorem with the given conclusion, or fail.

In summary it will:

1. Set the term as the goal of the subgoal package (or, more exactly, *tac_proof*).
2. Attempt to rewrite the term with the current default rewrite rules and given theorems.
3. Repeatedly apply *strip_tac* to the goal.
4. Attempt variable elimination, using *all_var_elim_asm_tac*.
5. In all resulting goals replace all Z quantifiers by their HOL equivalents in both assumptions and goal.
6. Apply *all_asm_fc_tac* once to each resulting goal.
7. Attempt to prove the resulting goals with resolution for up to 3 resolution steps, with goal's negated conclusion as a resolvant that must be used, and the assumptions as possible other resolvants.
8. Attempt to prove the resulting goals with resolution for up to 3 resolution steps amongst just the assumptions.
9. If the proof is successful, return ⊢ *term* ⇔ *T* and otherwise fail.

Note that in the stripping step may result in more than one subgoal, and thus the phrase "resulting goals" is used above.

Under the current interface to resolution this is equivalent to:

```
fun z_basic_prove_conv thms tm =
⇔_t_intro (
tac_proof(([],tm),
        TRY_T (rewrite_tac thms)  THEN
        REPEAT strip_tac THEN
        TRY_T all_var_elim_asm_tac THEN
        (z_quantifiers_elim_tac THEN
        basic_res_tac2 3 [⊢ ∀ x • x = x]
        ORELSE_T basic_res_tac3 3 [⊢ ∀ x • x = x]))
);
```

In the implementation however, partial evaluation with just the theorems is allowed.

Errors

```
76001 Could not prove theorem with conclusion ?0
```

SML

```
val z_basic_prove_tac : THM list -> TACTIC;
```

Description This is the tactic used for the automated proof tactic (the *pr_tac* field) of most supplied Z proof contexts, and is a reasonable, general-purpose, automatic proof tactic for Z.

In summary it will:

1. Attempt variable elimination, using *all_var_elim_asm_tac*.
2. Extract the assumption list, rewrite each extracted assumption with the current default rewrite rules and given theorems, and strip the results back into the assumption list.
3. Attempt to rewrite the conclusion of the resulting goal with the current default rewrite rules and given theorems.
4. Repeatedly apply *strip_tac* to the conclusions of the resulting goals.
5. Again attempt variable elimination, using *all_var_elim_asm_tac*.
6. In all resulting goals replace all Z quantifiers by their HOL equivalents in both assumptions and goal. This has no effect on any resulting goal if it is unsolved.
7. Apply *all_asm_fc_tac* once to each resulting goal.
8. Attempt to prove each of the resulting goals with resolution for up to 3 resolution steps, with goal's negated conclusion as a resolvant that must be used, and the assumptions as possible other resolvants. This has no effect on any resulting goal if it is unsolved.
9. Attempt to prove each of the resulting goals with resolution for up to 3 resolution steps amongst just the assumptions. This has no effect on any resulting goal if it is unsolved.

Note that either stripping step may result in more than one subgoal, and thus the plural "resulting goals".

Under the current interface to resolution this is

```
fun z_basic_prove_tac thms =
TRY_T all_var_elim_asm_tac THEN
DROP_ASMS_T (MAP_EVERY (strip_asm_tac o rewrite_rule thms) o rev) THEN
(TRY_T (rewrite_tac thms)) THEN
REPEAT strip_tac THEN
TRY_T all_var_elim_asm_tac THEN_TRY
(z_quantifiers_elim_tac
 THEN basic_res_tac2 3 [⊢ ∀ x • x = x]
 ORELSE_T basic_res_tac3 3 [⊢ ∀ x • x = x]);
```

SML
```
val Z_DECL_C : CONV -> CONV;
```

Description *Z_DECL_C* applies the supplied conversion to each member of a declaration and returns the conjunction of the results. It fails if its conversion fails on any member of the declaration.

```
fun z_decl_pred_conv = Z_DECL_C z_dec_pred_conv;
```

will convert a valid Z declaration into its implicit Z predicate.

Errors
```
47912 ?0 is not a Z declaration
41012 Supplied conversion failed on one or more members of ?0
```

SML
```
val Z_DECL_INTRO_C : CONV -> CONV;
```

Description *Z_DECL_INTRO_C* applies the supplied conversion to each conjunct of a predicate, flattening the conjunctive structure. If this is successful, it attempts to produce a declaration from the results.

Z_DECL_INTRO_C z_pred_dec_conv will convert certain Z predicates into Z declarations implicitly containing the predicates, and otherwise will fail.

Errors
```
41013 ?0 not of the form: ⓩtrue⌝ or ⓩc1 ∧ ...⌝ where all the ci
      may have the supplied conversion applied
41014 ?0 when converted to ?1 cannot be viewed as a declaration
```

The conversion fails if the supplied conversion fails on any conjunct, returning the error message of that conversion application.

SML

val **z_decl_pred_conv** : $CONV$;

Description A conversion which rewrites an explicit Z declaration (i.e. a *decl*) to its implicit predicate. An Z declaration may be found, e.g., as a component of a Z horizontal schema. A declaration consists of a list of components (each a *dec*), that are individually converted into predicates, and the results conjoined. The predicate implicit in a declaration, D, is also sometimes referred to as the "predicate from D".

The function is defined much as if by the following:

Definition

val z_decl_pred_conv = Z_DECL_C z_dec_pred_conv;

Thus the handling of the individual declarations is as shown in the following examples:

Conversion

$$\frac{}{\vdash\ {}_{ML}\ulcorner decl\_of\ {}_{Z}\ulcorner[x : X;\ y,\ z : Y;\ S]\urcorner\urcorner \Leftrightarrow x \in X \land \{y, z\} \subseteq Y \land S}\qquad z\_decl\_pred\_conv\ (decl\_of\ {}_{Z}\ulcorner[x : X;\ y,\ z : Y;\ S]\urcorner)$$

and

Conversion

$$\frac{}{\vdash\ {}_{ML}\ulcorner decl\_of\ {}_{Z}\ulcorner[\,]\urcorner\urcorner \Leftrightarrow true}\qquad z\_decl\_pred\_conv\ (decl\_of\ {}_{Z}\ulcorner[\,]\urcorner)$$

Note that a declaration on its own is not a Z expression, though it may be correctly embedded within certain forms of Z expressions.

See Also *z_dec_pred_conv*

Errors

47912 ?0 is not a Z declaration

SML

val **z_dec_pred_conv** : $CONV$;

Description A conversion which rewrites a *dec* part of a declaration to its implicit predicate. A *decsexp* type of declaration remains unchanged (since *decsexp* and *predsexp* are, in fact, the same thing).

Conversion

$$\frac{}{\vdash {}_{\text{ML}}\ulcorner mk\_z\_dec([{}_{\text{Z}}\ulcorner x\urcorner],{}_{\text{Z}}\ulcorner X\urcorner)\urcorner \Leftrightarrow x \in X} \quad \begin{array}{l} z\_dec\_pred\_conv \\ (mk\_z\_dec([{}_{\text{Z}}\ulcorner x\urcorner],{}_{\text{Z}}\ulcorner X\urcorner)) \end{array}$$

and

Conversion

$$\frac{}{\vdash {}_{\text{ML}}\ulcorner mk\_z\_dec([{}_{\text{Z}}\ulcorner x\urcorner,...],{}_{\text{Z}}\ulcorner X\urcorner)\urcorner \Leftrightarrow \{x_1,...\} \subseteq X} \quad \begin{array}{l} z\_dec\_pred\_conv \\ (mk\_z\_dec([{}_{\text{Z}}\ulcorner x\urcorner,...],{}_{\text{Z}}\ulcorner X\urcorner)) \end{array}$$

and

Conversion

$$\frac{}{\vdash S \Leftrightarrow S} \quad \begin{array}{l} z\_dec\_pred\_conv \\ {}_{\text{Z}}\ulcorner S\urcorner \end{array}$$

where S is a schema (here promoted to a predicate). In this last case if the schema as predicate expression is not well-formed Z (perhaps because of substitution of variables) the result will be further converted to correct Z of the form:

$binding \in schema$

Note that a declaration on its own is not a Z expression, though it may be correctly embedded within certain forms of Z expressions.

See Also *z_pred_dec_conv*

Errors

41010 ?0 is not a declaration

SML

val **z_fc_prove_conv** : $THM\ list\ {-}{>}\ CONV$

Description This is the automatic proof conversion supplied in the proof context *'z_fc*. It is based on the automatic proof tactic *z_fc_prove_tac*, q.v., and is defined, in effect as:

```
fun z_fc_prove_conv (thms: THM list) : CONV = (fn tm =>
        ⇔_t_intro (tac_proof(([],tm), z_fc_prove_tac thms))
);
```

SML

```
val z_fc_prove_tac : THM list -> TACTIC
```

Description The resolution-based proof procedure *z_basic_prove_tac* supplied as the automatic proof tactic in many of the the proof contexts for Z may be found to be somewhat slow on complex problems. *z_fc_prove_tac* supplies a less general but quicker alternative based on forward chaining (in the sense of *fc_tac*. It is supplied as the automatic proof tactic field in the proof context *'z_fc*. Its effect may be described as follows:

1. Attempt variable elimination, using *all_var_elim_asm_tac*.
2. Extract the assumption list, rewrite each assumption as it is extracted with the current default rewrite rules and given theorems, and strip the results back into the assumption list.
3. Attempt to rewrite the conclusions of the resulting goals with the current default rewrite rules and the argument theorems.
4. Apply *contr_tac*.
5. Again attempt variable elimination, using *all_var_elim_asm_tac*.
6. In all resulting goals replace all Z quantifiers by their HOL equivalents.
7. Apply *all_asm_fc_tac*.
8. Generate (universally quantified) implications from the assumptions using the canonicalisation function *fc_canon1*. The go through three forward chaining passes (in the sense of *fc_tac*) using these implications as a starting point. At the end of each pass any generated results are both stripped into the assumptions and processed with *fc_canon1* to be passed on as additional implications for the subsequent pass.

For example, the tactic will prove the following goal:

```
([], ⓩ
        (∀ x1 : ℤ • x1 ∈ A ⇒ x1 ∈ B) ∧
        (∀ x1 : 𝕌; x2 : 𝕌 • (x1, x2) ∈ B ◁ x ⇔ (x1, x2) ∈ B ◁ x′) ∧
        x1 ∈ A ∧
        x1 ∈ ℤ ∧
        (x1, x2) ∈ x ⇒ (x1, x2) ∈ x′
⌝);
```

SML

```
val z_gen_pred_elim : TERM list -> THM -> THM;
val z_gen_pred_elim1 : TERM -> THM -> THM;
```

Description Eliminate (some of) the generic formals of a generic predicate for actual values. If possible, the theorem will be type instantiated to allow generic formals to match the types of the supplied *TERM list*, otherwise the rule fails.

Rule

$$\frac{\Gamma \vdash [X1,...]\ (t[X1,...])}{\Gamma \vdash t[t1,...]} \qquad z\_gen\_pred\_elim\ [t1,...]$$

z_gen_pred_elim1 is just like *z_gen_pred_elim* except that its argument is a term rather than a list of terms. *z_gen_pred_elim1*⌜Z(*t1*, ...)⌝ is equivalent to *z_gen_pred_elim*[⌜Z *t1*⌝, ...]; if the term argument, *t*, is not a Z tuple, *z_gen_pred_elim1*⌜Z *t*⌝ is equivalent to *z_gen_pred_elim*[⌜Z *t*⌝]. The advantage of *z_gen_pred_elim1* is that in a call such as *z_gen_pred_elim1*⌜Z(𝕌, 𝕌, 𝕌)⌝, the Z type inferrer can assign a more general type to the occurrences of *U* than it does in the call *z_gen_pred_elim*[⌜Z 𝕌⌝, ⌜Z 𝕌⌝, ⌜Z 𝕌⌝].

Errors

41033 *?0 is not of the form:* ⌜Z Γ ⊢ [X1,...]t⌝ *where the types of the theorem can be instantiated to allow the types of the generic formals to match the types of the term list*
41034 *?0 is not of the form:* ⌜Z Γ ⊢ [X1,...]t⌝ *where there are sufficient Xi to match the supplied term list*

SML

```
val z_gen_pred_intro : TERM list -> THM -> THM;
```

Description Introduce a list of generic formals. The *TERM list* argument is of variables. Their types will be ignored, they are replaced by the variables ⌜Z *var* ⦂ ℙ *var*⌝.

Rule

$$\frac{\Gamma \vdash t[X1,...]}{\Gamma \vdash [X1,...]\ (t[X1,...])} \qquad z\_gen\_pred\_intro\ [X1,...]$$

Errors

3007 *?0 is not a term variable*
6005 *?0 occurs free in assumption list*

SML

```
val z_gen_pred_tac : TACTIC;
```

Description A tactic to eliminate generic predicates.

Tactic

$$\frac{\{\Gamma\}\ [X1,...]\ t}{\{\Gamma\}\ t} \qquad z\_gen\_pred\_tac$$

Errors

41035 *conclusion of goal is not of the form* ⌜Z [X1,...] t⌝

SML

```
val z_gen_pred_u_elim : THM -> THM;
```

Description Substitute 𝕌 for each of the generic formals of a generic predicate.

Rule

$$\frac{\Gamma \vdash [X1,X2,...]\ (t[X1,X2,...])}{\Gamma \vdash t[\mathbb{U},\mathbb{U}...]} \quad \textit{z\_gen\_pred\_u\_elim}$$

Each occurrence of 𝕌 is instantiated to the same type as the corresponding generic formal parameter.

SML

```
val z_get_spec : TERM -> THM;
```

Description This function returns the specification of a constant, based on its definiting theorem and, if one can be found, a consistency theorem. The defining theorem may have been created by Z paragraph processing, *new_axiom*, or a HOL definitional mechanism. This function should be the Z user's interface to definitional theorems, as *get_spec*(q.v.) is for the HOL user.

z_get_spec ⌜*const*⌝ will find the (first) definition or axiom in scope stored under key "name of *const*", in the theory in which the in-scope constant named *const* was defined. A definition will be taken in preference to an axiom in the same theory. *z_get_spec* ⌜*const t1 t2 ...*⌝ (i.e. a constant applied to an arbitrary number of arguments in HOL) will act as *z_get_spec* ⌜*const*⌝. This choice is made in the assumption that a naming convention has been followed that such a definition (or axiom) should be the definition of the constant named *const*. This convention has been followed throughout the implementation of ProofPower. In addition, there can only be one definition of a particular constant in scope (though the conventional key might be used elsewhere, or not at all). If there is no such constant in scope, or no definition with the given key, then the function fails.

If the definitional theorem is of the form:

```
⊢ ConstSpec p c
```

(i.e. its introduction requires a consistency assumption) the function will seek for a theorem or axiom stored with key *const* ^ "*_consistent*", starting at the theory in which the definition was found, and working "out" to the current theory. If conventions have been followed this theorem should be of the form:

```
Γ ⊢ Consistent p
```

(Ideally there should be no assumptions in the theorem, but the function caters for their presence.) If a theorem of this form is found then the theorem:

```
β_rule ‘Γ ⊢ p c‘
```

is formed. If not, then the theorem:

```
β_rule ‘Consistent p ⊢ p c‘
```

is formed. In all of the above cases, (i.e. with or without *ConstSpec*), the theorem formed is checked to see whether it is the definiton formed from processing a Z paragraph. If so, then the conclusion of the theorem is converted into a predicate (by *z_para_pred_conv*), and then returned as the result of *z_get_spec*. If not, then the theorem is returned without further processing as result of *z_get_spec*.

Errors

```
46005  There is no constant with name ?0 in scope
46006  There is no definition or axiom with key ?0 in
       the declaration theory of the constant
46009  ?0 is not a constant, or a constant applied to some arguments
```

SML

```
val z_intro_gen_pred_tac : (TERM * TERM) list -> TACTIC;
```

Description A tactic to introduce a generic predicate as the goal. The term list argument pairs is of a term and a variable (that is appropriate to be a generic formal), with the same set type i.e. the second is of the form ⌜Z *var* : ℙ ′*var*⌝.

Tactic

$$\frac{\{\Gamma\}\ t[t1,...]}{\{\Gamma\}\ [X1,...]\ (t[X1,...])} \qquad z\_intro\_gen\_pred\_tac\ [(t1,X1),...]$$

where either t_i is the same as X_i, or X_i does not appear free in the conclusion, $t[t1,...]$, of the original goal.

N.B. this tactic strengthens the goal, i.e. it may result in unprovable subgoals even when the original goal was provable.

Errors

28082 ?0 does not appear free in the goal
28083 ?0 appears free in the goal and is not the same as ?1
41032 ?0 is not of the form: ⌜Z var : ℙ ′var⌝
41036 ?0 does not have the same type as ?1

SML

```
val z_intro_∀_tac : TERM -> TACTIC;
```

Description Introduce a Z universal with reference to a binding.

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{\ \Gamma\ \}\ \forall\ x_1{:}\mathbb{U};\ ...\ |\ true\bullet\ t[x_1/t_1,...]} \qquad z\_intro\_\forall\_tac\ ⌜_Z(x_1 \mathrel{\widehat{=}} t_1,\ ...)⌝$$

Errors

41029 ?0 cannot be interpreted to be of the form: ⌜Z(x₁ ≙ t₁, ...)⌝

SML

```
val z_para_pred_canon : CANON;
val z_para_pred_conv : CONV;
```

Description This canonicalisation function and conversion change Z paragraphs to Z predicates. This change is also the one *z_get_spec* does, where appropriate.

Some paragraphs entered by the Z parser have "markers" applied to the rest of the theorem body to indicate their origin (i.e. the kind of paragraph). In addition the form of the term is likely to have a an explicit declaration as the left conjunct, rather than a "predicate implicit in a declaration". *z_para_pred_canon* is a canonicalisation function that removes these markers and converts, if present, a left conjunct declaration as *z_decl_pred_conv* would; *z_para_pred_conv* is a conversion that has the equivalent effect.

The following are instances in which markers are used:

Constraint Definitions
Free Type Definitions
Given Set Definitions
Axiomatic Definitions
Schema Boxes
Abbreviation Definitions

Example

If the following is entered:

```
╒[X,Y]══
│Ex : ℙ(X × Y)
├─────
│Ex = {}
└─────
```

z_para_pred_canon given the defining theorem, returns a singleton list containing:

$\vdash \ulcorner \forall\ X\ \ Y \bullet \ulcorner_Z Ex[X,\ Y] \in \mathbb{P}\ (X \times Y) \land Ex[X,\ Y] = \{\}\urcorner\urcorner$

Both functions remain within the Z language, though this is not checked, with the caveat on HOL universals representing generic formals.

Errors

41080 No Z markers found applied to conclusion body of ?0
41082 No Z markers found applied to body of ?0

SML
```
val z_pred_decl_conv : CONV;
```

Description A conversion which, given a predicate comprising a conjunction of the forms recognised by *z_pred_dec_conv*, rewrites the predicate as a declaration,

The function is defined much as if by the following:

Definition
```
val z_pred_decl_conv = Z_DECL_INTRO_C z_dec_pred_conv;
```

Thus the handling of the conjuncts is as shown in the following examples:

Conversion

$$\frac{}{\vdash x \in X \wedge \{y, z\} \subseteq Y \wedge S \Leftrightarrow {}_{ML}\ulcorner decl\_of {}_{Z}\ulcorner[x : X; y, z : Y; S]\urcorner\urcorner}$$ *z_pred_decl_conv* $\ulcorner_Z x \in X \wedge \{y, z\} \subseteq Y \wedge S\urcorner$

and

Conversion

$$\frac{}{\vdash true \Leftrightarrow {}_{ML}\ulcorner decl\_of {}_{Z}\ulcorner[]\urcorner\urcorner}$$ *z_pred_decl_conv* $\ulcorner_Z true\urcorner$

See Also *z_decl_pred_conv*

Errors
```
41011 ?0 cannot be rewritten to a declaration
```

SML
```
val z_pred_dec_conv : CONV;
```

Description A conversion which, given a certain form of predicate, rewrites the predicate as the *dec* component of a declaration. This acts as an inverse to the conversion *z_dec_pred_conv*, the four forms recognised being as shown below:

Conversion

$$\frac{}{\vdash x \in X \Leftrightarrow {}_{ML}\ulcorner mk\_z\_dec([\ulcorner_Z x\urcorner],\ulcorner_Z X\urcorner)\urcorner}$$ *z_pred_dec_conv* $\ulcorner_Z x \in X\urcorner$

where the x must be variable, and

Conversion

$$\frac{}{\vdash \{x_1,...\} \subseteq X \Leftrightarrow {}_{ML}\ulcorner mk\_z\_dec([\ulcorner_Z x\urcorner,...],\ulcorner_Z X\urcorner)\urcorner}$$ *z_pred_dec_conv* $\ulcorner_Z \{x_1,...\} \subseteq X\urcorner$

where the x_i must be variables, and

Conversion

$$\frac{}{\vdash S \Leftrightarrow S}$$ *z_pred_dec_conv* $\ulcorner_Z S\urcorner$

and

Conversion

$$\frac{}{\vdash (\theta S \in S) \Leftrightarrow S}$$ *z_pred_dec_conv* $\ulcorner_Z \theta S \in S\urcorner$

See Also *z_dec_pred_conv*

Errors
```
41011 ?0 cannot be rewritten to a declaration
```

SML

```
val z_push_consistency_goal : TERM -> unit;
```

Description *z_push_consistency_goal* ⌜$_Z$*const*⌝ will first determine the specification theorem of *const*, by executing *z_get_spec*. The *const* may either be a constant, or a constant applied to a list of arguments. If this theorem has an assumption, it will then push that specification assumption onto the stack of subgoals (using *push_subgoal*, q.v.), as a goal with no assumptions. By how *z_get_spec* is designed, this (single) assumption will be of the form:

```
⌜Consistent (λ vs[x1,...,xn]•p[x1,...,xn])⌝
```

or the consistency has already been proven, and saved, under some assumptions. Only in the former case will the function continue: it will apply a tactic (that may be undone by *undo*) which rewrites the goal to:

```
([], ⌜Z∃ D[x1,...,xn]•p[x1,...,xn]⌝)
```

where D is a declaration of the variables, *x1,...xn* representing the existence witnesses of the n constants declared in one paragraph. Otherwise, if the definition involves generic formals:

```
([], ⌜Z∃ D[x1,...,xn]•p[x1,...,xn]⌝)
```

If not, the function fails.

See Also *save_consistency_thm* to save the result in a conventional manner.

Errors

```
46005 There is no constant with name ?0 in scope
46006 There is no definition or axiom with key ?0 in
      the declaration theory of the constant
46007 Specification of ?0 is not of the form: ‘Consistent (λ vs[x1,...,xn]•p[x1,...,xn]) ⊢ ...‘
46009 ?0 is not a constant, or a constant applied to some arguments
```

SML

```
val z_quantifiers_elim_tac : TACTIC;
```

Description This tactic eliminates all Z ∀, ∃ and $\exists_1$ quantifiers in both conclusion and assumptions, in favour of HOL ∀ and ∃, using *z_∀_elim_conv2*, *z_∃_elim_conv2*, *z_∃₁_conv1*. All declarations introduced will be converted to their implicit Z predicates, and the following simplifications also done throughout:

```
⌜Z{x,y,...} ⊆ s⌝ ---> ⌜x ∈ s ∧ y ∈ s ∧ ...⌝
        (* only when the set display contains just variables *)
⌜Zx ∧ true⌝ ---> ⌜x⌝
⌜Ztrue ∧ x⌝ ---> ⌜x⌝
⌜Zx ⇒ true⌝ ---> ⌜true⌝
```

All assumptions will be stripped back into the assumption list, regardless of whether they were modified, using the current proof context.

This is done to prepare for some further processing, such as resolution. The result is unlikely to be in the Z language It has no effect (rather than failing) if there are no conversions to be done.

Uses Intended for implementing automated proof procedures.

SML

val **z_schema_pred_conv1** : $CONV$;

Description Convert an ill-formed schema as a predicate expression into a statement of a binding being a member of the schema. The input expression is ill-formed if it is of the form

$Z'SchemaPred\ bind\ schema$

where *bind* is not equal to $⌜_Z \theta\ schema⌝$.

Conversion

$$\frac{}{\vdash Z'SchemaPred\ bind\ schema \Leftrightarrow bind \in schema} \quad \begin{array}{l} z\_schema\_pred\_conv1 \\ ⌜Z'SchemaPred\ bind\ schema⌝ \end{array}$$

Uses In correcting the results of functions that leave Z because of substituting into the binding portion of a schema as predicate. In particular, in the proof context "'z_predicates".

Errors

41018 ?0 is not an ill−formed schema as predicate expression

SML

val **z_setd_⊆_conv** : $CONV$

Description Expand out expressions that state that a set display is a subset of some other set. This is particularly aimed at processing declarations of the form $x_1,...,x_n : X$.

Conversion

$$\frac{}{⌜_Z\{x_1,...\} \subseteq X \Leftrightarrow (x_1 \in X \wedge ...)} \quad \begin{array}{l} z\_setd\_\subseteq\_conv \\ ⌜_Z\{x_1,...\} \subseteq X⌝ \end{array}$$

and

Conversion

$$\frac{}{⌜_Z\{\} \subseteq X \Leftrightarrow true} \quad \begin{array}{l} z\_setd\_\subseteq\_conv \\ ⌜_Z\{\} \subseteq X⌝ \end{array}$$

The conversion will all simplify certain subterms involving *true* or terms of the form $x = x$.

See Also *z_setd_∈_ℙ_conv*

Errors

41017 ?0 is not of the form: $⌜_Z\{x_1,...\} \subseteq X⌝$

SML

```
val z_spec_asm_tac : TERM -> TERM -> TACTIC;
val z_spec_nth_asm_tac : int -> TERM -> TACTIC;
```

Description These are two methods of specialising a Z universally quantified assumption. Both leave the old assumption in place, and place the instantiated assumption onto the assumption list using *strip_asm_tac*. If the desired behaviour differs from any of those supplied then use *GET_ASM_T* and its cousins, with *z_∀_elim*, to create the desired functionality.

Tactic

$$\frac{\{\ \Gamma,\ \ulcorner_Z \forall\ D[x_1,...]\ |\ P_1 \bullet\ P_2 \urcorner\ \}\ t}{\{strip\ \ulcorner\ulcorner_Z D'[t_1,...]\urcorner \land \ulcorner_Z P'_1\urcorner \Rightarrow \ulcorner_Z P'_2\urcorner\urcorner,\ \Gamma,\ \ulcorner_Z \forall\ D\ |\ P_1 \bullet\ P_2\urcorner\}\ t1} \quad \begin{array}{l} z\_spec\_asm\_tac \\ \ulcorner_Z \forall\ D[x_1,...]\ |\ P_1 \bullet\ P_2 \urcorner \\ \ulcorner_Z (x_1\ \widehat{=}\ t_1,\ ...)\urcorner \end{array}$$

where D', P'_1 and P'_2 are specialised, and if necessary have bound variable renaming done, appropriately. Remains within the Z language (though failure to do this will be reported to be from *z_∀_elim*).

z_spec_nth_asm_tac uses an assumption number rather than an explicit statement of the assumption to be specialised.

Definition

```
fun z_spec_asm_tac (asm:TERM) (bind:TERM):TACTIC =
        GET_ASM_T asm (strip_asm_tac o z_∀_elim bind);
fun z_spec_nth_asm_tac (n:int) (bind:TERM):TACTIC =
        GET_NTH_ASM_T n (strip_asm_tac o z_∀_elim bind);
```

Errors As the constituents of the implementing functions (e.g. *GET_ASM_T* and *z_∀_elim*).

SML

```
val z_term_of_type : TYPE -> TERM;
val z_type_of : TERM -> TERM;
```

Description *z_term_of_type ty* is a term denoting the set of all elements of the type *ty*. The term is constructed using *mk_z_ℙ*, *mk_z_×*, *mk_z_h_schema*, given sets, and the relation symbol _ ↔ _ in order to display the structure of the type in a Z-like way. *mk_u* is used when all else fails.

For example:

```
z_term_of_type(type_of ⌜Z⟨1, 2, 3⟩⌝)          = ⌜Z ℤ ↔ ℤ⌝
z_term_of_type(type_of ⌜Z(1, 2, 3)⌝)          = ⌜Z ℤ × ℤ × ℤ⌝
z_term_of_type(type_of ⌜Z{1, 2, 3}⌝)          = ⌜Z ℙ ℤ⌝
z_term_of_type(type_of ⌜Z(a≙1, b≙2, c≙3)⌝)    = ⌜Z a, b, c : ℤ⌝
z_term_of_type(type_of ⌜[1; 2; 3]⌝)           = ⌜Z 𝕌⌝
```

Note that the quotation in the last example contains an HOL list display, the type of which, namely ⌜:ℕ *LIST*⌝, lies outside the representation of the Z type system in HOL.

z_type_of returns the set of all elements of the (HOL) type of a particular term.

Definition

```
val z_type_of = z_term_of_type o type_of;
```

SML

val **z_∈_setd_conv** : *CONV*;

Description A conversion of membership of a Z set display into equality with a member of the set.

Conversion

$$\frac{}{\vdash t \in \{\ t_1,\ t_2,\ ...\} = ((t = t_1) \lor (t = t_2)\ ...)}$$

z_∈_setd_conv
⌜$_Z t \in \{\ t_1,\ t_2,\ ...\}$⌝

See Also *z_∈_setd_conv1*

Errors

41015 ?0 is not of the form: ⌜$_Z x \in \{t1,...\}$⌝
41016 ?0 is an ill−formed fragment of the membership of a set display

SML

```
val z_∈_u_conv : CONV;
```

Description Simplifies to *true* a predicate of the forms: ⓩ$x \in S[\mathbb{U}]$⌝, ⓩ$x \subseteq S[\mathbb{U}]$⌝ or a schema as a predicate: ⓩ$[a,b\colon S[\mathbb{U}];\ c : S'[\mathbb{U}];\ ...]$⌝, where $S[\mathbb{U}]$, $S'[\mathbb{U}]$, ... are structures that can be simplified to $\mathbb{U}$. This uses the application of the built-in simplifications listed below, and conversions held in the "icl′u_simp" entry of the dictionary of nets field of the current proof context (the built-in's taking precedence).

Conversion

$$\frac{}{\vdash x \in S[\mathbb{U}] \Leftrightarrow true}$$ *z_∈_u_conv* ⓩ$x \in S[\mathbb{U}]$⌝

Conversion

$$\frac{}{\vdash x \subseteq S[\mathbb{U}] \Leftrightarrow true}$$ *z_∈_u_conv* ⓩ$x \subseteq S[\mathbb{U}]$⌝

Conversion

$$\frac{}{\vdash [a,b\colon S[\mathbb{U}];\ c : S'[\mathbb{U}];\ ...] \Leftrightarrow true}$$ *z_∈_u_conv* ⓩ$[a,b\colon S[\mathbb{U}];\ c : S'[\mathbb{U}];\ ...]$⌝

The conversion starts with the structure $S[\mathbb{U}]$ above. It will attempt to recursively prove equal to $\mathbb{U}$: the argument to $\mathbb{P}$, the consitituent sets of a cartesian tuple, the types of a declaration part of a set abstraction with a true predicate, and the types of a declaration part of a horizontal schema with a true predicate. If it can do so it will then use:

$\vdash \mathbb{P}\ \mathbb{U} = \mathbb{U}$
$\vdash (\mathbb{U} \times \mathbb{U} \times ...) = \mathbb{U}$
$\vdash \{lab1 : \mathbb{U};\ lab2 : \mathbb{U};\ lab3,lab4 : \mathbb{U};\ ...\ \} = \mathbb{U}$
$\vdash [lab1 : \mathbb{U};\ lab2 : \mathbb{U};\ lab3,lab4 : \mathbb{U};\ ...\ ] = \mathbb{U}$

to prove the set equal to $\mathbb{U}$. If it cannot complete the above proof it will use the first applicable conversion of the "icl′u_simp" entry of the dictionary of nets field of the current proof context, and then return to attempting to use the built-in algorithm.

If the set has been reduced to $\mathbb{U}$ the conversion will prove the input term *true*. If the expression cannot be proven the conversion fails.

Uses For stripping in proof contexts, and in eliminating redundant declarations that have been converted to the predicates implicit in them.

See Also *u_simp_eqn_cxt*, *theory_u_simp_eqn_cxt*, and *set_u_simp_eqn_cxt* for creating appropriate proof contexts.

Errors

41061 cannot prove ?0 equal to ⓩ*true*⌝
41062 ?0 is not of the form: ⓩ$x \subseteq s$⌝, ⓩ$x \in s$⌝ *or a schema as a predicate*

SML
```
val z_¬_gen_pred_conv : CONV;
```

Description Convert a negated generic predicate (which is not legal Z) into an existentially quantified negation (and therefore into Z).

Conversion

$$\frac{}{\vdash (\ulcorner\neg \text{⦂}_Z[X1,...]\ pred\urcorner\urcorner) \Leftrightarrow \exists\ X1{:}\mathbb{U};...\ \bullet\ \neg\ pred} \quad z\_\neg\_gen\_pred\_conv\ \ulcorner\neg \text{⦂}_Z[X1,...]\ pred\urcorner\urcorner$$

Uses In stripping for repaired the effects of, e.g., *contr_tac*.

Errors
41031 ?0 is not of the form: $\ulcorner\neg \text{⦂}_Z[X1,...]\ pred\urcorner\urcorner$

SML
```
val z_¬_in_conv : CONV;
```

Description This is a conversion which moves an outermost negation inside other Z predicate calculus connectives using whichever of the following rules applies:

$\neg\neg t$	=	t
$\neg(t1 \wedge t2)$	=	$\neg t1 \vee \neg t2$
$\neg(t1 \vee t2)$	=	$\neg t1 \wedge \neg t2$
$\neg(t1 \Rightarrow t2)$	=	$t1 \wedge \neg t2$
$\neg(t1 \Leftrightarrow t1)$	=	$false$
$\neg(t1 \Leftrightarrow t2)$	=	$(t1 \wedge \neg t2) \vee (t2 \wedge \neg t1)$
$\neg(t1 = t1)$	=	$false$
$\neg(\forall\ D \mid P\bullet\ V)$	=	$\exists\ D \mid P\bullet\ \neg\ V$
$\neg(\exists D \mid P\bullet\ V)$	=	$\forall\ D \mid P\bullet\ \neg\ V$
$\neg(\exists_1\ D \mid P\bullet\ V)$	=	$\forall D \mid P\bullet\neg(V \wedge \forall D' \mid P'\bullet V' \Rightarrow D = D')$
$\neg true$	=	$false$
$\neg false$	=	$true$

Uses Tactic and conversion programming.

Errors
47240 ?0 is not a Z negation
28131 No applicable rules for the term ?0

SML
```
val z_¬_rewrite_canon : THM -> THM list
```

Description This is a canonicalisation function used for breaking theorems up into lists of equations for use in rewriting. It performs the following transformations:

z_¬_rewrite_canon	$(\Gamma \vdash \neg(t1 \vee t2))$	$= (\Gamma \vdash \neg t1 \wedge \neg t2)$
z_¬_rewrite_canon	$(\Gamma \vdash \neg\exists D \mid P\bullet\ V)$	$= (\Gamma \vdash \forall D \mid P\bullet \neg V)$
z_¬_rewrite_canon	$(\Gamma \vdash \neg\neg t)$	$= (\Gamma \vdash t)$
z_¬_rewrite_canon	$(\Gamma \vdash \neg t)$	$= (\Gamma \vdash t \Leftrightarrow false)$

Remains within the Z language, though this is not checked.

See Also *simple_¬_rewrite_canon*, *simple_∀_rewrite_canon*.

Errors
26201 Failed as requested

The area given by the failure will be *fail_canon*.

SML

```
val z_¬_∀_conv : CONV;
val z_¬_∃_conv : CONV;
```

Description $z\_\neg\_\forall\_conv$ converts a negated Z universal quantification to a Z existential quantification.

Conversion

$$\frac{}{\vdash \neg(\forall\ D\ |\ P_1 \bullet\ P_2) \Leftrightarrow (\exists\ D\ |\ P_1 \bullet\ \neg\ P_2)} \quad z\_\neg\_\forall\_conv\ ⓩ\neg(\forall\ D\ |\ P_1 \bullet\ P_2)⌝$$

The dual is $z\_\neg\_\exists\_conv$:

Conversion

$$\frac{}{\vdash \neg(\exists\ D\ |\ P_1 \bullet\ P_2) \Leftrightarrow (\forall\ D\ |\ P_1 \bullet\ \neg\ P_2)} \quad z\_\neg\_\exists\_conv\ ⓩ\neg(\exists\ D\ |\ P_1 \bullet\ P_2)⌝$$

These two functions remain within the Z language, though this is not checked.

Errors

41050 ?0 not of the form: ⓩ¬(∀ D | P_1• P_2)⌝
41051 ?0 not of the form: ⓩ¬(∃ D | P_1• P_2)⌝

SML

```
val z_⇒_rewrite_canon : CANON;
```

Description This canonicalisation expects to be passed the canonicalisations of, e.g., a Z universal or the result of a $z\_\forall\_elim$. These are theorems of the form:

⊢ "*predicate from D*" ∧ P ⇒ V

In these cases it is intended to prove and discard "*predicate from D*" whose conjuncts can be proven true by $z\_\in\_u\_conv$ (q.v.), and a P that is identically *true*.

In fact, each conjunct of the antecedent of the supplied theorem will have $z\_\in\_u\_conv$ attempted upon it, the resulting antecedent will be rewritten with the theorems

⊢ ∀ x:𝕌 • x ∧ true ⇔ x
⊢ ∀ x:𝕌 • true ∧ x ⇔ x

and if the antecedent is thus proven *true* it will be discarded. Remains within the Z language, though this is not checked.

Errors

41083 ?0 is not of the form: $\Gamma \vdash P \Rightarrow Q$
41084 caused no change with ?0

SML

val **z_∀_elim_conv1** : *CONV*;

Description Turn a Z universally quantified predicate into a HOL universally quantified term, eliminating the declaration part of the original quantification using *z_∈_u_conv*. The function fails if the declaration cannot be eliminated.

Conversion

$$\frac{}{\vdash (\forall\ D[x_1,...] \mid P_1 \bullet\ P_2) \Leftrightarrow \ulcorner \forall\ x_1\ ... \bullet\ {}_{Z}\ulcorner P_1 \Rightarrow P_2 \urcorner\urcorner} \quad \begin{array}{l} z\_\forall\_elim\_conv1 \\ {}_{Z}\ulcorner\forall\ D[x_1,...] \mid P_1 \bullet\ P_2\urcorner \end{array}$$

The order of the resulting universally quantified variables will be in a sorted order, rather than what the declaration part might suggest.

Simplifications based on P_i being *true* or *false* will also be applied.

If there are no quantified variables and the declaration is $D[]$, the HOL universal quantification is not generated.

Remains within the Z language (with the caveat of using outer HOL universal quantification).

Uses For stripping in proof contexts.

See Also *z_∀_elim_conv2* and *z_∀_elim_conv*

Errors

41022 ?0 is not of the form: $_{Z}\ulcorner\forall\ D \mid P_1 \bullet\ P_2\urcorner$
41071 ?0 is of the form: $_{Z}\ulcorner\forall\ D \mid P_1 \bullet\ P_2\urcorner$ *but could not eliminate D*

SML

```
val z_∀_elim_conv2 : CONV;
val z_∀_intro_conv1 : CONV;
```

Description *z_∀_elim_conv2* turns a Z universally quantified predicate into a HOL universally quantified term. The result fails to be in the Z language because it contains a declaration used in a position requiring a predicate, which Z does not allow.

Conversion

$$\frac{}{\vdash (\forall\ D[x_1,...] \mid P_1 \bullet\ P_2) \Leftrightarrow \ulcorner\forall\ x_1\ ...\bullet\ {}_{Z}\ulcorner D[x_1,...]\urcorner \land {}_{Z}\ulcorner P_1\urcorner \Rightarrow {}_{Z}\ulcorner P_2\urcorner\urcorner} \quad \begin{array}{l} z\_\forall\_elim\_conv2 \\ {}_{Z}\ulcorner\forall\ D[x_1,...] \mid P_1 \bullet\ P_2\urcorner \end{array}$$

The order of the resulting universally quantified variables will be in a sorted order, rather than what the declaration part might suggest.

z_∀_intro_conv1 undoes this process, returning a theorem whose RHS should be in the Z language.

Conversion

$$\frac{}{\vdash \ulcorner\forall\ x_1\ ...\bullet\ {}_{Z}\ulcorner D[x_1,...] \land P_1 \Rightarrow P_2\urcorner\urcorner \Leftrightarrow \forall\ D[x_1,...] \mid P_1 \bullet\ P_2} \quad \begin{array}{l} z\_\forall\_intro\_conv1 \\ \ulcorner\forall\ x_1\ ...\bullet\ {}_{Z}\ulcorner D[x_1,...] \land P_1 \Rightarrow P_2\urcorner\urcorner \end{array}$$

If there are no quantified variables and the declaration is $D[]$, the HOL universal quantification is not generated by *z_∀_elim_conv2* nor expected by *z_∀_intro_conv1*.

Uses Used in the Z form of *strip_tac*, and handling negations with quantifiers. It will handle paired quantifiers, and quantifiers in any order, so long as the quantifiers and declaration can be matched in names and types.

See Also *z_∀_elim_conv1*, *z_∀_elim_conv*, *z_∀_intro_conv*

Errors

41022 ?0 is not of the form: $_{Z}\ulcorner\forall\ D \mid P_1 \bullet\ P_2\urcorner$
41023 ?0 is not of the form: $\ulcorner\forall\ x_1\ ...\bullet\ Decl \land P_1 \Rightarrow P_2\urcorner$
41024 ?0 is not of the form: $\ulcorner\forall\ x_1\ ...\bullet\ Decl \land P_1 \Rightarrow P_2\urcorner$
where the $\ulcorner x_i\urcorner$ *do not match the declaration*

SML

```
val z_∀_elim_conv : CONV;
```

Description Turn a Z universally quantified predicate into a HOL universally quantified term, converting the declaration part of the original quantification into its implicit predicate.

Conversion

$$\frac{}{\vdash (\forall\ D[x_1,...] \mid P_1 \bullet\ P_2) \Leftrightarrow \ulcorner\forall\ x_1\ ...\bullet\ {}_{Z}\ulcorner\text{"predicate from } D[x_1,...]\text{"} \land P_1 \Rightarrow P_2\urcorner\urcorner} \quad \begin{array}{l} z\_\forall\_elim\_conv \\ {}_{Z}\ulcorner\forall\ D[x_1,...] \mid P_1 \bullet\ P_2\urcorner \end{array}$$

The order of the resulting universally quantified variables will be in a sorted order, rather than what the declaration part might suggest.

If there are no quantified variables and the declaration is $D[]$, the HOL universal quantification is not generated.

Remains within the Z language (with the caveat of using outer HOL universal quantification).

Errors

41022 ?0 is not of the form: $_{Z}\ulcorner\forall\ D \mid P_1 \bullet\ P_2\urcorner$

SML

val **z_∀_elim** : *TERM* −> *THM* −> *THM*;

Description Specialise the variables introduced by a Z universally quantifier to given values of the right type, the values being taken from a binding.

Rule

$$\frac{\Gamma \vdash \forall\ D[x_1,...]\ |\ P_1[x_1,...]\bullet\ P_2[x_1,...]}{\Gamma \vdash \text{"predicate from } D'[t_1,...]\text{"} \wedge P'_1[t_1,...] \Rightarrow P'_2[t_1,...]} \quad z\_\forall\_elim \quad ⓩ(x_1 \mathrel{\widehat{=}} t_1,\ ...\ )⌝$$

where D is a declaration that binds the x_i, that is converted to its implicit predicate by *z_decl_pred_conv*. The theorem may be type instantiated or require bound variable renaming to allow the specialisation to be valid, thus the priming in the result.

If both the supplied binding and the declaration are recognisably "empty" then the universal quantification will be eliminated.

If instead the theorem's conclusion has a single universally quantified variable and the theorem can be type instantiated to match the supplied argument, then that supplied argument will be used directly.

Rule

$$\frac{\Gamma \vdash \forall\ x{:}X\ |\ P_1[x]\bullet\ P_2[x]}{\Gamma \vdash t \in X' \wedge P'_1[t] \Rightarrow P'_2[t]} \quad z\_\forall\_elim \quad ⓩt⌝$$

If neither of the above apply then the supplied binding may instead be anything else that has an appropriate binding type. In such cases projection functions will be used:

Rule

$$\frac{\Gamma \vdash \forall\ D[x_1,...]\ |\ P_1[x_1,...]\bullet\ P_2[x_1,...]}{\Gamma \vdash \text{"predicate from } D[t.x_1,...]\text{"} \wedge P'_1[t.x_1,...] \Rightarrow P'_2[t.x_1,...]} \quad z\_\forall\_elim \quad ⓩt⌝$$

See Also *z_∀_elim_conv2*

Errors

47310 ?0 is not a Z universal quantification
41021 ?0 cannot be interpreted as a binding that matches ?1

SML

val **z_∀_intro1** : *THM* −> *THM*;

Description A rule to introduce a Z universal quantification. The variables to be quantified over must not occur free in the assumptions, and are determined from the form of the input theorem.

Rule

$$\frac{\Gamma \vdash \text{"predicate from D"} \land P_1 \Rightarrow P_2}{\Gamma \vdash \forall\ D \mid P_1 \bullet P_2} \qquad z\_\forall\_intro1$$

where "predicate from D" is converted to a declaration in which this predicate is implicit by *Z_DECL_INTRO_C z_pred_dec_conv*.

An arbitrary conjunctive structure is allowed in "D as a predicate", including repeated bindings of single variables: only the ordering, as opposed to the nesting is significant in the conjunctive structure. The predicate *true* is converted to the empty declaration.

See Also *z_∀_intro* for implicit $x_i \in \mathbb{U}$ conjuncts, *all_z_∀_intro*, *z_∀_intro_conv1*.

Errors

6005 ?0 occurs free in assumption list
41026 ?0 not of the form '$\Gamma \vdash$ *"predicate from D"* $\land P_1 \Rightarrow P_2$*'*
41027 ?0 cannot be made into a declaration

SML

val **z_∀_intro_conv** : *CONV*;

Description *z_∀_intro_conv* converts an arbitrary simple HOL universally quantified term into the corresponding Z, returning a theorem whose RHS should be in the Z language (though this is not checked for).

Conversion

$$\frac{}{\vdash \ulcorner\forall\ x_1\ ...\bullet\ P\urcorner \Leftrightarrow (\forall\ x_1{:}\mathbb{U};\ ...\ \mid\ true\bullet\ P)} \qquad \begin{array}{l} z\_\forall\_intro\_conv \\ \ulcorner\forall\ x_1\ ...\bullet\ P\urcorner \end{array}$$

This conversion cannot introduce a Z universal quantification with an empty declaration.

See Also *z_∀_intro_conv1*

Errors

41047 ?0 is not of the form: $\ulcorner\forall\ x_1\ ...\ \bullet\ P\urcorner$

SML

val **z_∀_intro** : $TERM\ list\ \text{->}\ THM\ \text{->}\ THM$;

Description A rule to introduce a Z universal quantification. The variables to be quantified over must not occur free in the assumptions, and are determined from the variables from the supplied list.

Rule

$$\frac{\Gamma \vdash P_1 \Rightarrow P_2}{\Gamma \vdash \forall x_1{:}\mathbb{U};... \mid P_1 \bullet P_2} \qquad z\_\forall\_intro\ [⌜_Z x_1⌝,...]$$

or else:

Rule

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x_1{:}\mathbb{U};... \mid true \bullet P_2} \qquad z\_\forall\_intro\ [⌜_Z x_1⌝,...]$$

An arbitrary conjunctive structure is allowed, including repeated bindings of single variables: only the ordering, as opposed to the nesting is significant in the conjunctive structure.

See Also *z_∀_intro1* for use without additional $x_i \in \mathbb{U}$, *all_z_∀_intro*, *z_∀_intro_conv1*.

Errors

3007 ?0 is not a term variable
6005 ?0 occurs free in assumption list

SML

val **z_∀_inv_conv** : $CONV$;

Description Simplifies a Z universal quantification whose predicate or constraint is invariant w.r.t. the free variables bound by the declaration.

Conversion

$$\frac{}{\vdash \forall\ D \mid P_1 \bullet P_2 \Leftrightarrow (\forall\ D \mid P_1 \bullet false) \lor P_2} \qquad z\_\forall\_inv\_conv\ ⌜_Z \forall\ D \mid P_1 \bullet P_2⌝$$

if P_2 has no free variables bound by D, and

Conversion

$$\frac{}{\vdash \forall\ D \mid P_1 \bullet P_2 \Leftrightarrow P_1 \Rightarrow (\forall\ D \mid true \bullet P_2)} \qquad z\_\forall\_inv\_conv\ ⌜_Z \forall\ D \mid P_1 \bullet P_2⌝$$

if P_1 has no free variables bound by D, and

Conversion

$$\frac{}{\vdash \forall\ D \mid P_1 \bullet P_2 \Leftrightarrow P_1 \Rightarrow (\forall\ D \mid true \bullet false) \lor P_2} \qquad z\_\forall\_inv\_conv\ ⌜_Z \forall\ D \mid P_1 \bullet P_2⌝$$

if both have no free variables bound by D. The appropriate simplification will be avoided where the predicate P_1, is ⌜$_Z$*true*⌝ or the value, P_2 is ⌜$_Z$*false*⌝.

See Also *z_∃_inv_conv*

Errors

47310 ?0 is not a Z universal quantification
41025 ?0 is not of the form: ⌜$_Z \forall\ D \mid P_1 \bullet P_2$⌝ *where at least one of* P_1 *or* P_2 *are unbound by* D

SML

```
val z_∀_rewrite_canon : CANON;
```

Description Take a possibly Z universally quantified theorem and make it into, as far as possible, a HOL universally quantified theorem usable for rewriting.

Canon

$$\frac{\Gamma \vdash ⓩ(\forall\ D[x_1,...] \mid P_1 \bullet P_2)⌝}{[\Gamma \vdash ⌜\forall\ x_1\ ...\bullet\ ⓩ\text{"predicate from } D[x_1,...]\text{"} \land P_1 \Rightarrow P_2⌝⌝]} \quad z\_\forall\_rewrite\_canon$$

See *z_decl_pred_conv* for a description of the conversion of a declaration to its implicit predicate.

Remains within the Z language (under the relaxation that allows outermost HOL universals), though this is not checked.

See Also *z_defn_canon*

Errors

41081 ?0 is not of the form: $ⓩ(\forall\ D \mid P_1 \bullet P_2)⌝$

SML

```
val z_∀_tac : TACTIC;
```

Description Eliminate a Z universal in a goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ \forall\ D \mid P \bullet V}{\{\ \Gamma\ \}\ \text{"predicate from } D'\text{"} \land P' \Rightarrow V'} \quad z\_\forall\_tac$$

D is converted to its implicit predicate by *z_decl_pred_conv*. D, P and V are primed in the result because the tactic may require some renaming to avoid, e.g. variable capture of the names of free variables in the assumption list.

Errors

41030 Conclusion of the goal is not of the form: $ⓩ\forall\ D \mid P \bullet V⌝$

SML

```
val z_∃_elim_conv1 : CONV;
```

Description Turn a Z existentially quantified predicate into a HOL existentially quantified term, eliminating the declaration part of the original quantification using *z_∈_u_conv*. The function fails if the declaration cannot be eliminated.

Conversion

$$\frac{}{\vdash (\exists\ D[x_1,...] \mid P_1 \bullet\ P_2) \Leftrightarrow \ulcorner \exists\ x_1\ ... \bullet\ \ulcorner_Z P_1 \land P_2 \urcorner \urcorner} \quad z\_\exists\_elim\_conv1\ \ulcorner_Z \exists\ D[x_1,...] \mid P_1 \bullet\ P_2 \urcorner$$

The order of the resulting existentially quantified variables will be in a sorted order, rather than what the declaration part might suggest.

Simplifications based on P_i being *true* or *false* will also be applied.

If there are no quantified variables and the declaration is $D[]$, the HOL existential quantification is not generated.

Uses For stripping in proof contexts.

See Also *z_∃_elim_conv2*, *z_∃_elim_conv*

Errors

41042 ?0 is not of the form: $\ulcorner_Z \exists\ D \mid P_1 \bullet\ P_2 \urcorner$

41043 ?0 is of the form: $\ulcorner_Z \exists\ D \mid P_1 \bullet\ P_2 \urcorner$, *but D non−trivial*

SML

```
val z_∃_elim_conv2 : CONV;
val z_∃_intro_conv1 : CONV;
```

Description *z_∃_elim_conv2* turns a Z existentially quantified predicate into a HOL existentially quantified term. The result fails to be in the Z language because it contains a declaration used in a position that requires a predicate, which Z does not allow, as well as the outer HOL existential quantification.

Conversion

$$\frac{z\_\exists\_elim\_conv2 \quad ⓩ\exists\ D[x_1,...] \mid P_1 \bullet P_2⌝}{\vdash (\exists\ D[x_1,...] \mid P_1 \bullet P_2) \Leftrightarrow ⌜\exists\ x_1\ ... \bullet ⓩD[x_1,...]⌝ \land ⓩP_1⌝ \land ⓩP_2⌝⌝}$$

The order of the resulting existentially quantified variables will be in a sorted order, rather than what the declaration part might suggest.

z_∃_intro_conv1 undoes this process, returning a theorem whose RHS should be in the Z language (though this is not checked for).

Conversion

$$\frac{z\_\exists\_intro\_conv1 \quad ⌜\exists\ x_1\ ... \bullet ⓩD[x_1,...] \land P_1 \land P_2⌝⌝}{\vdash ⌜\exists\ x_1\ ... \bullet ⓩD[x_1,...] \land P_1 \land P_2⌝⌝ \Leftrightarrow (\exists\ D[x_1,...] \mid P_1 \bullet P_2)}$$

If there are no quantified variables and the declaration is $D[]$, the HOL existential quantification is not generated by *z_∃_elim_conv2* nor expected by *z_∃_intro_conv1*.

Uses Used in the Z form of *strip_tac*, and handling negations with quantifiers.

See Also *z_∃_elim_conv1*, *z_∃_elim_conv* and *z_∃_intro_conv*

Errors

41044 ?0 is not of the form: $ⓩ\exists\ D \mid P_1 \bullet P_2⌝$
41045 ?0 is not of the form: $⌜\exists\ x_1\ ... \bullet D[x_1,...] \land P_1 \land P_2⌝$
41041 ?0 is not of the form: $⌜\forall\ x_1\ ... \bullet D \land P_1 \Rightarrow P_2⌝$
where the $⌜x_i⌝$ *do not match the declaration*

SML

val **z_∃_elim_conv** : *CONV*;

Description Turn a Z existentially quantified predicate into a HOL existentially quantified term, converting the declaration part of the original quantification into its implicit predicate.

Conversion

$$\frac{}{\vdash (\exists\ D[x_1,...] \mid P_1 \bullet\ P_2) \Leftrightarrow \ulcorner \exists\ x_1\ ...\bullet\ \ulcorner_Z\text{"predicate from } D[x_1,...]\text{"} \land P_1 \land P_2 \urcorner\urcorner}$$

z_∃_elim_conv1
$\ulcorner_Z \exists\ D[x_1,...] \mid P_1 \bullet\ P_2 \urcorner$

The order of the resulting existentially quantified variables will be in a sorted order, rather than what the declaration part might suggest.

If there are no quantified variables and the declaration is $D[]$, the HOL existential quantification is not generated.

The result fails to be within the Z language, but only due to the outer HOL existential quantification.

See Also *z_∃_elim_conv2*, *z_∃_elim_conv1*

Errors

41042 ?0 is not of the form: $\ulcorner_Z \exists\ D \mid P_1 \bullet\ P_2 \urcorner$

SML

val **z_∃_intro_conv** : *CONV*;

Description *z_∃_intro_conv* converts an arbitrary simple HOL existentially quantified term into the corresponding Z, returning a theorem whose RHS should be in the Z language (though this is not checked for).

Conversion

$$\frac{}{\vdash \ulcorner \exists\ x_1\ ...\bullet\ P \urcorner \Leftrightarrow (\exists\ x_1{:}\mathbb{U};\ ...\ \mid true \bullet\ P)}$$

z_∃_intro_conv
$\ulcorner \exists\ x_1\ ...\bullet\ P \urcorner$

This conversion cannot introduce a Z existential quantification with an empty declaration.

See Also *z_∃_intro_conv1*

Errors

41046 ?0 is not of the form: $\ulcorner \exists\ x_1\ ...\ \bullet\ P \urcorner$

SML

```
val z_∃_inv_conv : CONV;
```

Description Simplifies a Z existential quantification whose predicate or constraint is invariant w.r.t. the free variables bound by the declaration.

Conversion

$$\frac{}{\vdash \exists\ D \mid P_1 \bullet\ P_2 \Leftrightarrow (\exists\ D \mid P_1 \bullet\ true) \land P_2} \quad z\_\exists\_inv\_conv\ ⓩ\exists\ D \mid P_1 \bullet\ P_2⌝$$

if P_2 has no free variables bound by D, and

Conversion

$$\frac{}{\vdash \exists\ D \mid P_1 \bullet\ P_2 \Leftrightarrow P_1 \land (\exists\ D \mid true \bullet\ P_2)} \quad z\_\exists\_inv\_conv\ ⓩ\exists\ D \mid P_1 \bullet\ P_2⌝$$

if P_1 has no free variables bound by D, and

Conversion

$$\frac{}{\vdash \exists\ D \mid P_1 \bullet\ P_2 \Leftrightarrow P_1 \land (\exists\ D \mid true \bullet\ true) \land P_2} \quad z\_\exists\_inv\_conv\ ⓩ\exists\ D \mid P_1 \bullet\ P_2⌝$$

if both have no free variables bound by D.

P_1 nor P_2 will be "extracted" if identical to *true*.

See Also *z_∀_inv_conv*

Errors

47290 ?0 is not a Z existential quantification
41040 ?0 is not of the form: ⓩ∃ D | P_1• P_2⌝ where at least one of P_1 or P_2 are unbound by D

SML

```
val z_∃_tac : TERM -> TACTIC ;
```

Description Given a binding of identifiers to witnesses, accept this as a "group witness" for a Z existentially quantified goal.

Tactic

$$\frac{\{\ \Gamma\ \}\ \exists\ D[x_1,...]\ |\ P_1[x_1,...] \bullet\ P_2[x_1,...]}{\{\ \Gamma\ \}\ \text{"predicate from } D[t'_1,...]\text{"} \land P'_1[t'_1,...] \land P'_2[t'_1,...]} \qquad z\_\exists\_tac\ \ulcorner_Z(x_1 \mathrel{\widehat{=}} t_1,...)\urcorner$$

where the t'_i are appropriately type instantiated forms of the t_i. Renaming of bound variables may be necessary, thus P'_1 rather than P_1, etc. See *z_decl_pred_conv* for the conversion of a declaration to its implicit predicate.

An empty declaration may be given an empty binding.

If the goal's conclusion has a single Z existentially bound variable, and the supplied argument can be type instantiated to match that, then it will be used as a witness.

Tactic

$$\frac{\{\ \Gamma\ \}\ \exists\ x{:}X\ |\ P_1[x_1] \bullet\ P_2[x_1]}{\{\ \Gamma\ \}\ t' \in X \land P'_1[t'] \land P'_2[t']} \qquad z\_\exists\_tac\ \ulcorner_Z t\urcorner$$

where t' is an appropriately type instantiated form of the t.

Finally, if none of the above apply, the supplied binding may instead be anything else that can be type instantiated to the appropriate binding type. In such cases projection functions will be used:

Tactic

$$\frac{\{\ \Gamma\ \}\ \exists\ D[x_1,...]\ |\ P_1[x_1,...] \bullet\ P_2[x_1,...]}{\{\ \Gamma\ \}\ \text{"predicate from } D[t.x'_1,...]\text{"} \land P'_1[t'.x_1,...] \land P'_2[t'.x_1,...]} \qquad z\_\exists\_tac\ \ulcorner_Z t\urcorner$$

where t' is an appropriately type instantiated form of the t.

Errors

47290 ?0 is not a Z existential quantification
41021 ?0 cannot be interpreted as a binding that matches ?1

SML

```
val z_∃₁_conv : CONV;
```

Description Converts a Z unique existential quantification to a Z existential quantification.

Conversion

$$\frac{}{\begin{array}{c}\vdash (\exists_1\ D \mid P_1 \bullet\ P_2) \\ \Leftrightarrow \\ (\exists\ D \mid P_1 \bullet\ P_2 \land \\ (\forall\ D' \mid P_1' \land P_2' \bullet \\ \text{"characteristic tuples} \\ \text{component-wise equal"}))\end{array}} \quad \begin{array}{l} z\_\exists_1\_conv \\ ⓩ\exists_1\ D \mid P_1 \bullet\ P_2 ⌝\end{array}$$

where the P_i' are variants of the P_i, to correspond to the difference between D and D'.

Additional decoration may be introduced as necessary to avoid free variable names capture, while maintaining the same decoration on each component (variable, schema, etc) of the declaration.

Example

```
z_∃₁_conv ⓩ ∃₁ [x,y : X; z:Y] | x = x′ y• z = f x⌝;
  -->
⊢ (∃₁ [x, y : X; z : Y | true] | x = x′ y• z = f x)
  ⇔ (∃ [x, y : X; z : Y | true]
    | (x = x′ y) ∧ (z = f x)
• ∀ [x, y : X; z : Y | true]″
      | (x″ = x′ y″) ∧ (z″ = f x″)
• (x″ = x) ∧ (y″ = y) ∧ (z″ = z))
```

See Also *z_∃₁_intro_conv*

Errors

```
41122 ?0 is not of the form: ⓩ∃₁D | P₁• P₂⌝
```

SML

```
val z_∃₁_intro_conv : CONV;
```

Description *z_∃₁_intro_conv* converts an arbitrary simple HOL unique existentially quantified term into the corresponding Z, returning a theorem whose RHS should be in the Z language.

It can only reason about a single bound variable.

Conversion

$$\frac{}{\begin{array}{c}\vdash ⌜\exists_1\ x \bullet\ P\ [x]⌝ \Leftrightarrow \\ (\exists_1\ x{:}\mathbb{U} \mid true \bullet\ P[x]\ )\end{array}} \quad \begin{array}{l} z\_\exists_1\_intro\_conv \\ ⌜\exists_1\ x \bullet\ P[x]⌝\end{array}$$

This conversion cannot introduce a Z unique existential quantification with an empty declaration.

See Also *z_∀_intro_conv1*

Errors

```
41048 ?0 is not of the form: ⌜∃₁ x₁ • P⌝
```

SML

$\vert$ *val* **z_∃₁_tac** : *TERM −> TACTIC*;

Description Provide a witness for a unique existential.

Tactic

$$\frac{\{\ \Gamma\ \}\ \exists_1\ D[x_1,...]\ |\ P[x_1,...]\ \bullet\ V[x_1,...]}{\{\ \Gamma\ \}\ \text{"}D[t_1,...]\ as\ a\ predicate\text{"}\ \wedge\ (\forall\ D'\ |\ P[x'_1,...]\ \wedge\ V[x'_1,...]\bullet\ \text{"}characteristic\ tuples\ of\ D'\ and\ witness\ component{-}wise\ equal\text{"}))}\quad z\_\exists_1\_tac\ \ulcorner_Z(x_1\ \hat{=}\ t1,...)\urcorner$$

Definition

$\vert$ *val z_∃₁_tac wit = conv_tac z_∃₁_conv THEN z_∃_tac wit*;

Errors

$\vert$ *41123 Conclusion of goal is not of the form:* $\ulcorner_Z \exists_1 D\ |\ P_1 \bullet\ P_2\urcorner$
$\vert$ *41021 ?0 cannot be interpreted as a binding that matches ?1*

SML

$\vert$ *val* **α_to_z_conv** : *CONV*;

Description This function will return the equality theorem between a term and one that adjusts all sub-terms that fail to be Z because either:

- The subterm is a *decl*-style binding, and the type of the binding does not match the names of the variables bound. This is adjusted using the Z renaming construct.
- The subterm is a *decl*-style binding whose bound variables are not in the canonical ordering that would result from the Z mapping. This is adjusted by reorganising the order of abstractions and arguments.

If a HOL α-conversion will suffice then that will be used instead.

Subterms that are not covered by these two cases will be traversed and their own subterms checked, regardless of their language.

NOT YET IMPLEMENTED.

See Also *α_to_z*

Errors

$\vert$ *41100 No adjustment took place*

SML

```
val α_to_z : TERM -> TERM;
```

Description This function will adjust all sub-terms that fail to be Z because either:

- The subterm involves a *decl*-style binding, and the type of the binding does not match the names of the variables bound. This is adjusted using the Z renaming construct.
- The subterm is a *decl*-style binding whose bound variables are not in the canonical ordering that would result from the Z mapping. This is adjusted by reorganising the order of abstractions and arguments.

If a HOL α-conversion will suffice then that will be used instead.

Subterms that are not covered by these two cases will be traversed and their own subterms checked, regardless of their language.

NOT YET IMPLEMENTED.

See Also *α_to_z_conv*

Errors

```
41100 No adjustment took place
```

8.3 Reasoning about Expressions

SML

signature **ZExpressions** = *sig*

Description This provides the rules of inference, conversions and theorems for Z language set theory, tuples and cartesian products in the Z proof support system.

SML

(∗ *Proof Context*: **′z_∈_set_lang** ∗)

Description A component proof context for handling the membership of expressions created by Z language set operations. It also provides some simplifications.

Set expressions treated by this proof context are constructs formed from:

set displays, *set comprehensions*, ℙ, λ, μ, *application*,
sequence displays

If there was proof context material for string literals, or bag displays, it would perhaps go here.

Contents

Rewriting:

z_∈_seta_conv1, *z_∈_setd_conv1*, *z_∈_λ_conv*, *z_∈_⟨⟩_conv*,
z_β_conv if its resulting theorem has no assumptions.

Stripping theorems:

z_∈_seta_conv1, *z_∈_setd_conv1*, *z_∈_λ_conv*, *z_∈_⟨⟩_conv*,
plus these all pushed in through ¬,
and z_β_conv, *∈_C z_β_conv if the resulting theorem has no assumptions*.

Stripping conclusions:

z_∈_seta_conv1, *z_∈_setd_conv1*, *z_∈_λ_conv*, *z_∈_⟨⟩_conv*,
plus these all pushed in through ¬,
and z_β_conv, *∈_C z_β_conv if the resulting theorem has no assumptions*.

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_sets*. It is intended to be used with proof context "z_predicates". It is not intended to be mixed with HOL proof contexts.

See Also *′z_sets_ext*

SML

|(∗ *Proof Context*: **′z_sets_ext_lang** ∗)

Description An aggressive component proof context for handling the manipulation of Z sets by breaking them into predicate calculus, within the Z language. It is intended to always be used in conjunction with "′z_set_lib".

Set expressions treated by this proof context are constructs formed from:

|*set displays*, *set comprehensions*, ℙ, λ, μ, *application*,
|*equality of two set expressions*, *sequence displays*

Contents

Rewriting:

|*z_sets_ext_conv*, *z_∈_ℙ_conv*, *z_setd_∈_ℙ_conv*,

Stripping theorems:

|*z_sets_ext_conv*, *z_∈_ℙ_conv*, *z_setd_∈_ℙ_conv*,
|*plus these all pushed in through* ¬

Stripping conclusions:

|*z_sets_ext_conv*, *z_∈_ℙ_conv*, *z_setd_∈_ℙ_conv*,
|*plus these all pushed in through* ¬

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover (2-tuples are handled in proof context "z_predicates").

Usage Notes It requires theory *z_sets*. It is intended to always be used in conjunction with "′z_set_lang".

It is not intended to be mixed with HOL proof contexts.

See Also *′z_∈_set*

SML

(∗ *Proof Context*: **′z_tuples_lang** ∗)

Description A component proof context for handling the manipulation of Z tuples and cartesian products within the Z language.

Expressions and predicates treated by this proof context are constructs formed from:

(membership of) ×, *equations of tuple displays*,
selection from tuple displays

Contents

Rewriting:

$z\_\in\_\times\_conv$,
$z\_tuple\_lang\_eq\_conv$, $z\_sel_t\_lang\_conv$

Stripping theorems:

$z\_\in\_\times\_conv$, $z\_tuple\_lang\_eq\_conv$, $\in\_C$ $z\_sel_t\_lang\_conv$,
$z\_sel_t\_lang\_conv$ *(for boolean compoents of tuples)*
plus these all pushed in through ¬

Stripping conclusions:

$z\_\in\_\times\_conv$, $z\_tuple\_lang\_eq\_conv$, $\in\_C$ $z\_sel_t\_lang\_conv$,
$z\_sel_t\_lang\_conv$ *(for boolean compoents of tuples)*
plus these all pushed in through ¬

Stripping also contains the above in negated forms.

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *_basic_prove_conv*, and no existence prover (2-tuples are handled in proof context "z_predicates").

Usage Notes It requires theory *z_sets*. It is intended to be used with proof context "z_predicates". It should not be used with "′z_tuples_lang". It is not intended to be mixed with HOL proof contexts.

SML

(∗ *Proof Context*: ′**z_bindings** ∗)

Description A component proof context for handling the manipulation of Z bindings.

Expressions and predicates treated by this proof context are constructs formed from:

equations of binding displays,
selection from binding displays

Contents

Rewriting:

$z\_binding\_eq\_conv2$, $z\_sel_s\_conv$

Stripping theorems:

$z\_binding\_eq\_conv2$, $\in\_C\ z\_sel_s\_conv$,
$z\_sel_s\_conv$ (*where component of binding is boolean*).
plus this pushed in through ¬

Stripping conclusions:

$z\_binding\_eq\_conv2$, $\in\_C\ z\_sel_s\_conv$,
$z\_sel_s\_conv$ (*where component of binding is boolean*).
plus this pushed in through ¬

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_language_ps*. It is intended to be used with proof context "z_predicates". It is not intended to be mixed with HOL proof contexts.

SML

```
val z_sets_ext_thm : THM;
val z_∈_ℙ_thm1 : THM;
val z_app_thm : THM;
val z_app_∈_thm : THM;
val z_∈_app_thm : THM;
```

Description The ML bindings of the theorems (other than consistency ones) in theory *z_language_ps*.

SML

```
val z_app_conv : CONV;
```

Description A function to convert a Z application into the corresponding μ expression (i.e. definite description).

Conversion

$$\frac{}{\vdash f\ a = (\mu\ f\_a : \mathbb{U} \mid (a,f\_a) \in f \bullet f\_a)} \quad z\_app\_conv \ ⌜_z f\ a⌝$$

Remains within the Z language though this is not checked.

See Also *z_app_thm*, *z_app_eq_tac*

Errors

47190 ?0 is not a Z function application

SML

val **z_app_eq_tac** : *TACTIC*;

Description Reduces a subgoal that states a Z application is equal to something to sufficient conditions for this to be provable. The conditions are not "necessary" only because they ignore the fact that in ProofPower-Z every predicate or expression is equal to itself, and other vacuous variants of this.

Tactic

$$\frac{\{\Gamma\}\ f\ \ a = v}{\{\Gamma\}\ (\forall\ f\_a : \mathbb{U} \mid (a,\ f\_a) \in f \bullet\ f\_a = v) \\ \wedge\ (a,v) \in f}\qquad z\_app\_eq\_tac$$

If this does not match the pattern of the goal then

Tactic

$$\frac{\{\Gamma\}\ v = f\ \ a}{\{\Gamma\}\ (\forall\ f\_a : \mathbb{U} \mid (a,\ f\_a) \in f \bullet\ f\_a = v) \\ \wedge\ (a,v) \in f}\qquad z\_app\_eq\_tac$$

will be tried instead. In addition an implicit "$\Leftrightarrow$ *true*" will be used if the conclusion of the goal is an application.

See Also *z_app_thm*, *z_app_conv*

Errors

42002 Conclusion of goal is not of the form: $⌜_Z f\ \ a = v⌝$, $⌜_Z v = f\ \ a⌝$ *or* $⌜_Z f\ \ x⌝$

SML

val **z_app_λ_rule** : *TERM −> THM*;

Description Given a Z β redex this function will return a theorem stating sufficient conditions for this redex to be proven equal to some arbitrary value.

Rule

$$\frac{}{\vdash ⌜_Z \forall\ x{:}\mathbb{U} \bullet (\forall\ f\_a : \mathbb{U} \mid (\exists\ D' \mid P' \bullet \\ charD' = t \wedge V' = f\_a) \bullet f\_a = x) \\ \wedge \\ (\exists\ D' \mid P' \bullet (charD' = t) \wedge V' = x) \\ \Rightarrow \\ (\lambda\ D \mid P \bullet\ V)\ t = x⌝}\qquad \begin{array}{l} z\_app\_\lambda\_rule \\ ⌜_Z(\lambda\ D \mid P \bullet\ V)\ t⌝ \end{array}$$

Some renaming of bound variables may occur, thus the priming of D, etc.

Errors

42008 ?0 is not of the form: $⌜_Z(\lambda\ D \mid P \bullet\ V)\ t⌝$

SML

val **z_bindingd_elim_conv** : *CONV*

Description Given a a binding display, that binds labels to the selection of that label to a single value, return that single value.

Conversion

$$\frac{}{\vdash (x_1 \widehat{=} b.x_1,\ ...) = b}\qquad \begin{array}{l} z\_bindingd\_elim\_conv \\ ⌜_Z(x_1 \widehat{=} b.x_1,\ ...)⌝ \end{array}$$

Errors

42018 ?0 is not of the form: $⌜_Z(x_1 \widehat{=} b.x_1,\ ...,\ x_N \widehat{=} b.x_N)⌝$ *where* $N \geq 1$

SML

val **z_bindingd_intro_conv** : *CONV*

Description Given a value with a binding type, prove it equal to a binding display.

Conversion

$$\frac{}{\vdash b = (x_1 \widehat{=} b.x_1, ...)} \quad z\_bindingd\_intro\_conv \ ⌜_Z b⌝$$

Errors

42017 ?0 does not have a binding type

SML

val **z_binding_eq_conv** : *CONV*;
val **z_binding_eq_conv1** : *CONV*;
val **z_binding_eq_conv2** : *CONV*;

Description A conversion for eliminating equations of bindings.

Conversion

$$\frac{}{\vdash (b_1 = b_2) \Leftrightarrow (b_1.s_1 = b_2.s_1) \land (b_1.s_2 = b_2.s_2) \land ...} \quad z\_binding\_eq\_conv \ ⌜_Z b_1 = b_2⌝$$

where b_1 (and thus b_2) has a binding type equal to the type of something of the form ⌜$_Z(s_1 \widehat{=} ..., s_2 \widehat{=} ..., ...)$⌝.

z_binding_eq_conv1 first applies conversion *z_binding_eq_conv*, and then, if either or both of b_1 and b_2 are binding constructions it eliminates the projection functions, in a manner similar to $z\_sel_s\_conv$.

z_binding_eq_conv2 requires both sides to be binding displays or have the empty schema type:

Conversion

$$\frac{}{\vdash ((l_1 \widehat{=} x_1,...) = (l_1 \widehat{=} y_1,...)) \Leftrightarrow (x_1 = y_2) \land ...} \quad z\_binding\_eq\_conv2 \ ⌜_Z(l_1 \widehat{=} x_1,...) = (l_1 \widehat{=} y_1,...)⌝$$

Conversion

$$\frac{}{\vdash ((b_1 \oplus\!\!\!\oplus []) = b_2) \Leftrightarrow true} \quad z\_binding\_eq\_conv2 \ ⌜_Z(b_1 \oplus\!\!\!\oplus []) = b_2⌝$$

See Also $z\_sel_s\_conv$, *z_binding_eq_conv3*

Errors

42013 ?0 is not of the form ⌜$_Z$*binding = binding*⌝
42021 ?0 is not of the form ⌜$_Z b_1 = b_2$⌝ *where* b_i *has the form* ⌜$_Z(x_1 \widehat{=} ..., ...)$⌝ *or has the empty schema*

SML

val **z_defn_simp_rule** : *THM* −> *THM*;

Description This rule is a method of processing a standard style of specification into a simple rewriting theorem.

Rule

$$\frac{\vdash x \in (\mathbb{P}\ y) \land (\forall z{:}y \bullet\ z \in x \Leftrightarrow f[z])}{\vdash \forall\ z{:}\ \mathbb{U} \bullet\ z \in x \Leftrightarrow z \in y \land f[z]} \quad z\_defn\_simp\_rule$$

The rule will also attempt to preprocess its input with *z_para_pred_conv*. This is on the basis that theorems that are of an appropriate form for this rule are often derived from a Z definition, and this pre-processing is all the processing required to convert the definition to acceptable input. The rule can also handle generic parameters to the theorem.

Errors

42011 ?0 cannot be converted to the form:
'Γ ⊢ x ∈ (ℙ y) ∧ (∀z:y• z ∈ x ⇔ f[z])'

SML

val **z_let_conv1** : *CONV*;

Description This conversion replaces a let-expression by an equivalent μ-expression.

Rule

$$\frac{}{\vdash (let\ v1\ \widehat{=}\ t1;\ ...\ \bullet\ b) = (\mu\ v1\ :\ \{t1\};\ ...\bullet\ b)} \quad \begin{array}{l} z\_let\_conv1 \\ \ulcorner_Z let\ v1\ \widehat{=}\ t1;\ ...\ \bullet\ b\urcorner \end{array}$$

This is mainly intended for use in programming proof procedures. *z_let_conv* may be used simply to expand let-expressions

See Also *z_let_conv*

Errors

47211 ?0 is not a Z let term

SML

val **z_let_conv** : *CONV*;

Description This conversion expands the local definitions in a let-expression.

Rule

$$\frac{}{\vdash (let\ v1\ \widehat{=}\ t1;\ ...\ \bullet\ b) = b[t1/v1,\ ...]} \quad \begin{array}{l} z\_let\_conv \\ \ulcorner_Z let\ v1\ \widehat{=}\ t1;\ ...\ \bullet\ b\urcorner \end{array}$$

The conversion will fail with message 42029 given a let-expression such as $\ulcorner_Z let\ x\ \widehat{=}\ 42;\ y\ \widehat{=}\ 99;\ x\ \widehat{=}\ 43\ \bullet\ x + y\urcorner$ that contains incompatible local definitions.

See Also *z_let_conv1*

Errors

47211 ?0 is not a Z let term
42029 The local definitions in the let−expression ?0 cannot be expanded

SML

val **Z_RAND_C** : *CONV* −> *CONV*;
val **Z_RANDS_C** : *CONV* −> *CONV*;
val **Z_LEFT_C** : *CONV* −> *CONV*;
val **Z_RIGHT_C** : *CONV* −> *CONV*;

Description *Z_RAND_C* (resp. *Z_RANDS_C*, *Z_LEFT_C*, *Z_RIGHT_C*) applies a conversional to the operand (resp. operands, left operand, right operand) of a Z function application.

SML

val **z_sel$_s$_conv** : *CONV*;

Description A conversion for selecting a component from a binding.

Conversion

$$\frac{}{\vdash (n_1 \mathrel{\widehat{=}} t_1,...).n_c = t_c}$$

z_sel$_s$_conv ⌜$_Z$ $(n_1 \mathrel{\widehat{=}} t_1,...).n_c$⌝

See Also *z_binding_eq_conv*

Errors

42014 ?0 is not of the form: ⌜$_Z$ $(n_1 \mathrel{\widehat{=}} t_1,...).n_c$⌝

SML

val **z_sel$_t$_intro_conv** : *CONV*;

Description This conversion carries out the introduction of a tuple display of tuple selections from the same tuple.

Conversion

$$\frac{}{\vdash t = (t.1,...,t.n)}$$

z_sel$_t$_intro_conv ⌜$_Z$ t⌝

Errors

42004 ?0 does not have a Z tuple type

SML

val **z_sel$_t$_lang_conv** : *CONV*;

Description This conversion carries out the selection from a tuple display.

Conversion

$$\frac{}{\vdash (t_1,...,t_i,...,t_n).i = t_i}$$

z_sel$_t$_lang_conv ⌜$_Z$ $(t_1,...,t_i,...,t_n).i$⌝

Errors

47185 ?0 is not a Z tuple selection
42006 ?0 is not of the form ⌜$_Z$ $(x,...).i$⌝

SML

val **z_setd_∈_ℙ_conv** : *CONV*

Description Expand out expressions that state that a set display is a member of a power set. .

Conversion

$$\frac{}{⌜_Z \{x_1,...\} \in \mathbb{P}\ X \Leftrightarrow (x_1 \in X \land ...)}$$

z_setd_∈_ℙ_conv ⌜$_Z$ $\{x_1,...\} \in \mathbb{P}\ X$⌝

and

Conversion

$$\frac{}{⌜_Z \{\} \in \mathbb{P}\ X \Leftrightarrow true}$$

z_setd_∈_ℙ_conv ⌜$_Z$ $\{\} \in \mathbb{P}\ X$⌝

The conversion will all simplify certain subterms involving *true* or terms of the form $x = x$.

See Also *z_setd_⊆_conv*

Errors

42019 ?0 is not of the form: ⌜$_Z$ $\{x_1,...\} \in \mathbb{P}\ X$⌝

SML
```
val z_sets_ext_conv : CONV;
```

Description Use the extensionality of sets in combination with knowledge about tuples. Given as input an equality of the form $v = w$ then:

If v is of type *ty SET* where *ty* is not a tuple type:

Conversion

$$\frac{}{\vdash (v = w) \Leftrightarrow (\forall\ xn : \mathbb{U} \bullet xn \in v \Leftrightarrow xn \in w)} \quad z\_sets\_ext\_conv \ ⓩv = w⌝$$

where xn is the first variable in the list $x1$, $x2$,... that doesn't appear in v or w (free or bound).

If w is of type *ty SET* where *ty* is an n-tuple type, or binding type, then sufficient x_i will be introduced, instead of just xn, to allow xn to be replaced by a construct of bindings and tuples of the x_i, such that no x_i has a binding or tuple type and appears exactly once in the construct.

Example
```
z_sets_ext_conv ⓩ(r × [a,b:X] × x2) = d⌝ =
⊢ r × [a, b : X] × x2 = d
    ⇔ (∀ x1 : 𝕌; x3 : 𝕌; x4 : 𝕌
      • (x1, x3, x4) ∈ r × [a, b : X] × x2 ⇔ (x1, x3, x4) ∈ d)
```

Notice how the introduced universal quantification "skips" $x2$ which is present in the input term.

See Also *z_sets_ext_thm*

Errors
```
42010 ?0 is not of the form: ⓩv = w⌝ where ⓩv⌝ has a set type
```

SML
```
val z_string_conv : CONV;
val z_∈_string_conv : CONV;
```

Description *z_string_conv* changes a Z string into a Z sequence of HOL characters. It thus does not remain in Z.

Conversion

$$\frac{}{\vdash \texttt{"}abc...\texttt{"} = \langle ⌜`a`⌝,⌜`b`⌝,⌜`c`⌝,...\rangle} \quad z\_string\_conv \ ⓩ\texttt{"}abc...\texttt{"}⌝$$

Definition
```
val z_∈_string_conv = ∈_C z_string_conv;
```

See Also *char_eq_conv* for the equality of HOL characters, *z_string_eq_conv* for the equality of Z strings.

Errors
```
42015 ?0 is not of the form: ⓩ"abc..."⌝
```

SML
val **z_tuple_lang_eq_conv** : *CONV*;

Description A conversion for eliminating tuples over equality.

Conversion

$$\frac{}{\vdash (t_1,t_2,...) = (u_1,u_2,...) \Leftrightarrow ((t_1 = u_1) \wedge (t_2 = u_2) \wedge ...)}$$

z_tuple_lang_eq_conv
$\ulcorner_Z (t_1,t_2,...) = (u_1,u_2,...)\urcorner$

Errors
42003 ?0 is not of the form: $\ulcorner_Z (x1,...) = (y1,...)\urcorner$

SML
val **z_tuple_lang_intro_conv** : *CONV*;

Description This conversion carries out the elimination of a tuple display of tuple selections from the same tuple.

Conversion

$$\frac{}{\vdash (t.1,...,t.n) = t}$$

z_tuple_lang_intro_conv
$\ulcorner_Z (t.1,...,t.n)\urcorner$

where n is the arity of t.

Errors
42005 ?0 is not of the form: $\ulcorner_Z (t.1,...,t.n)\urcorner$

SML
val **Z_∈_ELIM_C** : *CONV −> CONV*;

Description *Z_∈_ELIM_C cnv tm* takes a conversion *cnv* that can be applied to set memberships, and a set term *tm*. The conversion is then modified to make it applicable to the term. The resulting conversion will check to see if its term argument, *tm* is a set. If so it will form the term: $\ulcorner_Z xi \in {}_{ML}\ulcorner tm\urcorner\urcorner$(where *xi* is the first variable in *x1*, *x2*,... not present in *tm*), apply *cnv* to the result, gaining some equation:

$\vdash xi \in {}_{ML}\ulcorner tm\urcorner \Leftrightarrow f[xi]$

and then return the theorem

$\vdash {}_{ML}\ulcorner tm\urcorner = \{xi : \mathbb{U} \mid f[xi]\}$

Errors
42027 ?0 is not a Z set
42026 unable to convert ?0 to the form: $\ulcorner_Z x \in \{x{:}\mathbb{U}|s\}\urcorner$

And as conversion argument upon the membership term, with the error being passed through by the conversional untouched.

SML

```
val z_∈_seta_conv : CONV;
val z_∈_seta_conv1 : CONV;
```

Description A conversion of membership of a Z set abstraction into a Z existential quantification. Bound variables in the existential quantification are renamed as necessary.

Conversion

$$\frac{}{\vdash (t \in \{\ D \mid P \bullet\ T\}) \Leftrightarrow (\exists\ D' \mid P' \bullet\ T' = t)} \quad z\_{\in}\_seta\_conv \ ⓩt \in \{\ D \mid P \bullet\ T\}⌝$$

In the case of *z_∈_seta_conv1*, if T is a tuple or simple variable then the conversion will attempt to eliminate the existential quantification via the methods of *basic_prove_∃_conv*. In particular, this attempt should succeed if T is the characteristic tuple of D.

No simplification is attempted by *z_∈_seta_conv*

Renaming of bound variables may be necessary, thus the priming in the RHS.

Errors

42001 ?0 is not of the form: ⓩt ∈ seta⌝ where seta is a set abstraction

SML

```
val z_∈_setd_conv1 : CONV;
```

Description A conversion proving membership of a Z set display where the member is syntactically identical (up to α-conversion) to a member of the set.

Conversion

$$\frac{}{\vdash t \in \{\ ...,\ t,\ ...\} \Leftrightarrow true} \quad z\_{\in}\_setd\_conv1 \ ⓩt \in \{\ ...,\ t,\ ...\}⌝$$

See Also *z_∈_setd_conv*

Errors

42009 ?0 is not of the form: ⓩt ∈ {...,t,...}⌝

SML

```
val z_∈_×_conv : CONV;
```

Description A conversion for the membership of cartesian products.

Conversion

$$\frac{}{\vdash t \in (T_1 \times T_2 \times ...) \Leftrightarrow t.1 \in T_1 \land t.2 \in T_2 \land ...} \quad z\_{\in}\_{\times}\_conv \ ⓩt \in (T_1 \times T_2 \times ...)⌝$$

z_sel$_t$*_conv*, q.v., will be attempted on each of the tuple selections.

See Also *z_×_conv*

Errors

42007 ?0 is not of the form: ⓩ$t \in (T_1 \times T_2 \times ...)$⌝

SML

val **z_∈_ℙ_conv** : $CONV$;

Description Use $z\_\in\_\mathbb{P}\_thm1$ in combination with knowledge about tuples. Given as input a term of the form $v \in\mathbb{P}\ w$ then:

If w is of type $ty\ SET$ where ty is not a tuple type:

Conversion

$$\frac{}{\vdash (v \in \mathbb{P}\ w) \Leftrightarrow (\forall\ xn : \mathbb{U} \bullet xn \in v \Rightarrow xn \in w)} \quad \begin{array}{l} z\_\in\_\mathbb{P}\_conv \\ ⓩv \in \mathbb{P}\ w⌝ \end{array}$$

where xn is the first variable in the list $x1$, $x2$,... that doesn't appear in v or w (free or bound).

If w is of type $ty\ SET$ where ty is an n-tuple type, or binding type, then sufficient x_i will be introduced, instead of just xn, to allow xn to be replaced by a construct of bindings and tuples of the x_i, such that no x_i has a binding or tuple type and appears exactly once in the construct.

Example

$z\_\in\_\mathbb{P}\_conv$ ⓩ$p \in \mathbb{P}\ (r \times [a,b{:}X] \times x2)$⌝ =
$\vdash p \in \mathbb{P}\ (r \times [a,\ b : X] \times x2)$
$\quad \Leftrightarrow (\forall\ x1 : \mathbb{U};\ x3 : \mathbb{U};\ x4 : \mathbb{U};\ x5 : \mathbb{U}$
$\qquad \bullet (x1,\ (a \mathrel{\widehat{=}} x3,\ b \mathrel{\widehat{=}} x4),\ x5) \in p$
$\qquad\quad \Rightarrow (x1,\ (a \mathrel{\widehat{=}} x3,\ b \mathrel{\widehat{=}} x4),\ x5) \in r \times [a,\ b : X] \times x2)$

Notice how the introduced universal quantification "skips" $x2$ which is present in the input term.

See Also $z\_\in\_\mathbb{P}\_thm1$, $z\_\in\_\mathbb{P}\_thm$, $z\_\subseteq\_conv$

Errors

42016 ?0 is not of the form ⓩ$v \in \mathbb{P}\ w$⌝

SML

val **z_⟨⟩_conv** :$CONV$;
val **z_∈_⟨⟩_conv** :$CONV$;

Description Convert a sequence display into a set display.

Conversion

$$\frac{}{\vdash \langle x1,...,xn\rangle = \{(1,x1),...,(n,xn)\}} \quad \begin{array}{l} z\_\langle\rangle\_conv \\ ⓩ\langle x1,...,xn\rangle⌝ \end{array}$$

Definition

$val\ z\_\in\_\langle\rangle\_conv = \in\_C\ z\_\langle\rangle\_conv;$

Errors

42025 ?0 is not of the form: ⓩ⟨ ... ⟩⌝

SML

$val\ \mathbf{z\_\times\_conv} : CONV;$

Description A conversion for eliminating cartesian products.

Conversion

$$\frac{}{\vdash (T_1 \times T_2 \times ...) = \{t_1{:}T_1;\ t_2{:}T_2;\ ...\bullet(t_1,\ t_2,...)\}} \quad \begin{array}{l} z\_\times\_conv \\ \ulcorner_Z T_1 \times T_2 \times ...\urcorner \end{array}$$

The t_i used are distinct from any variable names in the T_i.

See Also $z\_\in\_\times\_conv$, which is a faster function, if appropriate.

Errors

47160 ?0 is not a Z cartesian product

SML

$val\ \mathbf{z\_\beta\_conv} : CONV;$

Description A conversion for a simple Z β redex. The λ-term of the redex must have only a single variable in its declaration part.

Conversion

$$\frac{}{\begin{array}{c} t \in X, \\ P[t] \\ \vdash\ (\lambda\ x{:}X \mid P[x]\bullet\ V[x])\ t = V'[t] \end{array}} \quad \begin{array}{l} z\_\beta\_conv \\ \ulcorner_Z(\lambda\ x{:}X \mid P[x]\bullet\ V[x])\ t\urcorner \end{array}$$

The assumptions will be eliminated if trivial (i.e. if the first assumption can be proven true by $z\_\in\_u\_conv$, the second if the assumption is just $\ulcorner_Z true\urcorner$). Some renaming of bound variables may occur, thus the priming of V.

Errors

42012 ?0 is not of the form $\ulcorner_Z(\lambda\ x{:}X \mid P\bullet\ V)\ t\urcorner$

SML

$val\ \mathbf{z\_\lambda\_conv} : CONV;$
$val\ \mathbf{z\_\in\_\lambda\_conv} : CONV;$

Description Convert a Z λ abstraction into a set abstraction.

Conversion

$$\frac{}{\vdash (\lambda\ D \mid P\bullet\ V) = \{\ D \mid P\bullet\ (charD,V)\}} \quad \begin{array}{l} z\_\lambda\_conv \\ \ulcorner_Z\lambda\ D \mid P\bullet\ V\urcorner \end{array}$$

Where *charD* is the characteristic tuple of *D*.

Definition

$val\ z\_\in\_\lambda\_conv = \in\_C\ z\_\lambda\_conv;$

See Also $z\_app\_\lambda\_rule$, z_β_tac

Errors

47200 ?0 is not a Z λ *abstraction*

SML

```
val z_μ_rule : TERM -> THM;
```

Description This rule is given a Z μ expression (i.e. a Z definite description), and returns a theorem that states what is required for this μ expression to be equal to some value, x. The requirement is that if any value satisfies the schema text of the μ expression then it must equal x, and that x satisfies the schema text of the μ expression.

Rule

$$\frac{}{\begin{array}{c}\vdash \forall\ x{:}\mathbb{U}\bullet \\ (\forall\ D' \mid P'\bullet\ V' = x) \land \\ (\exists\ D' \mid P'\bullet\ V' = x) \\ \Rightarrow \\ (\mu\ D \mid P\bullet\ V) = x\end{array}} \quad \begin{array}{l} z\_\mu\_rule \\ ⌜_Z \mu\ D \mid P\bullet\ V⌝ \end{array}$$

The result may require bound variable renaming and thus the priming of D, etc.

Errors

47210 ?0 is not a Z μ term

SML

```
val ∈_C : CONV -> CONV;
```

Description $\in\_C$ *cnv tm* takes a conversion *cnv*, that applies to set terms, will check to see if its term argument, *tm* is a membership statement. If so, it will apply its conversion to the set. If not it will fail. It does not check that its result remaining in Z (and indeed is applicable to HOL membership terms as well).

See Also *Z_∈_ELIM_C*

Errors

42028 ?0 is not of the form $⌜_Z v \in s⌝$ *or* $⌜v \in s⌝$

And as conversion argument upon the set, with the error being passed through by the conversional untouched.

8.4 Reasoning about Schema Expressions

SML
signature **ZSchemaCalculus** = *sig*

Description This provides the rules of inference for schema calculus in the Z proof support system. The material is implemented within the theory *z_language_ps*.

SML
(∗ *Proof Context*: ′**z_schemas** ∗)

Description A component proof context for handling the manipulation of Z schemas. It "understands" the membership, or schema as predicate, properties of each of the schema calculus operators. It will replace an appropriate ⌜Z v ∈ S⌝ by a "schema S as predicate".

Predicates and expressions treated by this proof context are constructs formed from:

(selection from) horizontal schemas, schemas as predicates, (selection from) θ expressions,
¬ₛ, ∧ₛ, ∨ₛ, ⇒ₛ, ⇔ₛ, ∀ₛ, ∃ₛ, ∃₁ₛ, decorₛ, preₛ, ↾ₛ, hideₛ, Δₛ, Ξₛ, ⨾ₛ, renameₛ,

Contents

Rewriting:

```
(RAND_C z_θ_conv THEN_C z_sel_s_conv)
        − which simplifies terms of the form: ⌜Z(θ s).nm⌝
z_θ_eq_conv, z_θ_conv1,
z_∈_¬_s_conv, z_∈_∧_s_conv, z_∈_∨_s_conv,
z_∈_⇒_s_conv, z_∈_⇔_s_conv, z_∈_∃_s_conv,
z_∈_∃1_s_conv, z_∈_∀_s_conv, z_∈_h_schema_conv,
z_∈_decor_s_conv, z_∈_pre_s_conv, z_∈_↾_s_conv,
z_∈_hide_s_conv, z_∈_Δ_s_conv, z_∈_Ξ_s_conv,
z_∈_⨾_s_conv, z_∈_rename_s_conv, z_schema_pred_intro_conv
```

Stripping theorems and conclusions:

```
(RAND_C z_θ_conv THEN_C z_sel_s_conv)
        − which simplifies boolean terms of the form: ⌜Z(θ s).nm⌝
∈_C (RAND_C z_θ_conv THEN_C z_sel_s_conv)
        − which simplifies terms of the form: ⌜Zx ∈ (θ s).nm⌝
z_θ_eq_conv, z_∈_¬_s_conv, z_∈_∧_s_conv, z_∈_∨_s_conv,
z_∈_⇒_s_conv, z_∈_⇔_s_conv, z_∈_∃_s_conv,
z_∈_∃1_s_conv, z_∈_∀_s_conv, z_∈_h_schema_conv,
z_∈_decor_s_conv, z_∈_pre_s_conv, z_∈_↾_s_conv,
z_∈_hide_s_conv, z_∈_Δ_s_conv, z_∈_Ξ_s_conv,
z_∈_⨾_s_conv, z_∈_rename_s_conv, z_schema_pred_intro_conv
plus these all pushed in through ¬
```

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_language_ps*. It is intended to be used with proof context "z_bindings". It is not intended to be mixed with HOL proof contexts.

SML

```
val z_decor_s_conv : CONV;
val z_∈_decor_s_conv : CONV;
```

Description A conversion which expands a statement of membership to a decorated schema.

Conversion

$$\frac{}{\vdash v \in (S)' \Leftrightarrow (x_1 \mathrel{\widehat{=}} v.x'_1, ...) \in S} \quad \begin{array}{l} z\_{\in}\_decor_s\_conv \\ ⓩv \in (S)'⌝ \end{array}$$

where the type of S is

$\mathbb{P}\ [x_1{:}\mathbb{U};...\ ]$

S may be a schema-reference, or (in extended Z) anything of the stated type. Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified.

Definition

```
val z_decor_s_conv = Z_∈_ELIM_C z_∈_decor_s_conv
```

Errors

```
43015 ?0 not of the form: ⓩv ∈ ds⌝ where ds is a decorated
      schema expression
```

SML

```
val z_dec_rename_s_conv : CONV;
```

Description This conversion turns an ill-formed schema-as-declaration into a well-formed one using renaming. The ill-formed schemas-as-declarations in question are those of the form

⌜Z′SchemaDec bind schema⌝;

where *bind* is not equal to ⓩθ *schema*⌝.

Conversion

$$\frac{}{\vdash Z'SchemaDec\ bind\ schema \Leftrightarrow schema[y1/x1,\ ...,\ yk/xk]} \quad \begin{array}{l} z\_dec\_rename_s\_conv \\ ⌜Z'SchemaDec\ bind\ schema⌝ \end{array}$$

Uses In correcting the results of functions which produce results outside Z because of substitution within variable binding constructs.

Errors

```
43060 ?0 is not an ill−formed schema−as−declaration
```

SML

val **z_hide$_s$_conv** : $CONV$;
val **z_∈_hide$_s$_conv** : $CONV$;

Description A conversion concerning the schema hiding.

Conversion

$$\frac{}{\vdash S \setminus_s (x_1, ...) = [y_1 : \mathbb{U}; ... \mid \exists\ x_1 : \mathbb{U}; ... \bullet S]} \quad \begin{array}{l} z\_hide_s\_conv \\ ⓩS \setminus_s (x_1, ...)⌝ \end{array}$$

where S is a schema that has signature variables x_1, x_2, ... and y_1, y_2, ...

Definition

$val\ z\_∈\_hide_s\_conv = ∈\_C\ z\_hide_s\_conv$

Schemas as predicates will be treated as membership statements by this conversion.

Errors

43018 ?0 is not of the form: ⓩ$S \setminus_s (x_1, ...)$⌝ *where S is a schema*

SML

val **z_h_schema_conv** : $CONV$;

Description A conversion from a horizontal schema to a set comprehension.

Conversion

$$\frac{}{\vdash [D|P] = \{D|P \bullet \theta D\}} \quad \begin{array}{l} z\_h\_schema\_conv \\ ⓩ[D|P]⌝ \end{array}$$

See Also *z_∈_h_schema_conv1* and *z_∈_h_schema_conv*, which are more appropriate if the schema expression occurs as a subterm of a membership expression.

Errors

43004 ?0 is not a horizontal schema

SML

val **z_h_schema_pred_conv** : $CONV$;

Description A conversion for eliminating a horizontal schema as a predicate.

Conversion

$$\frac{}{\vdash [D|P] \Leftrightarrow \texttt{"}D\ as\ Predicate\texttt{"} \land P} \quad \begin{array}{l} z\_h\_schema\_pred\_conv \\ ⓩ[D|P]⌝ \end{array}$$

Projections from bindings, which are likely to be introduced, are automatically expanded out. The user may do so with, e.g.,

$MAP\_C\ z\_sel_s\_conv$

The horizontal schema may be decorated.

See Also *z_schema_pred_conv* for a more general conversion.

Errors

43012 ?0 is not a horizontal schema as a predicate

SML

val **z_norm_h_schema_conv** : $CONV$;

Description A conversion for normalising horizontal schemas.

Conversion

$$\frac{}{\vdash [D|P] = [DU|D1 \land P]} \qquad z\_norm\_h\_schema\_conv \ \ulcorner[D|P]\urcorner$$

D1 is the implicit predicate formed from D by *z_decl_pred_conv*, and then simplified. The simplification is that conjuncts of the predicate that are provable by *z_∈_u_conv*, q.v., are proven and then eliminated from *D1*. *DU* is the signature formed from the variables bound by *D*, all of type 𝕌.

Example

$z\_norm\_h\_schema\_conv \ \ulcorner_Z[w{:}W;\ x,y{:}X;\ z{:}\mathbb{U} \mid p\ w\ x\ y\ z]\urcorner$
$=$
$\ulcorner_Z[w{:}\mathbb{U};\ x{:}\mathbb{U};\ y{:}\mathbb{U};\ z{:}\mathbb{U} \mid (w \in W \land x \in X \land y \in X) \land p\ w\ x\ y\ z]\urcorner$

Notice how, since $z \in \mathbb{U}$ can be proven by *z_∈_u_conv*, it is not included in *D1*.

Errors

43004 ?0 is not a horizontal schema

SML

val **z_pre$_s$_conv** : $CONV$;
val **z_∈_pre$_s$_conv** : $CONV$;

Description Schema precondition elimination.

Conversion

$$\frac{}{\vdash pre\ S = [DI \mid (\exists\ DF \bullet S1)]} \qquad z\_pre_s\_conv \ \ulcorner_Z pre\ S\urcorner$$

DI is the declaration formed from the unprimed and input ('?') variables of *S*, given type 𝕌. *DF* is the declaration formed from the primed and output ('!') variables of *S*, given type 𝕌. It is possible for one or both of *DI* and *DF* to be the empty declaration. *S1* is the schema *S* as a predicate.

Definition

$val\ z\_\in\_pre_s\_conv = \in\_C\ z\_pre_s\_conv$

Schemas as predicates will be treated as membership statements by this conversion.

Errors

43007 ?0 is not a schema precondition

SML

```
val z_rename_s_conv : CONV;
val z_∈_rename_s_conv : CONV;
```

Description A conversion concerning schema renaming.

Conversion

$$\frac{}{\vdash v \in S[x_1/y_1,...] \Leftrightarrow (y_1 \mathrel{\widehat{=}} v.x_1,...,z_1 \mathrel{\widehat{=}} v.z_1,...) \in S} \quad \begin{array}{l} z\_\in\_rename_s\_conv \\ ⓩv \in S[x_1/y_1,...]⌝ \end{array}$$

where S has signature variables y_1, ... and z_1, ... Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified. The conversion will fail with error 43035 if applied to a renaming that renames one component to an already existent, unrenamed, component.

Definition

```
val z_rename_s_conv = Z_∈_ELIM_C z_∈_rename_s_conv;
```

Errors

43031 ?0 is not of the form: ⓩ$S[x_1/y_1,...]$⌝ *where S is a schema*
43035 ?0 is of the form ⓩ$S[...,x_i/y_i,...]$⌝ *where* x_i *is already an unrenamed component of S*

SML

```
val z_schema_pred_conv : CONV;
val z_θ_∈_schema_intro_conv : CONV;
```

Description *z_schema_pred_conv* is a conversion from a schema as a predicate to the predicate asserting that its θ-term is a member of the schema.

Conversion

$$\frac{}{\vdash S \Leftrightarrow \theta S \in S} \quad \begin{array}{l} z\_\theta\_\in\_schema\_intro\_conv \\ ⌜S⌝ \end{array}$$

S is any schema as a predicate, including both schema references and horizontal schemas.

z_schema_pred_conv is an alias for *z_θ_∈_schema_intro_conv*.

See Also *z_h_schema_pred_conv* for alternative, *z_θ_conv*, and *z_θ_∈_schema_conv*.

Errors

43014 ?0 is not a schema as a predicate

SML

```
val z_schema_pred_intro_conv : CONV;
```

Description This conversion attempts to convert a predicate that is a membership of a schema into a schema as a predicate.

Conversion

$$\frac{}{\vdash ((x_1 \mathrel{\widehat{=}} x_1,\ ...) \in S) \Leftrightarrow S} \quad \begin{array}{l} z\_schema\_pred\_intro\_conv \\ ⓩ(x_1 \mathrel{\widehat{=}} x_1,\ ...) \in S⌝ \end{array}$$

The input term must have a binding display that binds to each label a variable with the label's name (maintaining decoration).

Errors

43032 ?0 cannot be converted to a schema as a predicate

SML

```
val z_strip_tac : TACTIC;
```

Description z_*strip_tac* is a general purpose tactic for simplifying away the outermost connective of a Z goal. It first attempts to apply *z_∀_tac*. If that fails it then tries to apply the current proof context's conclusion stripping conversion, to rewrite the outermost connective in the goal. Failing that it tries to simplify the goal by applying an applicable member of the following collection of tactics (only one could possibly apply):

```
simple_∀_tac,          ∧_tac,
⇒_T strip_asm_tac,     t_tac
```

Failing either being successful, it tries *concl_in_asms_tac* to prove the goal, and failing that, returns the error message below.

finally, it will attempt to make the goal a "schema as predicate", if possible, by using *z_schema_pred_intro_conv*.

Note how new assumptions generated by the tactic are processed using *strip_asm_tac*, which uses the current proof context's theorem stripping conversion. *z_strip_tac* may produce several new subgoals, or may prove the goal.

The tactic is defined as:

Definition

```
val z_strip_tac = (z_∀_tac ORELSE_T strip_tac)
                  THEN_TRY_T conv_tac z_schema_pred_intro_conv;
```

Uses This is the usual way of simplifying a goal involving Z predicate calculus connectives, and other functions "understood" by the current prof context.

See Also *STRIP_CONCL_T* and *STRIP_THM_THEN* which are used to implement this function. *taut_tac* for an alternative simplifier. *swap_∨_tac* to rearrange the conclusion for tailored stripping.

Errors

28003 There is no stripping technique for ?0 in the current proof context

SML

```
val z_Δs_conv : CONV;
val z_∈_Δs_conv : CONV;
```

Description A conversion concerning the delta schemas.

Conversion

$$\frac{}{\vdash \Delta\ S = [S;\ S']}\qquad z\_\Delta_s\_conv\ \ulcorner_Z \Delta\ S\urcorner$$

Definition

```
val z_∈_Δs_conv = ∈_C z_Δs_conv
```

Schemas as predicates will be treated as membership statements by this conversion.

Errors

43022 ?0 is not of the form: ⌜Z Δ S⌝ where S is a schema

SML

val **z_∈_h_schema_conv1** : *CONV*;

Description A conversion from a predicate asserting membership of a horizontal schema to an existential quantification.

Conversion

$$\frac{}{\vdash v \in [D|P] \Leftrightarrow \exists D'|P' \bullet \theta[D'] = v} \quad \begin{array}{l} z\_\in\_h\_schema\_conv1 \\ \ulcorner v \in [D|P] \urcorner \end{array}$$

Bound variable renaming may be necessary, and thus the priming in the RHS of the result. Schemas as predicates will be treated as membership statements by this conversion.

See Also *z_∈_h_schema_conv* for a faster, if more verbose result from simplifying the same category of terms, *z_h_schema_conv* for a horizontal schema term without and outer ∈.

Errors

43003 ?0 is not of the form $\ulcorner v \in [D|P] \urcorner$
43033 Unable to prove ?0 equal to something of the form: $\ulcorner_Z \exists D'|P' \bullet \theta[D'] = v \urcorner$
use z_∈_h_schema_conv instead, and then work by hand

Error 43033 indicates that there is some sort of variable capture problem preventing the conversion from functioning correctly. As indicated, *z_∈_h_schema_conv* is a conversion that does apply to simplify the input term.

SML

val **z_∈_h_schema_conv** : *CONV*;

Description A conversion from a predicate asserting membership of a horizontal schema to a predicate.

Conversion

$$\frac{}{\vdash v \in [D|P] \Leftrightarrow D' \land P'} \quad \begin{array}{l} z\_\in\_h\_schema\_conv \\ \ulcorner v \in [D|P] \urcorner \end{array}$$

where, if D declares variables x_1, x_2,..., then D' is

"*predicate from* $D[x_1 \setminus v.x_1, ...]$"

as converted by *z_decl_pred_conv*, and P' is

$P[x_1 \setminus v.x_1, ...]$

The execution of the conversion may also involve bound variable renaming. If v is a binding display then $v.x_i$ will be simplified. Though this conversion gives a rather verbose result, it evaluates faster than *z_∈_h_schema_conv1*, and is probably of more practical value in a proof. Schemas as predicates will be treated as membership statements by this conversion.

See Also *z_∈_h_schema_conv1*

Errors

43003 ?0 is not of the form $\ulcorner v \in [D|P] \urcorner$

SML

val **z_Ξs_conv** : $CONV$;
val **z_∈_Ξs_conv** : $CONV$;

Description A conversion concerning Ξ schemas.

Conversion

$$\frac{}{\vdash \Xi\ S = [S;\ S' \mid \theta S = \theta S']}$$

$z\_\Xi_s\_conv$
⌜$_Z$ $\Xi\ S$⌝

Definition

$val\ z\_\in\_\Xi_s\_conv = \in\_C\ z\_\Xi_s\_conv$

Schemas as predicates will be treated as membership statements by this conversion.

Errors

43023 ?0 is not of the form: ⌜$_Z$ $\Xi\ S$⌝ *where S is a schema*

SML

val **z_⇔s_conv** : $CONV$;
val **z_∈_⇔s_conv** : $CONV$;

Description A conversion concerning the membership of a schema bi-implication.

Conversion

$$\frac{}{\begin{array}{l}\vdash v \in (R \Leftrightarrow S) \Leftrightarrow \\ (bind1 \in R \Leftrightarrow bind2 \in S)\end{array}}$$

$z\_\in\_\Leftrightarrow_s\_conv$
⌜$_Z$ $v \in (R \Leftrightarrow S)$⌝

where R and S are schemas that have signature variables r_1, r_2, ... and s_1, s_2, ... respectively, and

$bind1 = (r_1 \mathrel{\widehat{=}} v.r_1, ...)$
$bind2 = (s_1 \mathrel{\widehat{=}} v.s_1, ...)$

Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified.

Definition

$val\ z\_\Leftrightarrow_s\_conv = Z\_\in\_ELIM\_C\ z\_\in\_\Leftrightarrow_s\_conv;$

Errors

43016 ?0 is not of the form: ⌜$_Z$ $v \in (R \Leftrightarrow S)$⌝ *where R and S are schemas*

SML

$val\ \mathbf{z\_\wedge_s\_conv} : CONV;$
$val\ \mathbf{z\_\in\_\wedge_s\_conv} : CONV;$

Description A conversion concerning the membership of a schema conjunction.

Conversion

$$\frac{}{\vdash v \in (R \wedge S) \Leftrightarrow bind1 \in R \wedge bind2 \in S} \quad \begin{array}{l} z\_\in\_\wedge_s\_conv \\ ⓩv \in (R \wedge S)⌝ \end{array}$$

where R and S are schemas that have signature variables r_1, r_2, ... and s_1, s_2, ... respectively, and

$bind1 = (r_1 \mathrel{\widehat{=}} v.r_1, ...)$
$bind2 = (s_1 \mathrel{\widehat{=}} v.s_1, ...)$

Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified.

Definition

$val\ z\_\wedge_s\_conv = Z\_\in\_ELIM\_C\ z\_\in\_\wedge_s\_conv;$

Errors

43001 ?0 is not of the form: ⓩ$v \in (R \wedge S)$⌝ *where R and S are schemas*

SML

$val\ \mathbf{z\_\vee_s\_conv} : CONV;$
$val\ \mathbf{z\_\in\_\vee_s\_conv} : CONV;$

Description A conversion concerning the membership of a schema disjunction.

Conversion

$$\frac{}{\vdash v \in (R \vee S) \Leftrightarrow bind1 \in R \vee bind2 \in S} \quad \begin{array}{l} z\_\in\_\vee_s\_conv \\ ⓩv \in (R \vee S)⌝ \end{array}$$

where R and S are schemas that have signature variables r_1, r_2, ... and s_1, s_2, ... respectively, and

$bind1 = (r_1 \mathrel{\widehat{=}} v.r_1, ...)$
$bind2 = (s_1 \mathrel{\widehat{=}} v.s_1, ...)$

Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified.

Definition

$val\ z\_\vee_s\_conv = Z\_\in\_ELIM\_C\ z\_\in\_\vee_s\_conv;$

Errors

43005 ?0 is not of the form: ⓩ$v \in (R \vee S)$⌝ *where R and S are schemas*

SML

$val\ \mathbf{z\_\neg_s\_conv} : CONV;$
$val\ \mathbf{z\_\in\_\neg_s\_conv} : CONV;$

Description A conversion concerning the membership of a schema negation.

Conversion

$$\frac{}{\vdash v \in (\neg\ S) \Leftrightarrow \neg\ (v \in S)} \quad z\_\in\_\neg_s\_conv\ ⓩv \in (\neg\ S)⌝$$

where S is a schema. Schemas as predicates will be treated as membership statements by this conversion.

Definition

$val\ z\_\neg_s\_conv = Z\_\in\_ELIM\_C\ z\_\in\_\neg_s\_conv;$

Errors

43017 ?0 is not of the form: ⓩv ∈ (¬ S)⌝ where S is a schema

SML

$val\ \mathbf{z\_\Rightarrow_s\_conv} : CONV;$
$val\ \mathbf{z\_\in\_\Rightarrow_s\_conv} : CONV;$

Description A conversion concerning the membership of a schema implication.

Conversion

$$\frac{}{\vdash v \in (R \Rightarrow S) \Leftrightarrow (bind1 \in R \Rightarrow bind2 \in S)} \quad z\_\in\_\Rightarrow_s\_conv\ ⓩv \in (R \Rightarrow S)⌝$$

where R and S are schemas that have signature variables variables r_1, r_2, ... and s_1, s_2, ... respectively, and

$bind1 = (r_1 \mathrel{\widehat{=}} v.r_1, ...)$
$bind2 = (s_1 \mathrel{\widehat{=}} v.s_1, ...)$

Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified.

Definition

$val\ z\_\Rightarrow_s\_conv = Z\_\in\_ELIM\_C\ z\_\in\_\Rightarrow_s\_conv;$

Errors

43006 ?0 is not of the form: ⓩv ∈ (R ⇒ S)⌝ where R and S are schemas

SML

$val\ \mathbf{z\_\forall_s\_conv} : CONV;$
$val\ \mathbf{z\_\in\_\forall_s\_conv} : CONV;$

Description A conversion concerning schema universals.

Conversion

$$\frac{}{\begin{array}{c}\vdash v \in (\exists\ D \mid P\bullet\ S) = \\ \forall\ y : \mathbb{U} \bullet (\text{"}predicate\ from\ D[y.y_1/y_1,...]\text{"} \\ \wedge\ P[y.y_1/y_1,...]) \Rightarrow \\ (x_1 \mathrel{\widehat{=}} v.x_1,...,y_1 \mathrel{\widehat{=}} y.y_1,...) \in S\end{array}} \quad \begin{array}{l} z\_\in\_\forall_s\_conv \\ \ulcorner_Z v \in (\forall\ D \mid P\bullet\ S)\urcorner \end{array}$$

where S is a schema that has signature variables x_1, x_2, ... and y_1, y_2, ... D a declaration that declares y_1, y_2, ... The "predicate from D" will also have schemas as predicates eliminated in favour of bindings being members of schemas. Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified.

Definition

$val\ z\_\forall_s\_conv = Z\_\in\_ELIM\_C\ z\_\in\_\forall_s\_conv;$

Errors

43030 ?0 is not of the form: $\ulcorner_Z v \in (\forall\ D \mid P\bullet\ S)\urcorner$ *where S is a schema*

SML

$val\ \mathbf{z\_\exists_{1s}\_conv} : CONV;$
$val\ \mathbf{z\_\in\_\exists_{1s}\_conv} : CONV;$

Description A conversion concerning schema unique existentials.

Conversion

$$\frac{}{\begin{array}{c}\vdash v \in (\exists_1\ D \mid P\bullet\ S) = \\ \exists_1\ y : \mathbb{U} \bullet \\ (\text{"}predicate\ from\ D[y.y_1/y_1,...]\text{"} \\ \wedge\ P[y.y_1/y_1,...]) \wedge \\ (x_1 \mathrel{\widehat{=}} v.x_1,...,y_1 \mathrel{\widehat{=}} y.y_1,...) \in S\end{array}} \quad \begin{array}{l} z\_\in\_\exists_{1s}\_conv \\ \ulcorner_Z v \in (\exists_1\ D \mid P\bullet\ S)\urcorner \end{array}$$

where S is a schema that has signature variables x_1, x_2, ... and y_1, y_2, ... D a declaration that declares y_1, y_2, ... The "predicate from D" will also have schemas as predicates eliminated in favour of bindings being members of schemas. Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified.

Definition

$val\ z\_\exists_{1s}\_conv = Z\_\in\_ELIM\_C\ z\_\in\_\exists_{1s}\_conv;$

Errors

43021 ?0 is not of the form: $\ulcorner_Z v \in (\exists_1\ D \mid P\bullet\ S)\urcorner$ *where S is a schema*

SML
```
val z_∃s_conv : CONV;
val z_∈_∃s_conv : CONV;
```

Description A conversion concerning membership of schema existentials.

Conversion

$$\frac{}{\vdash v \in (\exists\ D \mid P \bullet\ S) = \exists\ y : \mathbb{U} \bullet (\text{"predicate from } D[y.y_1/y_1,...]\text{"} \land P[y.y_1/y_1,...]) \land (x_1 \mathrel{\widehat{=}} v.x_1,...,y_1 \mathrel{\widehat{=}} y.y_1,...) \in S} \quad \begin{array}{l} z\_\in\_\exists_s\_conv \\ \ulcorner_Z v \in (\exists\ D \mid P \bullet\ S)\urcorner \end{array}$$

where S is a schema that has signature variables x_1, x_2, ... and y_1, y_2, ... D a declaration that declares y_1, y_2, ... The "predicate from D" will also have schemas as predicates eliminated in favour of bindings being members of schemas. Schemas as predicates will be treated as membership statements by this conversion. If v is a binding display then $v.x_i$ will be simplified.

Definition
```
val z_∃s_conv = Z_∈_ELIM_C z_∈_∃s_conv;
```

Errors
```
43020 ?0 is not of the form: ⌜Z v ∈ (∃ D | P• S)⌝ where S is a schema
```

SML
```
val z_⨾s_conv : CONV;
val z_∈_⨾s_conv : CONV;
```

Description A conversion concerning schema sequential composition.

Conversion

$$\frac{}{\vdash (R \mathbin{⨾_s} S) = [x_1{:}\mathbb{U};\ ...;\ z'_1{:}\mathbb{U};\ ...;\ v_1{:}\mathbb{U};\ ...;\ w'_1{:}\mathbb{U};\ ... \mid \exists\ x1{:}\mathbb{U};\ ... \bullet (x_1 \mathrel{\widehat{=}} x_1,\ ...,\ y'_1 \mathrel{\widehat{=}} x1,\ ...,\ z'_1 \mathrel{\widehat{=}} z'_1,\ ...) \in R \land (v_1 \mathrel{\widehat{=}} v_1,\ ...,\ w'_1 \mathrel{\widehat{=}} w'_1,\ ...,\ y_1 \mathrel{\widehat{=}} x1,\ ...) \in S]} \quad \begin{array}{l} z\_⨾_s\_conv \\ \ulcorner_Z (R \mathbin{⨾_s} S)\urcorner \end{array}$$

where R and S are schemas with signature variables as follows:

R		S	
unprimed	primed	unprimed	primed
$x_1, x_2, ...$		$(x_{c_1}, x_{c_2}, ...)$	$(x'_{d_1}, x'_{d_2}, ...)$
$(z_{a_1}, z_{a_2}, ...)$	$z'_1, z'_2, ...$		$(z'_{e_1}, z'_{e_2}, ...)$
$(y_{b_1}, y_{b_2}, ...)$	$y'_1, y'_2, ...$	$y_1, y_2, ...$	$(y'_{f_1}, y'_{f_2}, ...)$
		$v_1, v_2, ...$	$(v'_{g_1}, v'_{g_2}, ...)$
			$w'_1, w'_2, ...$

and $x1$, $x2$, ... are variables whose names are not used for variables, or as labels for the binding types of R or S. The signature variables enclosed in parentheses in the table are not shown in the theorem returned by the conversion but, if present, may result in extra schema declarations of the form $v : \mathbb{U}$ and binding maplets of the form $v \mathrel{\widehat{=}} v$ where v is e.g. z_{a_1}.

Definition
```
val z_∈_⨾s_conv = ∈_C z_⨾s_conv
```

Schemas as predicates will be treated as membership statements by this conversion.

Errors
```
43025 ?0 is not of the form: ⌜Z R ⨾s S⌝ where R and S are schemas
```

SML

```
val z_θ_conv : CONV;
val z_θ_conv1 : CONV;
```

Description *z_θ_conv* conversion from a θ-term to the binding constructor for the schema.

Conversion

$$\frac{}{\vdash \theta S = (n_1 \mathrel{\widehat{=}} n_1,\ n_2 \mathrel{\widehat{=}} n_2,\ ...)} \qquad \begin{array}{l} z\_\theta\_conv \\ ⓩ\theta S⌝ \end{array}$$

z_θ_conv1 is as *z_θ_conv*, except that the conversion only succeeds if the θ term is ill-formed (i.e. is not Z).

Errors

43010 ?0 is not a θ−term
43011 ?0 is not an ill−formed θ−term

SML

```
val z_θ_eq_conv : CONV;
```

Description A conversion from an equality of two θ-terms, or a θ term and a binding display, to an elementwise equality condition.

Conversion

$$\frac{}{\begin{array}{l}\vdash (\theta\ S\ decS = \theta\ T\ decT) \\ \Leftrightarrow n_1 decS = n_1 decT \land ...\end{array}} \qquad \begin{array}{l} z\_\theta\_eq\_conv \\ ⓩ\theta S = \theta T⌝ \end{array}$$

where *decS* and *decT* are the decoration of the respective schemas. Also:

Conversion

$$\frac{}{\begin{array}{l}\vdash (\theta S = (n_1 \mathrel{\widehat{=}} x_1,\ ...)) \\ \Leftrightarrow (n_1 = x_1) \land ...\end{array}} \qquad \begin{array}{l} z\_\theta\_eq\_conv \\ ⓩ\theta S = (n_1 \mathrel{\widehat{=}} x_1,\ ...)⌝ \end{array}$$

Uses Used in combination with *z_binding_eq_conv2* to give η-terms the same status as binding displays.

Errors

43034 ?0 is not of the form: ⓩθS = θT⌝ *or* ⓩθS = (n₁ ≙ x₁, ...)⌝

SML

```
val z_θ_∈_schema_conv : CONV;
```

Description A conversion from a predicate asserting that the θ-term of a schema is a member of the schema to that schema as a predicate.

Conversion

$$\frac{}{\vdash \theta S \in S \Leftrightarrow S} \qquad \begin{array}{l} z\_\theta\_\in\_schema\_conv \\ ⓩ\theta S \in S⌝ \end{array}$$

Note that the schemas cannot be decorated, as the type of ⓩθ S ′⌝ is the same as the type of ⓩθ S⌝. Other than that *S* may be any schema as a predicate, including schema references and horizontal schemas.

See Also *z_θ_∈_schema_intro_conv*; and *z_pred_dec_conv*, which subsumes this conversion.

Errors

43002 ?0 is not of the form ⌜θS ∈ S⌝ where ⌜S⌝ is an undecorated schema

SML

$val\ \mathbf{z\_\upharpoonright_s\_conv} : CONV;$
$val\ \mathbf{z\_\in\_\upharpoonright_s\_conv} : CONV;$

Description A conversion concerning the membership of a schema projection.

Conversion

$$\frac{}{\vdash (R \upharpoonright_s S) = (R \land S) \setminus_s (x_1, x_2, ...)} \qquad \begin{array}{l} z\_\upharpoonright_s\_conv \\ \ulcorner_Z (R \upharpoonright_s S)\urcorner \end{array}$$

where R and S are schemas and x_1, x_2, ... are the signature variables of R that are not signature variables of S.

Definition

$val\ z\_\in\_\upharpoonright_s\_conv = \in\_C\ z\_\upharpoonright_s\_conv$

Schemas as predicates will be treated as membership statements by this conversion.

Errors

43019 ?0 is not of the form: $\ulcorner_Z R \upharpoonright_s S\urcorner$ *where R and S are schemas*

THEORIES

9.1 Theory Listings

This section contains the listings of each theory.

9.1.1 The Theory z_arithmetic_tools

9.1.1.1 Parents

z_numbers

9.1.1.2 Children

z_numbers1 *z_library*

9.1.1.3 Constants

ℤ_z	$\mathbb{Z} \rightarrow \mathbb{Z}$
z_ℤ	$\mathbb{Z} \rightarrow \mathbb{Z}$

9.1.1.4 Definitions

z_ℤ
ℤ_z

$$
\begin{array}{l}
\vdash ConstSpec \\
\quad (\lambda\ (\$"z\_\mathbb{Z}'",\ \mathbb{Z}\_z') \\
\qquad \bullet\ \mathbb{Z}\_z'\ (\mathbb{N}\mathbb{Z}\ 1) = \ulcorner_Z 1 \urcorner \\
\qquad\quad \land\ (\forall\ i\ j \\
\qquad\quad \bullet\ \mathbb{Z}\_z'\ (i + j) = \ulcorner_Z \ulcorner \mathbb{Z}\_z'\ i \urcorner + \ulcorner \mathbb{Z}\_z'\ j \urcorner \urcorner) \\
\qquad\quad \land\ (\forall\ i \bullet\ \mathbb{Z}\_z'\ (\sim\ i) = \ulcorner_Z \sim \ulcorner \mathbb{Z}\_z'\ i \urcorner \urcorner) \\
\qquad\quad \land\ \$"z\_\mathbb{Z}'"\ \ulcorner_Z 1 \urcorner = \mathbb{N}\mathbb{Z}\ 1 \\
\qquad\quad \land\ (\forall\ i\ j \\
\qquad\quad \bullet\ \$"z\_\mathbb{Z}'"\ \ulcorner_Z i + j \urcorner = \$"z\_\mathbb{Z}'"\ i + \$"z\_\mathbb{Z}'"\ j) \\
\qquad\quad \land\ (\forall\ i \bullet\ \$"z\_\mathbb{Z}'"\ \ulcorner_Z \sim\ i \urcorner = \sim\ (\$"z\_\mathbb{Z}'"\ i)) \\
\qquad\quad \land\ (\forall\ x \bullet\ \$"z\_\mathbb{Z}'"\ (\mathbb{Z}\_z'\ x) = x) \\
\qquad\quad \land\ (\forall\ y \bullet\ \mathbb{Z}\_z'\ (\$"z\_\mathbb{Z}'"\ y) = y)) \\
\quad (z\_\mathbb{Z},\ \mathbb{Z}\_z)
\end{array}
$$

9.1.1.5 Theorems

z_ℤ_consistent
ℤ_z_consistent

$$
\begin{array}{l}
\vdash Consistent \\
\quad (\lambda\ (\$"z\_\mathbb{Z}'",\ \mathbb{Z}\_z') \\
\qquad \bullet\ \mathbb{Z}\_z'\ (\mathbb{N}\mathbb{Z}\ 1) = \ulcorner_Z 1 \urcorner \\
\qquad\quad \land\ (\forall\ i\ j \\
\qquad\quad \bullet\ \mathbb{Z}\_z'\ (i + j) = \ulcorner_Z \ulcorner \mathbb{Z}\_z'\ i \urcorner + \ulcorner \mathbb{Z}\_z'\ j \urcorner \urcorner) \\
\qquad\quad \land\ (\forall\ i \bullet\ \mathbb{Z}\_z'\ (\sim\ i) = \ulcorner_Z \sim \ulcorner \mathbb{Z}\_z'\ i \urcorner \urcorner) \\
\qquad\quad \land\ \$"z\_\mathbb{Z}'"\ \ulcorner_Z 1 \urcorner = \mathbb{N}\mathbb{Z}\ 1 \\
\qquad\quad \land\ (\forall\ i\ j \\
\qquad\quad \bullet\ \$"z\_\mathbb{Z}'"\ \ulcorner_Z i + j \urcorner = \$"z\_\mathbb{Z}'"\ i + \$"z\_\mathbb{Z}'"\ j) \\
\qquad\quad \land\ (\forall\ i \bullet\ \$"z\_\mathbb{Z}'"\ \ulcorner_Z \sim\ i \urcorner = \sim\ (\$"z\_\mathbb{Z}'"\ i)) \\
\qquad\quad \land\ (\forall\ x \bullet\ \$"z\_\mathbb{Z}'"\ (\mathbb{Z}\_z'\ x) = x) \\
\qquad\quad \land\ (\forall\ y \bullet\ \mathbb{Z}\_z'\ (\$"z\_\mathbb{Z}'"\ y) = y))
\end{array}
$$

z_ℤ_plus_thm $\vdash \forall\ i\ j \bullet\ z\_\mathbb{Z}\ \ulcorner_Z i + j \urcorner = z\_\mathbb{Z}\ i + z\_\mathbb{Z}\ j$

z_ℤ_times_thm

$$\vdash \forall\ i\ j \bullet\ z\_\mathbb{Z}\ \ulcorner_Z i * j \urcorner = z\_\mathbb{Z}\ i * z\_\mathbb{Z}\ j$$

z_ℤ_subtract_thm
$\vdash \forall\ i\ j\bullet\ z\_\mathbb{Z}\ ⌜_Z i - j⌝ = z\_\mathbb{Z}\ i - z\_\mathbb{Z}\ j$

z_ℤ_minus_thm
$\vdash \forall\ i\bullet\ z\_\mathbb{Z}\ ⌜_Z\sim\ i⌝ = \sim\ (z\_\mathbb{Z}\ i)$

ℤ_z_plus_thm $\vdash \forall\ i\ j\bullet\ \mathbb{Z}\_z\ (i + j) = ⌜_Z⌜\mathbb{Z}\_z\ i⌝ + ⌜\mathbb{Z}\_z\ j⌝⌝$

ℤ_z_times_thm
$\vdash \forall\ i\ j\bullet\ \mathbb{Z}\_z\ (i * j) = ⌜_Z⌜\mathbb{Z}\_z\ i⌝ * ⌜\mathbb{Z}\_z\ j⌝⌝$

ℤ_z_subtract_thm
$\vdash \forall\ i\ j\bullet\ \mathbb{Z}\_z\ (i - j) = ⌜_Z⌜\mathbb{Z}\_z\ i⌝ - ⌜\mathbb{Z}\_z\ j⌝⌝$

ℤ_z_minus_thm
$\vdash \forall\ i\bullet\ \mathbb{Z}\_z\ (\sim\ i) = ⌜_Z\sim\ ⌜\mathbb{Z}\_z\ i⌝⌝$

z_ℤ_one_one_thm
$\vdash \forall\ i\ j\bullet\ z\_\mathbb{Z}\ i = z\_\mathbb{Z}\ j \Leftrightarrow i = j$

ℤ_z_one_one_thm
$\vdash \forall\ i\ j\bullet\ \mathbb{Z}\_z\ i = \mathbb{Z}\_z\ j \Leftrightarrow i = j$

z_≤_ℤ_≤_thm $\vdash \forall\ i\ j\bullet\ ⌜_Z(i, j)⌝ \in ⌜_Z(\_ \leq \_)⌝ \Leftrightarrow z\_\mathbb{Z}\ i \leq z\_\mathbb{Z}\ j$

z_less_ℤ_less_thm
$\vdash \forall\ i\ j\bullet\ ⌜_Z(i, j)⌝ \in ⌜_Z(\_ < \_)⌝ \Leftrightarrow z\_\mathbb{Z}\ i < z\_\mathbb{Z}\ j$

9.1.2 The Z Theory z_bags

9.1.2.1 Parents

z_sequences

9.1.2.2 Children

z_library

9.1.2.3 Global Variables

bag X	$\mathbb{P}\ (X \leftrightarrow \mathbb{Z})$
count[**X**]	$(X \leftrightarrow \mathbb{Z}) \leftrightarrow X \leftrightarrow \mathbb{Z}$
(_ **in** _)[**X**]	$X \leftrightarrow X \leftrightarrow \mathbb{Z}$
(_ ⊎ _)[**X**]	$(X \leftrightarrow \mathbb{Z}) \times (X \leftrightarrow \mathbb{Z}) \leftrightarrow X \leftrightarrow \mathbb{Z}$
items[**X**]	$(\mathbb{Z} \leftrightarrow X) \leftrightarrow X \leftrightarrow \mathbb{Z}$
(⟦ ... ⟧)[**X**]	$(\mathbb{Z} \leftrightarrow X) \leftrightarrow X \leftrightarrow \mathbb{Z}$

9.1.2.4 Fixity

fun 0 rightassoc

(⟦ ... ⟧)

fun 30 leftassoc

(_ ⊎ _)

gen 70 rightassoc

(*bag* _)

rel (_ *in* _)

9.1.2.5 Axioms

count
$$\vdash [X](count[X] \in bag\ X \rightarrowtail\!\!\!\!\twoheadrightarrow X \rightarrow \mathbb{N} \\ \land (\forall\ x : X;\ B : bag\ X \\ \bullet\ count[X]\ B = (\lambda\ x : X \bullet 0) \oplus B))$$

_ in _
$$\vdash [X]((\_\ in\ \_)[X] \in X \leftrightarrow bag\ X \\ \land (\forall\ x : X;\ B : bag\ X \\ \bullet\ (x,\ B) \in (\_\ in\ \_)[X] \Leftrightarrow x \in dom\ B))$$

_ ⊎ _
$$\vdash [X]((\_ \uplus \_)[X] \in bag\ X \times bag\ X \rightarrow bag\ X \\ \land (\forall\ B,\ C : bag\ X;\ x : X \\ \bullet\ count\ ((\_ \uplus \_)[X]\ (B,\ C))\ x \\ = count\ B\ x + count\ C\ x))$$

items
$$\vdash [X](items[X] \in seq\ X \rightarrow bag\ X \\ \land (\forall\ s : seq\ X;\ x : X \\ \bullet\ count\ (items[X]\ s)\ x \\ = \#\ \{i : dom\ s \mid s\ i = x\}))$$

⟦ ... ⟧
$$\vdash [X]((⟦\ ...\ ⟧)[X] \in seq\ X \rightarrow bag\ X \\ \land (⟦\ ...\ ⟧)[X]\ \langle\rangle = \{\} \\ \land (\forall\ x : X;\ s : seq\ X \\ \bullet\ (⟦\ ...\ ⟧)[X]\ (\langle x\rangle \frown s) \\ = (⟦\ ...\ ⟧)[X]\ s \oplus \{x \mapsto (⟦\ ...\ ⟧)[X]\ s\ x + 1\}))$$

9.1.2.6 Definitions

bag _ $\vdash [X](bag\ X = X \nrightarrow \mathbb{N}_1)$

9.1.3 The Z Theory z_functions

9.1.3.1 Parents

$z\_relations$

9.1.3.2 Children

$z\_functions1$ $z\_numbers$

9.1.3.3 Global Variables

$\mathbf{X} \mathbin{\ooalign{$\rightarrow$\cr\hidewidth$\mid$\hidewidth}} \mathbf{Y}$	$\mathbb{P}\ (X \leftrightarrow Y)$
$\mathbf{X} \rightarrowtail\!\!\!\mid \mathbf{Y}$	$\mathbb{P}\ (X \leftrightarrow Y)$
$\mathbf{X} \rightarrowtail \mathbf{Y}$	$\mathbb{P}\ (X \leftrightarrow Y)$
$\mathbf{X} \twoheadrightarrow\!\!\!\mid \mathbf{Y}$	$\mathbb{P}\ (X \leftrightarrow Y)$
$\mathbf{X} \twoheadrightarrow \mathbf{Y}$	$\mathbb{P}\ (X \leftrightarrow Y)$
$\mathbf{X} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathbf{Y}$	$\mathbb{P}\ (X \leftrightarrow Y)$

9.1.3.4 Fixity

gen 5 rightassoc

$(\_ \rightarrowtail\!\!\!\!\twoheadrightarrow \_)$ $(\_ \twoheadrightarrow\!\!\!\mid \_)$ $(\_ \rightarrow\!\!\!\mid \_)$ $(\_ \rightarrowtail \_)$ $(\_ \twoheadrightarrow \_)$ $(\_ \rightarrowtail\!\!\!\mid \_)$

9.1.3.5 Definitions

$\_ \rightarrow\!\!\!\mid \_$

$$\vdash [X, Y](X \rightarrow\!\!\!\mid Y = \{f : X \leftrightarrow Y \mid \forall x : X;\ y1, y2 : Y \bullet x \mapsto y1 \in f \land x \mapsto y2 \in f \Rightarrow y1 = y2\})$$

$\_ \rightarrowtail\!\!\!\mid \_$

$$\vdash [X, Y](X \rightarrowtail\!\!\!\mid Y = \{f : X \rightarrow\!\!\!\mid Y \mid \forall x1, x2 : dom\ f \bullet f\ x1 = f\ x2 \Rightarrow x1 = x2\})$$

$\_ \rightarrowtail \_$ $\vdash [X, Y](X \rightarrowtail Y = (X \rightarrowtail\!\!\!\mid Y) \cap (X \rightarrow Y))$

$\_ \twoheadrightarrow\!\!\!\mid \_$ $\vdash [X, Y](X \twoheadrightarrow\!\!\!\mid Y = \{f : X \rightarrow\!\!\!\mid Y \mid ran\ f = Y\})$

$\_ \twoheadrightarrow \_$ $\vdash [X, Y](X \twoheadrightarrow Y = (X \twoheadrightarrow\!\!\!\mid Y) \cap (X \rightarrow Y))$

$\_ \rightarrowtail\!\!\!\!\twoheadrightarrow \_$ $\vdash [X, Y](X \rightarrowtail\!\!\!\!\twoheadrightarrow Y = (X \twoheadrightarrow Y) \cap (X \rightarrowtail Y))$

9.1.3.6 Theorems

z_⇸_thm
$$\begin{aligned}&\vdash \forall\ f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet f \in X ⇸ Y\\ &\quad\quad \Leftrightarrow f \in X \leftrightarrow Y\\ &\quad\quad\quad \land (\forall\ x : X;\ y1,\ y2 : Y\\ &\quad\quad\quad\quad \bullet (x,\ y1) \in f \land (x,\ y2) \in f \Rightarrow y1 = y2)\end{aligned}$$

z_⇸_thm1
$$\begin{aligned}&\vdash \forall\ f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet f \in X ⇸ Y\\ &\quad\quad \Leftrightarrow f \in X \leftrightarrow Y\\ &\quad\quad\quad \land (\forall\ x : \mathbb{U};\ y1,\ y2 : \mathbb{U}\\ &\quad\quad\quad\quad \mid x \in X \land y1 \in Y \land y2 \in Y\\ &\quad\quad\quad\quad \bullet (x,\ y1) \in f \land (x,\ y2) \in f \Rightarrow y1 = y2)\end{aligned}$$

z_→_thm
$$\begin{aligned}&\vdash \forall\ f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet f \in X \rightarrow Y \Leftrightarrow f \in X ⇸ Y \land dom\ f = X\end{aligned}$$

z_⤔_thm
$$\begin{aligned}&\vdash \forall\ f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet f \in X ⤔ Y\\ &\quad\quad \Leftrightarrow f \in X ⇸ Y\\ &\quad\quad\quad \land (\forall\ x1,\ x2 : \mathbb{U}\\ &\quad\quad\quad\quad \mid x1 \in dom\ f \land x2 \in dom\ f\\ &\quad\quad\quad\quad \bullet f\ x1 = f\ x2 \Rightarrow x1 = x2)\end{aligned}$$

z_↣_thm
$$\begin{aligned}&\vdash \forall\ f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet f \in X \rightarrowtail Y\\ &\quad\quad \Leftrightarrow f \in X \rightarrow Y\\ &\quad\quad\quad \land (\forall\ x1,\ x2 : \mathbb{U}\\ &\quad\quad\quad\quad \mid x1 \in dom\ f \land x2 \in dom\ f\\ &\quad\quad\quad\quad \bullet f\ x1 = f\ x2 \Rightarrow x1 = x2)\end{aligned}$$

z_⤀_thm
$$\begin{aligned}&\vdash \forall\ f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet f \in X ⤀ Y \Leftrightarrow f \in X ⇸ Y \land ran\ f = Y\end{aligned}$$

z_↠_thm
$$\begin{aligned}&\vdash \forall\ f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet f \in X \twoheadrightarrow Y \Leftrightarrow f \in X \rightarrow Y \land ran\ f = Y\end{aligned}$$

z_⤖_thm
$$\begin{aligned}&\vdash \forall\ f : \mathbb{U};\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet f \in X ⤖ Y\\ &\quad\quad \Leftrightarrow f \in X \rightarrow Y\\ &\quad\quad\quad \land ran\ f = Y\\ &\quad\quad\quad \land (\forall\ x1,\ x2 : \mathbb{U}\\ &\quad\quad\quad\quad \mid x1 \in dom\ f \land x2 \in dom\ f\\ &\quad\quad\quad\quad \bullet f\ x1 = f\ x2 \Rightarrow x1 = x2)\end{aligned}$$

z_→_app_thm
$$\begin{aligned}&\vdash \forall\ X : \mathbb{U};\ Y : \mathbb{U};\ f : \mathbb{U};\ x : \mathbb{U}\\ &\quad \bullet f \in X \rightarrow Y \land x \in X \Rightarrow f\ x \in Y \land (x,\ f\ x) \in f\end{aligned}$$

z_∈_first_thm
$$\vdash \forall\ x : \mathbb{U} \bullet x \in first \Leftrightarrow x.1.1 = x.2$$

z_∈_second_thm
$$\vdash \forall\ x : \mathbb{U} \bullet x \in second \Leftrightarrow x.1.2 = x.2$$

z_→_app_∈_rel_thm
$$\vdash \forall\ X : \mathbb{U};\ Y : \mathbb{U} \bullet \forall\ f : X \rightarrow Y;\ x : X \bullet (x,\ f\ x) \in f$$

z_→_app_eq_⇔_∈_rel_thm
$$\begin{aligned}&\vdash \forall\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet \forall\ f : X \rightarrow Y;\ x : X;\ z : \mathbb{U} \bullet f\ x = z \Leftrightarrow (x,\ z) \in f\end{aligned}$$

z_→_∈_rel_⇔_app_eq_thm
$$\begin{aligned}&\vdash \forall\ X : \mathbb{U};\ Y : \mathbb{U}\\ &\quad \bullet \forall\ f : X \rightarrow Y;\ x : X;\ z : \mathbb{U} \bullet (x,\ z) \in f \Leftrightarrow f\ x = z\end{aligned}$$

z_⇸_clauses $\vdash \forall Y : \mathbb{U} \bullet \{\} \pfun Y = \{\{\}\} \land Y \pfun \{\} = \{\{\}\}$

z_→_clauses $\vdash \forall Y : \mathbb{U} \bullet \{\} \fun Y = \{\{\}\} \land Y \fun \{\} = \{x : \mathbb{U} \mid x = \{\} \land Y = \{\}\}$

z_⤔_clauses $\vdash \forall Y : \mathbb{U} \bullet \{\} \pinj Y = \{\{\}\} \land Y \pinj \{\} = \{\{\}\}$

z_↣_clauses $\vdash \forall Y : \mathbb{U} \bullet \{\} \inj Y = \{\{\}\} \land Y \inj \{\} = \{x : \mathbb{U} \mid x = \{\} \land Y = \{\}\}$

z_⤀_clauses $\vdash \forall Y : \mathbb{U} \bullet \{\} \psurj Y = \{x : \mathbb{U} \mid x = \{\} \land Y = \{\}\} \land Y \psurj \{\} = \{\{\}\}$

z_↠_clauses $\vdash \forall Y : \mathbb{U} \bullet \{\} \surj Y = \{x : \mathbb{U} \mid x = \{\} \land Y = \{\}\} \land Y \surj \{\} = \{x : \mathbb{U} \mid x = \{\} \land Y = \{\}\}$

z_⤖_clauses $\vdash \forall Y : \mathbb{U} \bullet \{\} \bij Y = \{x : \mathbb{U} \mid x = \{\} \land Y = \{\}\} \land Y \bij \{\} = \{x : \mathbb{U} \mid x = \{\} \land Y = \{\}\}$

z_fun_app_clauses

$$\vdash \forall f : \mathbb{U}; x : \mathbb{U}; y : \mathbb{U}; X : \mathbb{U}; Y : \mathbb{U} \bullet (f \in X \pfun Y \lor f \in X \pinj Y \lor f \in X \psurj Y \lor f \in X \fun Y \lor f \in X \inj Y \lor f \in X \surj Y \lor f \in X \bij Y) \land (x, y) \in f \Rightarrow f\ x = y$$

z_fun_∈_clauses

$$\vdash \forall f : \mathbb{U}; x : \mathbb{U}; X : \mathbb{U}; Y : \mathbb{U} \bullet ((f \in X \fun Y \lor f \in X \inj Y \lor f \in X \surj Y \lor f \in X \bij Y) \land x \in X \Rightarrow f\ x \in Y) \land ((f \in X \pfun Y \lor f \in X \pinj Y \lor f \in X \psurj Y) \land x \in dom\ f \Rightarrow f\ x \in Y)$$

z_fun_dom_clauses

$$\vdash \forall f : \mathbb{U}; X : \mathbb{U}; Y : \mathbb{U} \bullet (f \in X \pfun Y \lor f \in X \pinj Y \lor f \in X \psurj Y \Rightarrow dom\ f \subseteq X) \land (f \in X \fun Y \lor f \in X \inj Y \lor f \in X \surj Y \lor f \in X \bij Y \Rightarrow dom\ f = X)$$

z_fun_ran_clauses

$$\vdash \forall f : \mathbb{U}; X : \mathbb{U}; Y : \mathbb{U} \bullet (f \in X \fun Y \lor f \in X \pfun Y \lor f \in X \pinj Y \lor f \in X \inj Y \Rightarrow ran\ f \subseteq Y) \land (f \in X \psurj Y \lor f \in X \surj Y \lor f \in X \bij Y \Rightarrow ran\ f = Y)$$

9.1.4 The Z Theory z_functions1

9.1.4.1 Parents

z_functions

9.1.4.2 Children

z_numbers1

9.1.4.3 Theorems

z_⊕_↦_app_thm
$\vdash \forall f : \mathbb{U};\ x : \mathbb{U};\ y : \mathbb{U} \bullet (f \oplus \{x \mapsto y\})\ x = y$

z_dom_⊕_↦_thm
$\vdash \forall f : \mathbb{U};\ x : \mathbb{U};\ y : \mathbb{U}$
$\bullet\ dom\ (f \oplus \{x \mapsto y\}) = dom\ f \cup \{x\}$

z_⊕_↦_∈_→_thm
$\vdash [X,$
$Y](\forall f : X \rightarrow Y;\ x : X;\ y : Y \bullet f \oplus \{x \mapsto y\} \in X \rightarrow Y)$

z_⊕_↦_app_thm1
$\vdash [X,$
$Y](\forall f : X \rightarrow Y;\ x2 : X;\ x1 : \mathbb{U};\ y : \mathbb{U}$
$\mid \neg$
$x2 = x1$
$\bullet\ (f \oplus \{x1 \mapsto y\})\ x2 = f\ x2)$

z_◁_→_thm
$\vdash [Y,$
$Z](\forall X : \mathbb{U};\ f : Y \rightarrow Z$
$\bullet\ X \subseteq Y \Rightarrow X \lhd f \in X \rightarrow ran\ (X \lhd f))$

z_ran_◁_thm
$\vdash [Y,$
$Z](\forall X : \mathbb{U};\ f : Y \rightarrow Z$
$\bullet\ ran\ (X \lhd f)$
$= ran\ f$
$\setminus \{y : \mathbb{U}$
$\mid \forall x : \mathbb{U} \mid (x, y) \in f \bullet \neg x \in X\})$

z_∈_→_thm
$\vdash \forall X : \mathbb{U};\ Y : \mathbb{U}$
$\bullet\ \forall f : X \rightarrow Y;\ x : \mathbb{U};\ y : \mathbb{U}$
$\mid (x, y) \in f$
$\bullet\ x \in X \land y \in Y$

z_→_ran_eq_↠_thm
$\vdash \forall A : \mathbb{U};\ B : \mathbb{U}$
$\bullet\ (\exists f : A \rightarrow B \bullet ran\ f = B) \Leftrightarrow (\exists f : A \twoheadrightarrow B \bullet true)$

z_↣_ran_eq_⤖_thm
$\vdash \forall A : \mathbb{U};\ B : \mathbb{U}$
$\bullet\ (\exists f : A \rightarrowtail B \bullet ran\ f = B) \Leftrightarrow (\exists f : A \leftrightarrow\!\!\!\!\twoheadrightarrow B \bullet true)$

z_ran_mono_thm
$\vdash \forall X : \mathbb{U};\ Y, Z : \mathbb{U};\ f : \mathbb{U}$
$\mid f \in X \rightarrow Y \land ran\ f \subseteq Z$
$\bullet\ f \in X \rightarrow Z$

z_⇸_thm2
$\vdash \forall A : \mathbb{U};\ B : \mathbb{U};\ f : \mathbb{U}$
$\bullet\ f \in A \nrightarrow B \Leftrightarrow f \in dom\ f \rightarrow B \land dom\ f \subseteq A$

z_↠_thm1 $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ f : \mathbb{U}$
$\bullet\ f \in A \twoheadrightarrow B \Leftrightarrow f \in A \rightarrow B \land B \subseteq ran\ f$

z_⤔_thm1 $\vdash [X,$
$Y](X \rightarrowtail\!\!\!\!\!\!\!\shortmid\ \ Y$
$= \{f : X \nrightarrow Y$
$\mid \forall\ x1,\ x2 : \mathbb{U};\ y : \mathbb{U}$
$\bullet\ (x1,\ y) \in f \land (x2,\ y) \in f \Rightarrow x1 = x2\})$

z_→_dom_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ f : \mathbb{U} \bullet f \in A \rightarrow B \Rightarrow f \in dom\ f \rightarrow B$

z_↣_thm1 $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ f : \mathbb{U}$
$\bullet\ f \in A \rightarrowtail B$
$\Leftrightarrow f \in A \rightarrow B$
$\land (\forall\ x,\ y : \mathbb{U};\ z : \mathbb{U}$
$\bullet\ (x,\ z) \in f \land (y,\ z) \in f \Rightarrow x = y)$

z_∪_↔_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ D : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \leftrightarrow B \land g \in C \leftrightarrow D \Rightarrow f \cup g \in A \cup C \leftrightarrow B \cup D$

z_ran_∪_thm $\vdash \forall\ f : \mathbb{U};\ g : \mathbb{U} \bullet ran\ (f \cup g) = ran\ f \cup ran\ g$

z_∪_→_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ D : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \rightarrow B \land g \in C \rightarrow D \land A \cap C = \{\}$
$\Rightarrow f \cup g \in A \cup C \rightarrow B \cup D$

z_∪_↣_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ D : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \rightarrowtail B \land g \in C \rightarrowtail D \land A \cap C = \{\} \land B \cap D = \{\}$
$\Rightarrow f \cup g \in A \cup C \rightarrowtail B \cup D$

z_∪_↠_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ D : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \twoheadrightarrow B \land g \in C \twoheadrightarrow D \land A \cap C = \{\}$
$\Rightarrow f \cup g \in A \cup C \twoheadrightarrow B \cup D$

z_∪_⤖_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ D : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \rightarrowtail\!\!\!\!\rightarrow B \land g \in C \rightarrowtail\!\!\!\!\rightarrow D \land A \cap C = \{\} \land B \cap D = \{\}$
$\Rightarrow f \cup g \in A \cup C \rightarrowtail\!\!\!\!\rightarrow B \cup D$

z_∘_→_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \rightarrow B \land g \in B \rightarrow C \Rightarrow g \circ f \in A \rightarrow C$

z_∘_↠_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \twoheadrightarrow B \land g \in B \twoheadrightarrow C \Rightarrow g \circ f \in A \twoheadrightarrow C$

z_∘_↣_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \rightarrowtail B \land g \in B \rightarrowtail C \Rightarrow g \circ f \in A \rightarrowtail C$

z_∘_⤖_thm $\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ C : \mathbb{U};\ f : \mathbb{U};\ g : \mathbb{U}$
$\bullet\ f \in A \rightarrowtail\!\!\!\!\rightarrow B \land g \in B \rightarrowtail\!\!\!\!\rightarrow C \Rightarrow g \circ f \in A \rightarrowtail\!\!\!\!\rightarrow C$

z_rel_inv_⤖_thm
$\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ f : \mathbb{U} \bullet f \in A \rightarrowtail\!\!\!\!\rightarrow B \Rightarrow f\ ^{\sim} \in B \rightarrowtail\!\!\!\!\rightarrow A$

z_id_thm1 $\vdash \forall\ X : \mathbb{U};\ x,\ y : \mathbb{U} \bullet (x,\ y) \in id\ X \Leftrightarrow x \in X \land x = y$

z_id_⤖_thm $\vdash \forall\ X : \mathbb{U} \bullet id\ X \in X \rightarrowtail\!\!\!\!\rightarrow X$

z_simple_swap_⤖_thm
$\vdash \forall\ x,\ y : \mathbb{U} \bullet \{(x,\ y),\ (y,\ x)\} \in \{x,\ y\} \rightarrowtail\!\!\!\!\rightarrow \{x,\ y\}$

z_swap_⤖_thm $\vdash \forall\ X : \mathbb{U}$
$\bullet\ \forall\ x,\ y : X$
$\bullet\ \exists\ g : X \rightarrowtail\!\!\!\!\rightarrow X \bullet (x,\ y) \in g \land (y,\ x) \in g$

z_⤖_trans_thm
$\vdash \forall\ X : \mathbb{U} \bullet \forall\ x,\ y : X \bullet \exists\ g : X \rightarrowtail\!\!\!\!\rightarrow X \bullet (x,\ y) \in g$

z_dom_f_↔_f_thm
$\vdash \forall\ A : \mathbb{U};\ B : \mathbb{U};\ f : \mathbb{U}$
$\bullet\ f \in A \leftrightarrow B$
$\Rightarrow \{x : A;\ y : B$

$| (x, y) \in f$
$\bullet (x, (x, y))\}$
$\in dom\ f \leftrightarrow f$

z_dom_f_→_f_thm
$\vdash \forall A : \mathbb{U}; B : \mathbb{U}; f : \mathbb{U}$
$\bullet f \in A \rightarrow B$
$\Rightarrow \{x : A; y : B$
$| (x, y) \in f$
$\bullet (x, (x, y))\}$
$\in dom\ f \rightarrow f$

z_dom_f_↠_f_thm
$\vdash \forall A : \mathbb{U}; B : \mathbb{U}; f : \mathbb{U}$
$\bullet f \in A \rightarrow B$
$\Rightarrow \{x : A; y : B$
$| (x, y) \in f$
$\bullet (x, (x, y))\}$
$\in dom\ f \twoheadrightarrow f$

z_dom_f_↣_f_thm
$\vdash \forall A : \mathbb{U}; B : \mathbb{U}; f : \mathbb{U}$
$\bullet f \in A \rightarrow B$
$\Rightarrow \{x : A; y : B$
$| (x, y) \in f$
$\bullet (x, (x, y))\}$
$\in dom\ f \rightarrowtail f$

z_dom_f_⤖_f_thm
$\vdash \forall A : \mathbb{U}; B : \mathbb{U}; f : \mathbb{U}$
$\bullet f \in A \rightarrow B$
$\Rightarrow \{x : A; y : B$
$| (x, y) \in f$
$\bullet (x, (x, y))\}$
$\in dom\ f \twoheadrightarrowtail f$

z_∩_↔_thm $\vdash \forall X : \mathbb{U}; Y : \mathbb{U}; f, g : \mathbb{U}$
$| f \in X \leftrightarrow Y$
$\bullet f \cap g \in dom\ (f \cap g) \leftrightarrow ran\ (f \cap g)$

z_↔_ran_thm $\vdash \forall X : \mathbb{U}; f : \mathbb{U} \bullet f \in X \leftrightarrow ran\ f \Leftrightarrow f \in X \leftrightarrow \mathbb{U}$

z_→_ran_thm $\vdash \forall X : \mathbb{U}; f : \mathbb{U} \bullet f \in X \rightarrow ran\ f \Leftrightarrow f \in X \rightarrow \mathbb{U}$

z_∩_→_thm $\vdash \forall X : \mathbb{U}; Y : \mathbb{U}; f, g : \mathbb{U}$
$| f \in X \rightarrow Y$
$\bullet f \cap g \in dom\ (f \cap g) \rightarrow ran\ (f \cap g)$

z_∩_↣_thm $\vdash \forall X : \mathbb{U}; Y : \mathbb{U}; f, g : \mathbb{U}$
$| f \in X \rightarrowtail Y$
$\bullet f \cap g \in dom\ (f \cap g) \rightarrowtail ran\ (f \cap g)$

z_↠_ran_thm $\vdash \forall X : \mathbb{U}; Y : \mathbb{U}; f : \mathbb{U} \mid f \in X \rightarrow Y \bullet f \in dom\ f \twoheadrightarrow ran\ f$

z_∩_↠_thm $\vdash \forall X : \mathbb{U}; Y : \mathbb{U}; f, g : \mathbb{U}$
$| f \in X \twoheadrightarrow Y$
$\bullet f \cap g \in dom\ (f \cap g) \twoheadrightarrow ran\ (f \cap g)$

z_∩_⤖_thm $\vdash \forall X : \mathbb{U}; Y : \mathbb{U}; f, g : \mathbb{U}$
$| f \in X \twoheadrightarrowtail Y$
$\bullet f \cap g \in dom\ (f \cap g) \twoheadrightarrowtail ran\ (f \cap g)$

z_→_diff_singleton_thm
$\vdash \forall X : \mathbb{U}; Y : \mathbb{U}$

$\bullet \forall f : X \rightarrow Y; x : \mathbb{U}; y : \mathbb{U}$
$\bullet (x, y) \in f \Rightarrow f \setminus \{(x, y)\} \in X \setminus \{x\} \rightarrow Y$

z_⤖_diff_singleton_thm
$\vdash \forall X : \mathbb{U}; Y : \mathbb{U}$
$\bullet \forall f : X \rightarrowtail\!\!\!\!\twoheadrightarrow Y; x : X; y : Y$
$\bullet (x, y) \in f \Rightarrow f \setminus \{(x, y)\} \in X \setminus \{x\} \rightarrowtail\!\!\!\!\twoheadrightarrow Y \setminus \{y\}$

z_singleton_app_thm
$\vdash \forall x : \mathbb{U}; y : \mathbb{U} \bullet \{(x, y)\}\ x = y$

z_empty_⇸_thm
$\vdash \forall X : \mathbb{U} \bullet (\exists f : \{\} \twoheadrightarrow X \bullet true) \Leftrightarrow X = \{\}$

z_→_empty_thm
$\vdash \forall X : \mathbb{U} \bullet (\exists f : X \rightarrow \{\} \bullet true) \Leftrightarrow X = \{\}$

9.1.5 The Theory z_language

9.1.5.1 Parents

$\mathbb{Z}$ *hol*

9.1.5.2 Children

z_language_ps

9.1.5.3 Notes

This theory is a cache theory; its contents have not been listed.

9.1.6 The Z Theory z_language_ps

9.1.6.1 Parents

z_language

9.1.6.2 Children

z_sets

9.1.6.3 Theorems

z_app_thm $\vdash \forall\ a : \mathbb{U}; f : \mathbb{U}; x : \mathbb{U}$
$\bullet\ (\forall\ f\_a : \mathbb{U} \mid (a, f\_a) \in f \bullet f\_a = x) \land (a, x) \in f$
$\Rightarrow f\ a = x$

z_sets_ext_thm
$\vdash \forall\ x : \mathbb{U}; y : \mathbb{U} \bullet x = y \Leftrightarrow (\forall\ z : \mathbb{U} \bullet z \in x \Leftrightarrow z \in y)$

z_∈_ℙ_thm1 $\vdash \forall\ t : \mathbb{U}; u : \mathbb{U} \bullet t \in \mathbb{P}\ u \Leftrightarrow (\forall\ z : \mathbb{U} \bullet z \in t \Rightarrow z \in u)$

z_∈_app_thm $\vdash \forall\ a : \mathbb{U}; x : \mathbb{U}; f : \mathbb{U}$
$\bullet\ (\exists\ f\_x : \mathbb{U}$
$\bullet\ a \in f\_x$
$\land (x, f\_x) \in f$
$\land (\forall\ f\_x1 : \mathbb{U} \bullet (x, f\_x1) \in f \Rightarrow f\_x1 = f\_x))$
$\Rightarrow a \in f\ x$

z_app_∈_thm $\vdash \forall\ a : \mathbb{U}; x : \mathbb{U}; f : \mathbb{U}$
$\bullet\ (\exists\ f\_x : \mathbb{U}$
$\bullet\ f\_x \in a$
$\land (x, f\_x) \in f$
$\land (\forall\ f\_x1 : \mathbb{U} \bullet (x, f\_x1) \in f \Rightarrow f\_x1 = f\_x))$
$\Rightarrow f\ x \in a$

9.1.7 The Z Theory z_library

9.1.7.1 Parents

z_sequences1 *z_arithmetic_tools* *z_bags*

9.1.8 The Z Theory z_numbers

9.1.8.1 Parents

z_functions

9.1.8.2 Children

z_reals z_sequences
z_numbers1 *z_arithmetic_tools*

9.1.8.3 Global Variables

$\mathbb{Z}$	$\mathbb{P}\ \mathbb{Z}$
$\mathbb{N}$	$\mathbb{P}\ \mathbb{Z}$
$(\sim\ \_)$	$\mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_ + \_)$	$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_ - \_)$	$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_ * \_)$	$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_ \leq \_)$	$\mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_ < \_)$	$\mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_ \geq \_)$	$\mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_ > \_)$	$\mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\mathbf{abs}\ \_)$	$\mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_\ \mathbf{div}\ \_)$	$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
$(\_\ \mathbf{mod}\ \_)$	$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{Z}$
$\mathbb{N}_{\mathbf{1}}$	$\mathbb{P}\ \mathbb{Z}$
succ	$\mathbb{Z} \leftrightarrow \mathbb{Z}$
iter[**X**]	$\mathbb{Z} \leftrightarrow (X \leftrightarrow X) \leftrightarrow X \leftrightarrow X$
$(\_\ ^{-}\ )[\mathbf{X}]$	$(X \leftrightarrow X) \times \mathbb{Z} \leftrightarrow X \leftrightarrow X$
$(\_ \mathrel{..} \_)$	$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{P}\ \mathbb{Z}$
$\mathbb{F}\ \mathbf{X}$	$\mathbb{P}\ (\mathbb{P}\ X)$
$\mathbb{F}_{\mathbf{1}}\ \mathbf{X}$	$\mathbb{P}\ (\mathbb{P}\ X)$
#[**X**]	$\mathbb{P}\ X \leftrightarrow \mathbb{Z}$
X ⇻ **Y**	$\mathbb{P}\ (X \leftrightarrow Y)$
X ⤘ **Y**	$\mathbb{P}\ (X \leftrightarrow Y)$
min	$\mathbb{P}\ \mathbb{Z} \leftrightarrow \mathbb{Z}$
max	$\mathbb{P}\ \mathbb{Z} \leftrightarrow \mathbb{Z}$

9.1.8.4 Fixity

fun 20 leftassoc

$(\_ \mathrel{..} \_)$

fun 30 leftassoc

$(\_ + \_)(\_ - \_)$

fun 40 leftassoc

$(\_\ div\ \_)$ $(\_\ mod\ \_)$ $(\_ * \_)$

fun 50 rightassoc
$(abs\ \_)\ (\sim\ \_)$

fun 70 rightassoc
$(\_\ ^{-}\ )$

gen 5 rightassoc
$(\_ \ \rightarrowtail\!\!\!\!\!\twoheadrightarrow \ \_)$ $(\_ \ \twoheadrightarrow\!\!\!\!\!\!\!+\ \ \_)$

gen 70 rightassoc
$(\mathbb{F}\ \_)$ $(\mathbb{F}_1\ \_)$

rel $(\_ < \_)(\_ > \_)(\_ \leq \_)(\_ \geq \_)$

9.1.8.5 Axioms

$\mathbb{N}$
$\sim \_$
$\_ + \_$ $\vdash ((\_ + \_) \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$
$\land (\sim \_) \in \mathbb{Z} \rightarrow \mathbb{Z}$
$\land \mathbb{N} \in \mathbb{P}\ \mathbb{Z})$
$\land (\forall\ i, j, k : \mathbb{Z}$
$\bullet\ i + j + k = i + (j + k)$
$\land\ i + j = j + i$
$\land\ i + \sim i = 0$
$\land\ i + 0 = i)$
$\land (\forall\ h : \mathbb{P}\ \mathbb{Z}$
$\bullet\ 1 \in h \land (\forall\ i, j : h \bullet i + j \in h \land \sim i \in h)$
$\Rightarrow h = \mathbb{Z})$
$\land \mathbb{N} = \bigcap \{s : \mathbb{P}\ \mathbb{Z} \mid 0 \in s \land \{i : s \bullet i + 1\} \subseteq s\}$
$\land \sim 1 \notin \mathbb{N}$

z′int_def
Z′Int $\vdash \ulcorner \forall\ i \bullet\ {}_{\mathbb{Z}}\ulcorner\ulcorner Z'Int\ (i + 1)\urcorner\urcorner = {}_{\mathbb{Z}}\ulcorner\ulcorner Z'Int\ i\urcorner + 1\urcorner\urcorner$
$\_ - \_$ $\vdash (\_ - \_) \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z} \land (\forall\ i, j : \mathbb{Z} \bullet i - j = i + \sim j)$
$\_ * \_$ $\vdash (\_ * \_) \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z}$
$\land (\forall\ i, j, k : \mathbb{Z}$
$\bullet\ i * j * k = i * (j * k)$
$\land\ i * j = j * i$
$\land\ i * (j + k) = i * j + i * k$
$\land\ 1 * i = i)$

$\_ \leq \_$
$\_ < \_$
$\_ \geq \_$
$\_ > \_$ $\vdash \{(\_ \leq \_), (\_ < \_), (\_ \geq \_), (\_ > \_)\} \subseteq \mathbb{Z} \leftrightarrow \mathbb{Z}$
$\land (\forall\ i, j : \mathbb{Z}$
$\bullet\ (i \leq j \Leftrightarrow j - i \in \mathbb{N})$
$\land\ (i < j \Leftrightarrow i + 1 \leq j)$
$\land\ (i \geq j \Leftrightarrow j \leq i)$
$\land\ (i > j \Leftrightarrow j < i))$

abs _ $\vdash (abs\ \_) \in \mathbb{Z} \rightarrow \mathbb{N} \land (abs\ \_) = (\sim\ \_) \oplus id\ \mathbb{N}$

_ div _

_ mod _ $\vdash \{(\_\ div\ \_), (\_\ mod\ \_)\} \subseteq \mathbb{Z} \times \mathbb{Z} \setminus \{0\} \rightarrow \mathbb{Z} \land (\forall\ i : \mathbb{Z}; j : \mathbb{Z} \setminus \{0\} \bullet i = i\ div\ j * j + i\ mod\ j \land 0 \leq i\ mod\ j \land i\ mod\ j < abs\ j)$

succ $\vdash succ \in \mathbb{N} \rightarrow \mathbb{N} \land (\forall\ n : \mathbb{N} \bullet succ\ n = n + 1)$

iter $\vdash [X](iter[X] \in \mathbb{Z} \rightarrow (X \leftrightarrow X) \rightarrow X \leftrightarrow X \land (\forall\ r : X \leftrightarrow X \bullet iter[X]\ 0\ r = id\ X \land (\forall\ k : \mathbb{N} \bullet iter[X]\ (k + 1)\ r = r \fatsemi iter[X]\ k\ r) \land (\forall\ k : \mathbb{N} \bullet iter[X]\ (\sim k)\ r = iter[X]\ k\ (r^{\sim}))))$

_ $^{-}$ $\vdash [X]((\_\ ^{-})[X] \in (X \leftrightarrow X) \times \mathbb{Z} \rightarrow X \leftrightarrow X \land (\forall\ r : X \leftrightarrow X; k : \mathbb{Z} \bullet (\_\ ^{-})[X]\ (r, k) = iter\ k\ r))$

_ .. _ $\vdash (\_ .. \_) \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{P}\ \mathbb{Z} \land (\forall\ x, y : \mathbb{Z} \bullet x .. y = \{k : \mathbb{Z} \mid x \leq k \land k \leq y\})$

$\vdash [X](\#[X] \in \mathbb{F}\ X \rightarrow \mathbb{N} \land (\forall\ S : \mathbb{F}\ X \bullet \#[X]\ S = (\mu\ n : \mathbb{N} \mid (\exists\ f : 1 .. n \rightarrowtail S \bullet ran\ f = S))))$

min $\vdash min \in \mathbb{P}_1\ \mathbb{Z} \nrightarrow \mathbb{Z} \land min = \{S : \mathbb{P}_1\ \mathbb{Z}; m : \mathbb{Z} \mid m \in S \land (\forall\ n : S \bullet m \leq n) \bullet S \mapsto m\}$

max $\vdash max \in \mathbb{P}_1\ \mathbb{Z} \nrightarrow \mathbb{Z} \land max = \{S : \mathbb{P}_1\ \mathbb{Z}; m : \mathbb{Z} \mid m \in S \land (\forall\ n : S \bullet m \geq n) \bullet S \mapsto m\}$

9.1.8.6 Definitions

$\mathbb{Z}$ $\vdash \mathbb{Z} = \mathbb{U}$

$\mathbb{N}_1$ $\vdash \mathbb{N}_1 = \mathbb{N} \setminus \{0\}$

$\mathbb{F}$ _ $\vdash [X](\mathbb{F}\ X = \{S : \mathbb{P}\ X \mid \exists\ n : \mathbb{N} \bullet \exists\ f : 1 .. n \rightarrow S \bullet ran\ f = S\})$

$\mathbb{F}_1$ _ $\vdash [X](\mathbb{F}_1\ X = \mathbb{F}\ X \setminus \{\varnothing\})$

_ ⇻ _ $\vdash [X, Y](X ⇻ Y = \{f : X \nrightarrow Y \mid dom\ f \in \mathbb{F}\ X\})$

_ ⤕ _ $\vdash [X, Y](X ⤕ Y = (X ⇻ Y) \cap (X ⤔ Y))$

9.1.8.7 Theorems

z_plus_comm_thm

$\vdash \forall\ i, j : \mathbb{U} \bullet i + j = j + i$

z_plus_assoc_thm

$\vdash \forall\ i, j, k : \mathbb{U} \bullet i + j + k = i + (j + k)$

z_plus_assoc_thm1
$\vdash \forall\ i, j, k : \mathbb{U} \bullet i + (j + k) = i + j + k$

z_plus_order_thm
$\vdash \forall\ i : \mathbb{U} \bullet \forall\ j, k : \mathbb{U} \bullet j + i = i + j \land i + j + k = i + (j + k) \land j + (i + k) = i + (j + k)$

z_plus0_thm $\vdash \forall\ i : \mathbb{U} \bullet i + 0 = i \land 0 + i = i$

z_plus_minus_thm
$\vdash \forall\ i : \mathbb{U} \bullet i + \sim i = 0 \land \sim i + i = 0$

z_ℕ_thm $\vdash \mathbb{N} = \bigcap \{s : \mathbb{U} \mid 0 \in s \land \{i : s \bullet i + 1\} \subseteq s\} \land \neg \sim 1 \in \mathbb{N}$

z_plus_cyclic_group_thm
$\vdash \forall\ h : \mathbb{U} \bullet 1 \in h \land (\forall\ i, j : h \bullet i + j \in h \land \sim i \in h) \Rightarrow h = \mathbb{U}$

z_int_homomorphism_thm
$\vdash \ulcorner \forall\ i\ j \bullet \ulcorner_Z \ulcorner Z'Int\ (i + j) \urcorner\urcorner = \ulcorner_Z \ulcorner Z'Int\ i \urcorner + \ulcorner Z'Int\ j \urcorner\urcorner\urcorner$

z_ℤ_induction_thm
$\vdash \ulcorner \forall\ p \bullet p\ \ulcorner_Z 1 \urcorner \land (\forall\ i \bullet p\ i \Rightarrow p\ \ulcorner_Z \sim i \urcorner) \land (\forall\ i\ j \bullet p\ i \land p\ j \Rightarrow p\ \ulcorner_Z i + j \urcorner) \Rightarrow (\forall\ m \bullet p\ m) \urcorner$

z_ℕ_plus1_thm
$\vdash \forall\ i : \mathbb{N} \bullet i + 1 \in \mathbb{N}$

z_0_ℕ_thm $\vdash 0 \in \mathbb{N}$

z_ℕ_induction_thm
$\vdash \ulcorner \forall\ p \bullet p\ \ulcorner_Z 0 \urcorner \land (\forall\ i \bullet i \in \ulcorner_Z \mathbb{N} \urcorner \land p\ i \Rightarrow p\ \ulcorner_Z i + 1 \urcorner) \Rightarrow (\forall\ m \bullet m \in \ulcorner_Z \mathbb{N} \urcorner \Rightarrow p\ m) \urcorner$

z_ℕ_plus_thm $\vdash \forall\ i, j : \mathbb{N} \bullet i + j \in \mathbb{N}$

z_ℤ_eq_thm $\vdash \forall\ i, j : \mathbb{U} \bullet i = j \Leftrightarrow i + \sim j = 0$

z_minus_thm $\vdash \forall\ i, j : \mathbb{U} \bullet \sim \sim i = i \land i + \sim i = 0 \land \sim i + i = 0 \land \sim (i + j) = \sim i + \sim j \land \sim 0 = 0$

z_minus_clauses
$\vdash \forall\ i : \mathbb{U} \bullet \sim \sim i = i \land \sim 0 = 0 \land i + \sim i = 0 \land \sim i + i = 0$

z_ℕ_cases_thm
$\vdash \forall\ i : \mathbb{N} \bullet i = 0 \lor (\exists\ j : \mathbb{N} \bullet i = j + 1)$

z_¬_ℕ_thm $\vdash \forall\ i : \mathbb{U} \bullet \neg\ i \in \mathbb{N} \Rightarrow \sim i \in \mathbb{N}$

z_ℤ_cases_thm
$\vdash \forall\ i : \mathbb{U} \bullet \exists\ j : \mathbb{N} \bullet i = j \lor i = \sim j$

z_ℕ_¬_plus1_thm

$\vdash \forall\ i : \mathbb{N} \bullet \neg\ i + 1 = 0$

z_ℤ_cases_thm1

$\vdash \forall\ i : \mathbb{U} \bullet i \in \mathbb{N} \lor (\exists\ j : \mathbb{N} \bullet i = \sim (j + 1))$

z_ℕ_¬_minus_thm

$\vdash \forall\ i : \mathbb{N} \bullet i = 0 \lor \neg \sim i \in \mathbb{N}$

z_plus_clauses

$\vdash \forall\ i, j, k : \mathbb{U}$
$\quad \bullet (i + k = j + k \Leftrightarrow i = j)$
$\quad \land (k + i = j + k \Leftrightarrow i = j)$
$\quad \land (i + k = k + j \Leftrightarrow i = j)$
$\quad \land (k + i = k + j \Leftrightarrow i = j)$
$\quad \land (i + k = k \Leftrightarrow i = 0)$
$\quad \land (k + i = k \Leftrightarrow i = 0)$
$\quad \land (k = k + j \Leftrightarrow j = 0)$
$\quad \land (k = j + k \Leftrightarrow j = 0)$
$\quad \land i + 0 = i$
$\quad \land 0 + i = i$
$\quad \land \neg$
$\quad 1 = 0$
$\quad \land \neg$
$\quad 0 = 1$

z_times_comm_thm

$\vdash \forall\ i, j : \mathbb{U} \bullet i * j = j * i$

z_times_assoc_thm

$\vdash \forall\ i, j, k : \mathbb{U} \bullet i * j * k = i * (j * k)$

z_times_assoc_thm1

$\vdash \forall\ i, j, k : \mathbb{U} \bullet i * (j * k) = i * j * k$

z_times_order_thm

$\vdash \forall\ i : \mathbb{U}$
$\quad \bullet \forall\ j, k : \mathbb{U}$
$\quad\quad \bullet j * i = i * j$
$\quad\quad \land i * j * k = i * (j * k)$
$\quad\quad \land j * (i * k) = i * (j * k)$

z_times1_thm $\vdash \forall\ i : \mathbb{U} \bullet i * 1 = i \land 1 * i = i$

z_times_plus_distrib_thm

$\vdash \forall\ i, j, k : \mathbb{U}$
$\quad \bullet i * (j + k) = i * j + i * k$
$\quad \land (i + j) * k = i * k + j * k$

z_times0_thm $\vdash \forall\ i : \mathbb{U} \bullet 0 * i = 0 \land i * 0 = 0$

z_minus_times_thm

$\vdash \forall\ i, j : \mathbb{U}$
$\quad \bullet \sim i * j = \sim (i * j)$
$\quad \land i * \sim j = \sim (i * j)$
$\quad \land \sim i * \sim j = i * j$

z_ℕ_times_thm

$\vdash \forall\ i, j : \mathbb{N} \bullet i * j \in \mathbb{N}$

z_times_eq_0_thm

$\vdash \forall\ i, j : \mathbb{U} \bullet i * j = 0 \Leftrightarrow i = 0 \lor j = 0$

z_times_clauses

$\vdash \forall\ i, j : \mathbb{U}$
$\quad \bullet 0 * i = 0 \land i * 0 = 0 \land i * 1 = i \land 1 * i = i$

z_≤_trans_thm
$\vdash \forall i, j, k : \mathbb{U} \mid i \leq j \land j \leq k \bullet i \leq k$
z_less_trans_thm
$\vdash \forall i, j, k : \mathbb{U} \mid i < j \land j < k \bullet i < k$
z_less_≤_trans_thm
$\vdash \forall i, j, k : \mathbb{U} \mid i < j \land j \leq k \bullet i < k$
z_≤_less_trans_thm
$\vdash \forall i, j, k : \mathbb{U} \mid i \leq j \land j < k \bullet i < k$
z_minus_ℕ_≤_thm
$\vdash \forall i : \mathbb{U}; j : \mathbb{N} \bullet i + \sim j \leq i$
z_≤_plus_ℕ_thm
$\vdash \forall i : \mathbb{U}; j : \mathbb{N} \bullet i \leq i + j$
z_≤_cases_thm
$\vdash \forall i, j : \mathbb{U} \bullet i \leq j \lor j \leq i$
z_≤_refl_thm $\vdash \forall i : \mathbb{U} \bullet i \leq i$
z_∈_ℕ_thm $\vdash \forall i : \mathbb{U} \bullet i \in \mathbb{N} \Leftrightarrow 0 \leq i$
z_≤_≤_0_thm $\vdash \forall i, j : \mathbb{U} \bullet i \leq j \Leftrightarrow i + \sim j \leq 0$
z_≤_antisym_thm
$\vdash \forall i, j : \mathbb{U} \mid i \leq j \land j \leq i \bullet i = j$
z_¬_less_thm $\vdash \forall i, j : \mathbb{U} \bullet \neg\ i < j \Leftrightarrow j \leq i$
z_¬_≤_thm $\vdash \forall i, j : \mathbb{U} \bullet \neg\ i \leq j \Leftrightarrow j < i$
z_≤_clauses $\vdash \forall i, j, k : \mathbb{U}$
$\bullet (i + k \leq j + k \Leftrightarrow i \leq j)$
$\land (k + i \leq j + k \Leftrightarrow i \leq j)$
$\land (i + k \leq k + j \Leftrightarrow i \leq j)$
$\land (k + i \leq k + j \Leftrightarrow i \leq j)$
$\land (i + k \leq k \Leftrightarrow i \leq 0)$
$\land (k + i \leq k \Leftrightarrow i \leq 0)$
$\land (i \leq k + i \Leftrightarrow 0 \leq k)$
$\land (i \leq i + k \Leftrightarrow 0 \leq k)$
$\land i \leq i$
$\land \neg$
$1 \leq 0$
$\land 0 \leq 1$
z_less_clauses
$\vdash \forall i, j, k : \mathbb{U}$
$\bullet (i + k < j + k \Leftrightarrow i < j)$
$\land (k + i < j + k \Leftrightarrow i < j)$
$\land (i + k < k + j \Leftrightarrow i < j)$
$\land (k + i < k + j \Leftrightarrow i < j)$
$\land (i + k < k \Leftrightarrow i < 0)$
$\land (k + i < k \Leftrightarrow i < 0)$
$\land (i < k + i \Leftrightarrow 0 < k)$
$\land (i < i + k \Leftrightarrow 0 < k)$
$\land \neg$
$i < i$
$\land 0 < 1$
$\land \neg$
$1 < 0$
z_less_irrefl_thm
$\vdash \forall i, j : \mathbb{U} \bullet \neg (i < j \land j < i)$

z_abs_thm $\vdash \forall\ i : \mathbb{N} \bullet abs\ i = i \land abs \sim i = i$

z_abs_minus_thm
$\vdash \forall\ i : \mathbb{U} \bullet abs \sim i = abs\ i$

z_abs_ℕ_thm $\vdash \forall\ i : \mathbb{U} \bullet abs\ i \in \mathbb{N}$

z_abs_times_thm
$\vdash \forall\ i, j : \mathbb{U} \bullet abs\ (i * j) = abs\ i * abs\ j$

z_abs_plus_thm
$\vdash \forall\ i, j : \mathbb{U} \bullet abs\ (i + j) \leq abs\ i + abs\ j$

z_abs_eq_0_thm
$\vdash \forall\ i : \mathbb{U} \bullet abs\ i = 0 \Leftrightarrow i = 0$

z_ℕ_abs_minus_thm
$\vdash \forall\ i, j : \mathbb{N} \mid j \leq i \bullet abs\ (i + \sim j) \leq i$

z_≤_induction_thm
$\vdash \ulcorner \forall\ j\ p$
$\bullet\ p\ j \land (\forall\ i \bullet \ulcorner_Z (j, i)\urcorner \in \ulcorner_Z (\_ \leq \_)\urcorner \land p\ i \Rightarrow p\ \ulcorner_Z i + 1\urcorner)$
$\Rightarrow (\forall\ m \bullet \ulcorner_Z (j, m)\urcorner \in \ulcorner_Z (\_ \leq \_)\urcorner \Rightarrow p\ m)\urcorner$

z_less_plus1_thm
$\vdash \forall\ m, n : \mathbb{U} \bullet m < n + 1 \Leftrightarrow m = n \lor m < n$

z_cov_induction_thm
$\vdash \ulcorner \forall\ j\ p$
$\bullet\ (\forall\ i$
$\bullet\ \ulcorner_Z (j, i)\urcorner \in \ulcorner_Z (\_ \leq \_)\urcorner$
$\land \ulcorner_Z \forall\ k : \mathbb{Z} \bullet j \leq k \land k < i \Rightarrow \ulcorner p\ k\urcorner\urcorner$
$\Rightarrow p\ i)$
$\Rightarrow (\forall\ i \bullet \ulcorner_Z (j, i)\urcorner \in \ulcorner_Z (\_ \leq \_)\urcorner \Rightarrow p\ i)\urcorner$

z_div_mod_unique_thm
$\vdash \forall\ i, j, d, r : \mathbb{U}$
$\mid \lnot$
$j = 0$
$\bullet\ i = d * j + r \land 0 \leq r \land r < abs\ j$
$\Leftrightarrow d = i\ div\ j \land r = i\ mod\ j$

z_≤_less_eq_thm
$\vdash \forall\ x, y : \mathbb{U} \bullet x \leq y \Leftrightarrow x < y \lor x = y$

z_∈_ℕ₁_thm $\vdash \forall\ x : \mathbb{U} \bullet x \in \mathbb{N}_1 \Leftrightarrow 0 < x$

z_𝔽_thm $\vdash \forall\ X : \mathbb{U}$
$\bullet\ \mathbb{F}\ X$
$= \{S : \mathbb{P}\ X$
$\mid \exists\ n : \mathbb{N} \bullet \exists\ f : 1 \mathinner{..} n \to S \bullet ran\ f = S\}$

z_𝔽₁_thm $\vdash \forall\ X : \mathbb{U} \bullet \mathbb{F}_1\ X = \mathbb{F}\ X \setminus \{\varnothing\}$

z_𝔽_empty_thm
$\vdash \mathbb{F}\ \{\} = \mathbb{P}\ \{\}$

9.1.9 The Z Theory z_numbers1

9.1.9.1 Parents

z_arithmetic_tools *z_numbers* *z_functions1*

9.1.9.2 Children

z_sequences1

9.1.9.3 Theorems

z_dot_dot_clauses

$\vdash \forall\ i,\ i1,\ i2,\ j1,\ j2 : \mathbb{U}$
$\bullet\ (i \in i1\ ..\ i2 \Leftrightarrow i1 \le i \land i \le i2)$
$\land\ (i1\ ..\ i2 = \{\} \Leftrightarrow i2 < i1)$
$\land\ (i1\ ..\ i2 \subseteq j1\ ..\ j2$
$\Leftrightarrow i2 < i1 \lor j1 \le i1 \land i2 \le j2)$

z_dot_dot_plus_thm

$\vdash \forall\ n,\ i1,\ i2 : \mathbb{U}$
$\bullet\ \{i : i1\ ..\ i2 \bullet i + n\} = i1 + n\ ..\ i2 + n$

z_less_cases_thm

$\vdash \forall\ i,\ j : \mathbb{U} \bullet i < j \lor i = j \lor j < i$

z_≤_≤_plus1_thm

$\vdash \forall\ i,\ j : \mathbb{U} \bullet i \le j \land j \le i + 1 \Leftrightarrow j = i \lor j = i + 1$

z_dot_dot_diff_thm

$\vdash \forall\ i : \mathbb{N} \bullet (1\ ..\ i + 1) \setminus \{i + 1\} = 1\ ..\ i$

z_dot_dot_∪_thm

$\vdash \forall\ i : \mathbb{N} \bullet (1\ ..\ i) \cup \{i + 1\} = 1\ ..\ i + 1$

z_dot_dot_∩_thm

$\vdash \forall\ i : \mathbb{N} \bullet (1\ ..\ i) \cap \{i + 1\} = \{\}$

z_empty_𝔽_thm

$\vdash [X](\{\} \in \mathbb{F}\ X)$

z_𝔽_∪_singleton_thm

$\vdash [X](\forall\ x : X;\ a : \mathbb{F}\ X \bullet a \cup \{x\} \in \mathbb{F}\ X)$

z_𝔽_thm1

$\vdash [X](\mathbb{F}\ X$
$= \bigcap$
$\{u : \mathbb{P}\ \mathbb{P}\ X$
$\mid \{\} \in u \land (\forall\ x : X;\ a : u \bullet a \cup \{x\} \in u)\})$

z_𝔽_induction_thm

$\vdash \ulcorner \forall\ X\ p$
$\bullet\ p\ \ulcorner_Z\{\}\urcorner$
$\land\ (\forall\ x\ a$
$\bullet\ p\ a \land a \in \ulcorner_Z \mathbb{F}\ X\urcorner \land x \in X \land \neg\ x \in a$
$\Rightarrow p\ \ulcorner_Z a \cup \{x\}\urcorner)$
$\Rightarrow (\forall\ a \bullet\ a \in \ulcorner_Z \mathbb{F}\ X\urcorner \Rightarrow p\ a)\urcorner$

z_𝔽_ℙ_thm

$\vdash [X](\mathbb{F}\ X = \mathbb{P}\ X \cap (\mathbb{F}\ \_))$

z_𝔽_size_thm

$\vdash \forall\ A : \mathbb{U};\ f : \mathbb{U};\ n : \mathbb{N}$
$\mid f \in 1\ ..\ n \rightarrowtail\!\!\!\!\!\twoheadrightarrow A$
$\bullet\ A \in (\mathbb{F}\ \_) \land \#\ A = n$

z_size_empty_thm

$\vdash \{\} \in (\mathbb{F}\ \_) \land \#\ \{\} = 0$

z_size_singleton_thm
$\vdash \forall\ x : \mathbb{U} \bullet \{x\} \in (\mathbb{F}\ \_) \land \#\ \{x\} = 1$

z_size_dot_dot_thm
$\vdash \forall\ n : \mathbb{N} \bullet 1\ ..\ n \in (\mathbb{F}\ \_) \land \#\ (1\ ..\ n) = n$

z_size_⇸_thm $\vdash \forall\ X : \mathbb{U};\ Y : \mathbb{U};\ f : \mathbb{U}$
$\mid f \in X \twoheadrightarrow\!\!\!\!\!\!\shortmid\ \ Y$
$\bullet\ f \in (\mathbb{F}\ \_) \land \#\ f = \#\ (dom\ f)$

z_size_seq_thm
$\vdash \forall\ X : \mathbb{U};\ f : \mathbb{U};\ n : \mathbb{N} \mid f \in 1\ ..\ n \rightarrow X \bullet \#\ f = n$

z_size_∪_singleton_thm
$\vdash \forall\ a : (\mathbb{F}\ \_);\ x : \mathbb{U} \mid \neg\ x \in a \bullet \#\ (a \cup \{x\}) = \#\ a + 1$

z_𝔽_∩_thm $\vdash \forall\ a, b : \mathbb{U} \mid a \in (\mathbb{F}\ \_) \lor b \in (\mathbb{F}\ \_) \bullet a \cap b \in (\mathbb{F}\ \_)$

z_𝔽_diff_thm $\vdash \forall\ a, b : \mathbb{U} \mid a \in (\mathbb{F}\ \_) \bullet a \setminus b \in (\mathbb{F}\ \_)$

z_⊆_𝔽_thm $\vdash \forall\ a : (\mathbb{F}\ \_);\ b : \mathbb{U} \mid b \subseteq a \bullet b \in (\mathbb{F}\ \_)$

z_size_∪_thm $\vdash \forall\ a, b : (\mathbb{F}\ \_)$
$\bullet\ a \cup b \in (\mathbb{F}\ \_) \land \#\ (a \cup b) + \#\ (a \cap b) = \#\ a + \#\ b$

z_⋃_𝔽_thm $\vdash \forall\ u : \mathbb{F}\ (\mathbb{F}\ \_) \bullet \bigcup\ u \in (\mathbb{F}\ \_)$

z_size_diff_thm
$\vdash \forall\ a : (\mathbb{F}\ \_);\ b : \mathbb{U}$
$\bullet\ a \setminus b \in (\mathbb{F}\ \_) \land \#\ (a \setminus b) + \#\ (a \cap b) = \#\ a$

z_size_ℕ_thm $\vdash \forall\ a : (\mathbb{F}\ \_) \bullet \#\ a \in \mathbb{N}$

z_𝔽_size_thm1
$\vdash \forall\ a : (\mathbb{F}\ \_) \bullet \exists\ f : 1\ ..\ \#\ a \rightarrowtail\!\!\!\!\!\twoheadrightarrow a \bullet true$

z_size_mono_thm
$\vdash \forall\ a : (\mathbb{F}\ \_);\ b : \mathbb{U} \mid b \subseteq a \bullet \#\ b \leq \#\ a$

z_size_∪_≤_thm
$\vdash \forall\ a, b : (\mathbb{F}\ \_) \bullet \#\ (a \cup b) \leq \#\ a + \#\ b$

z_size_eq_thm
$\vdash \forall\ a, b : (\mathbb{F}\ \_) \mid a \subseteq b \land \#\ a = \#\ b \bullet a = b$

z_size_0_thm $\vdash \forall\ a : (\mathbb{F}\ \_) \bullet \#\ a = 0 \Leftrightarrow a = \{\}$

z_size_1_thm $\vdash \forall\ a : (\mathbb{F}\ \_) \bullet \#\ a = 1 \Leftrightarrow (\exists\ x : \mathbb{U} \bullet a = \{x\})$

z_size_pair_thm
$\vdash \forall\ x, y : \mathbb{U} \mid \neg\ x = y \bullet \{x, y\} \in (\mathbb{F}\ \_) \land \#\ \{x, y\} = 2$

z_size_2_thm $\vdash \forall\ a : (\mathbb{F}\ \_)$
$\bullet\ \#\ a = 2 \Leftrightarrow (\exists\ x, y : \mathbb{U} \bullet \neg\ x = y \land a = \{x, y\})$

z_size_×_thm $\vdash \forall\ a : (\mathbb{F}\ \_);\ b : (\mathbb{F}\ \_)$
$\bullet\ a \times b \in (\mathbb{F}\ \_) \land \#\ (a \times b) = \#\ a * \#\ b$

z_size_≤_1_thm
$\vdash \forall\ a : (\mathbb{F}\ \_) \mid \#\ a \leq 1 \bullet a = \{\} \lor (\exists\ x : \mathbb{U} \bullet a = \{x\})$

z_size_dot_dot_thm1
$\vdash \forall\ i, j : \mathbb{Z}$
$\bullet\ i\ ..\ j \in (\mathbb{F}\ \_)$
$\land\ (i \leq j \Rightarrow \#\ (i\ ..\ j) = j + \sim i + 1)$
$\land\ (j < i \Rightarrow \#\ (i\ ..\ j) = 0)$

z_pigeon_hole_thm
$\vdash \forall\ u : \mathbb{F}\ (\mathbb{F}\ \_) \mid \#\ (\bigcup\ u) > \#\ u \bullet \exists\ a : u \bullet \#\ a > 1$

z_div_thm $\vdash \forall\ i, j, k : \mathbb{Z}$
$\mid \neg$
$j = 0$
$\bullet\ i\ div\ j = k$

$\Leftrightarrow (\exists\ m : \mathbb{Z} \bullet i = k * j + m \land 0 \le m \land m < abs\ j)$

z_mod_thm $\vdash \forall\ i, j, k : \mathbb{Z}$
$\mid \neg$
$j = 0$
$\bullet\ i\ mod\ j = k$
$\Leftrightarrow (\exists\ d : \mathbb{Z} \bullet i = d * j + k \land 0 \le k \land k < abs\ j)$

z_abs_pos_thm
$\vdash \forall\ i : \mathbb{Z} \mid 0 < i \bullet abs\ i = i \land abs \sim i = i$

z_abs_neg_thm
$\vdash \forall\ i : \mathbb{Z} \mid i < 0 \bullet abs\ i = \sim i \land abs \sim i = \sim i$

z_abs_≤_times_thm
$\vdash \forall\ i, j : \mathbb{Z} \mid \neg\ i = 0 \land \neg\ j = 0 \bullet abs\ j \le abs\ (i * j)$

z_abs_0_less_thm
$\vdash \forall\ i : \mathbb{Z} \mid \neg\ i = 0 \bullet 0 < abs\ i$

z_0_less_times_thm
$\vdash \forall\ i, j : \mathbb{Z}$
$\bullet\ 0 < i * j \Leftrightarrow 0 < i \land 0 < j \lor i < 0 \land j < 0$

z_times_less_0_thm
$\vdash \forall\ i, j : \mathbb{Z}$
$\bullet\ i * j < 0 \Leftrightarrow 0 < i \land j < 0 \lor i < 0 \land 0 < j$

z_∈_succ_thm $\vdash \ulcorner \forall\ i\ j$
$\bullet\ \ulcorner_Z (i, j) \urcorner \in \ulcorner_Z succ \urcorner$
$\Leftrightarrow \ulcorner_Z (0, i) \urcorner \in \ulcorner_Z (\_ \le \_) \urcorner \land j = \ulcorner_Z i + 1 \urcorner \urcorner$

z_succ$^{\mathbf{0}}$_thm
$\vdash succ\ ^{0} = id\ \mathbb{Z}$

z_succ$^{\mathbf{n}}$_thm
$\vdash \forall\ n : \mathbb{Z} \mid 1 \le n \bullet succ\ ^{n} = \{m : \mathbb{N} \bullet m \mapsto m + n\}$

z_succ$^{\mathbf{minus_n}}$_thm
$\vdash \forall\ n : \mathbb{N} \mid 1 \le n \bullet succ\ ^{\sim n} = \{m : \mathbb{N} \bullet m + n \mapsto m\}$

9.1.10 The Z Theory z_reals

9.1.10.1 Parents

$\mathbb{R}$ *z_numbers*

9.1.10.2 Global Variables

$\mathbb{R}$	$\mathbb{P}\ \mathbb{R}$
$(\mathbf{abs_R}\ \_)$	$\mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ /_\mathbf{R}\ \_)$	$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ *_\mathbf{R}\ \_)$	$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ +_\mathbf{R}\ \_)$	$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{R}$
$(\sim_\mathbf{R}\ \_)$	$\mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ \leq_\mathbf{R}\ \_)$	$\mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ <_\mathbf{R}\ \_)$	$\mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ -_\mathbf{R}\ \_)$	$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ \geq_\mathbf{R}\ \_)$	$\mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ >_\mathbf{R}\ \_)$	$\mathbb{R} \leftrightarrow \mathbb{R}$
real	$\mathbb{Z} \leftrightarrow \mathbb{R}$
$(\_\ /_\mathbf{Z}\ \_)$	$\mathbb{Z} \times \mathbb{Z} \leftrightarrow \mathbb{R}$
$(\_\ \hat{}_\mathbf{Z}\ \_)$	$\mathbb{R} \times \mathbb{Z} \leftrightarrow \mathbb{R}$
$(\_\ \ldotp\ldotp_\mathbf{R}\ \_)$	$\mathbb{R} \times \mathbb{R} \leftrightarrow \mathbb{P}\ \mathbb{R}$
$(\_\ \mathbf{lb_R}\ \_)$	$\mathbb{R} \leftrightarrow \mathbb{P}\ \mathbb{R}$
$\mathbf{glb_R}$	$\mathbb{P}\ \mathbb{R} \leftrightarrow \mathbb{R}$
$(\_\ \mathbf{ub_R}\ \_)$	$\mathbb{R} \leftrightarrow \mathbb{P}\ \mathbb{R}$
$\mathbf{lub_R}$	$\mathbb{P}\ \mathbb{R} \leftrightarrow \mathbb{R}$
Z'Float	$\mathbb{U}$

9.1.10.3 Fixity

fun 20 leftassoc

$(\_\ \ldotp\ldotp_R\ \_)$

fun 30 leftassoc

$(\_\ +_R\ \_)$ $(\_\ -_R\ \_)$

fun 40 leftassoc

$(\_\ *_R\ \_)$ $(\_\ /_R\ \_)$ $(\_\ /_Z\ \_)$

fun 50 rightassoc

$(abs_R\ \_)$ $(\sim_R\ \_)$

fun 60 rightassoc

$(\_\ \hat{}_Z\ \_)$

rel

$(\_\ lb_R\ \_)$ $(\_\ <_R\ \_)$ $(\_\ \leq_R\ \_)$
$(\_\ ub_R\ \_)$ $(\_\ >_R\ \_)$ $(\_\ \geq_R\ \_)$

9.1.10.4 Axioms

abs$_{\mathbf{R}}$ _
_ $/_{\mathbf{R}}$ _
_ $*_{\mathbf{R}}$ _
_ $+_{\mathbf{R}}$ _
$\sim_{\mathbf{R}}$ _
_ $\leq_{\mathbf{R}}$ _
_ $<_{\mathbf{R}}$ _

$$\begin{array}{l}
\vdash ((_ <_R _) \in \mathbb{R} \leftrightarrow \mathbb{R} \\
\quad \land (_ \leq_R _) \in \mathbb{R} \leftrightarrow \mathbb{R} \\
\quad \land (\sim_R _) \in \mathbb{R} \rightarrow \mathbb{R} \\
\quad \land (_ +_R _) \in \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \\
\quad \land (_ *_R _) \in \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \\
\quad \land (_ /_R _) \in \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \\
\quad \land (abs_R _) \in \mathbb{R} \rightarrow \mathbb{R}) \\
\quad \land (\forall x, y : \mathbb{R} \bullet x <_R y \Leftrightarrow \ulcorner x < y \urcorner) \\
\quad \land (\forall x, y : \mathbb{R} \bullet x \leq_R y \Leftrightarrow \ulcorner x \leq y \urcorner) \\
\quad \land (\forall x : \mathbb{R} \bullet \sim_R x = \ulcorner \sim x \urcorner) \\
\quad \land (\forall x, y : \mathbb{R} \bullet x +_R y = \ulcorner x + y \urcorner) \\
\quad \land (\forall x, y : \mathbb{R} \bullet x *_R y = \ulcorner x * y \urcorner) \\
\quad \land (\forall x, y : \mathbb{R} \bullet x /_R y = \ulcorner x / y \urcorner) \\
\quad \land (\forall x : \mathbb{R} \bullet abs_R\ x = \ulcorner Abs\ x \urcorner)
\end{array}$$

_ $-_{\mathbf{R}}$ _
_ $\geq_{\mathbf{R}}$ _
_ $>_{\mathbf{R}}$ _

$$\begin{array}{l}
\vdash ((_ >_R _) \in \mathbb{R} \leftrightarrow \mathbb{R} \\
\quad \land (_ \geq_R _) \in \mathbb{R} \leftrightarrow \mathbb{R} \\
\quad \land (_ -_R _) \in \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}) \\
\quad \land (\forall x, y : \mathbb{R} \bullet x >_R y \Leftrightarrow y <_R x) \\
\quad \land (\forall x, y : \mathbb{R} \bullet x \geq_R y \Leftrightarrow y \leq_R x) \\
\quad \land (\forall x, y : \mathbb{R} \bullet x -_R y = x +_R \sim_R y)
\end{array}$$

real

$$\begin{array}{l}
\vdash real \in \mathbb{Z} \rightarrow \mathbb{R} \\
\quad \land real\ 1 = \ulcorner 1. \urcorner \\
\quad \land (\forall i : \mathbb{Z} \bullet real\ (\sim i) = \sim_R real\ i) \\
\quad \land (\forall i, j : \mathbb{Z} \bullet real\ (i + j) = real\ i +_R real\ j)
\end{array}$$

_ $/_{\mathbf{Z}}$ _

$$\begin{array}{l}
\vdash (_ /_Z _) \in \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{R} \\
\quad \land (\forall i, j : \mathbb{Z} \bullet i /_Z j = real\ i /_R real\ j)
\end{array}$$

_ $\hat{}_{\mathbf{Z}}$ _

$$\begin{array}{l}
\vdash (_ \hat{}_Z _) \in \mathbb{R} \times \mathbb{Z} \rightarrow \mathbb{R} \\
\quad \land \ulcorner \forall x\ m \bullet \ulcorner_Z x\ \hat{}_Z \ulcorner Z'Int\ m \urcorner \urcorner = x\ \hat{}\ m \urcorner \\
\quad \land \ulcorner \forall x\ m \\
\quad \bullet \ulcorner_Z x\ \hat{}_Z (\sim \ulcorner Z'Int\ (m + 1) \urcorner) \urcorner = 1. / x\ \hat{}\ (m + 1) \urcorner
\end{array}$$

_ $.._{\mathbf{R}}$ _

$$\begin{array}{l}
\vdash (_ .._R _) \in \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{P}\ \mathbb{R} \\
\quad \land (\forall x, y : \mathbb{R} \\
\qquad \bullet x .._R y = \{t : \mathbb{R} \mid x \leq_R t \land t \leq_R y\})
\end{array}$$

_ **lb**$_{\mathbf{R}}$ _

$$\begin{array}{l}
\vdash (_ lb_R _) \in \mathbb{R} \leftrightarrow \mathbb{P}\ \mathbb{R} \\
\quad \land (\forall r : \mathbb{R}; sr : \mathbb{P}\ \mathbb{R} \\
\qquad \bullet r\ lb_R\ sr \Leftrightarrow (\forall x : sr \bullet r \leq_R x))
\end{array}$$

glb$_{\mathbf{R}}$

$$\begin{array}{l}
\vdash glb_R \in \mathbb{P}\ \mathbb{R} \twoheadrightarrow \mathbb{R} \\
\quad \land (\forall sr : \mathbb{P}\ \mathbb{R}; glb : \mathbb{R} \\
\qquad \bullet sr \mapsto glb \in glb_R \\
\qquad \quad \Leftrightarrow glb\ lb_R\ sr \\
\qquad \qquad \land (\forall lb : \mathbb{R} \mid lb\ lb_R\ sr \bullet lb \leq_R glb))
\end{array}$$

_ **ub**$_{\mathbf{R}}$ _

$$\vdash (\_ ub_R \_) \in \mathbb{R} \leftrightarrow \mathbb{P}\ \mathbb{R}$$

$\land (\forall\ r : \mathbb{R};\ sr : \mathbb{P}\ \mathbb{R}$
$\bullet\ r\ ub_R\ sr \Leftrightarrow (\forall\ x : sr \bullet r \geq_R x))$

lub$_\mathbf{R}$ $\vdash lub_R \in \mathbb{P}\ \mathbb{R} \nrightarrow \mathbb{R}$
$\land (\forall\ sr : \mathbb{P}\ \mathbb{R};\ lub : \mathbb{R}$
$\bullet\ sr \mapsto lub \in lub_R$
$\Leftrightarrow lub\ ub_R\ sr$
$\land (\forall\ ub : \mathbb{R} \mid ub\ ub_R\ sr \bullet ub \geq_R lub))$

9.1.10.5 Definitions

$\mathbb{R}$ $\vdash \mathbb{R} = \mathbb{U}$

Z′Float $\vdash \ulcorner\forall\ m\ p\ e$
$\bullet\ {}_{\mathrm{Z}}\ulcorner\ulcorner Z'Float\ m\ p\ e\urcorner\urcorner$
$= {}_{\mathrm{Z}}\ulcorner real\ m *_R real\ 10\ \hat{}_Z\ (e + \sim p)\urcorner\urcorner$

9.1.10.6 Theorems

z_ℝ_unbounded_below_thm
$\vdash \forall\ x : \mathbb{R} \bullet \exists\ y : \mathbb{R} \bullet y <_R x$

z_ℝ_unbounded_above_thm
$\vdash \forall\ x : \mathbb{R} \bullet \exists\ y : \mathbb{R} \bullet x <_R y$

z_ℝ_less_irrefl_thm
$\vdash \forall\ x : \mathbb{R} \bullet \neg\ x <_R x$

z_ℝ_less_antisym_thm
$\vdash \forall\ x,\ y : \mathbb{R} \bullet \neg\ (x <_R y \land y <_R x)$

z_ℝ_less_trans_thm
$\vdash \forall\ x,\ y,\ z : \mathbb{R} \bullet x <_R y \land y <_R z \Rightarrow x <_R z$

z_ℝ_less_cases_thm
$\vdash \forall\ x,\ y : \mathbb{R} \bullet x <_R y \lor x = y \lor y <_R x$

z_ℝ_≤_cases_thm
$\vdash \forall\ x,\ y : \mathbb{R} \bullet x \leq_R y \lor y \leq_R x$

z_ℝ_≤_less_cases_thm
$\vdash \forall\ x,\ y : \mathbb{R} \bullet x \leq_R y \lor y <_R x$

z_ℝ_eq_≤_thm $\vdash \forall\ x,\ y : \mathbb{R} \bullet x = y \Leftrightarrow x \leq_R y \land y \leq_R x$

z_ℝ_≤_antisym_thm
$\vdash \forall\ x,\ y : \mathbb{R} \bullet x \leq_R y \land y \leq_R x \Rightarrow x = y$

z_ℝ_less_≤_trans_thm
$\vdash \forall\ x,\ y,\ z : \mathbb{R} \bullet x <_R y \land y \leq_R z \Rightarrow x <_R z$

z_ℝ_≤_less_trans_thm
$\vdash \forall\ x,\ y,\ z : \mathbb{R} \bullet x \leq_R y \land y <_R z \Rightarrow x <_R z$

z_ℝ_≤_refl_thm
$\vdash \forall\ x : \mathbb{R} \bullet x \leq_R x$

z_ℝ_≤_trans_thm
$\vdash \forall\ x,\ y,\ z : \mathbb{R} \bullet x \leq_R y \land y \leq_R z \Rightarrow x \leq_R z$

z_ℝ_≤_¬_less_thm
$\vdash \forall\ x,\ y : \mathbb{R} \bullet x \leq_R y \Leftrightarrow \neg\ y <_R x$

z_ℝ_¬_≤_less_thm
$\vdash \forall\ x,\ y : \mathbb{R} \bullet \neg\ x \leq_R y \Leftrightarrow y <_R x$

z_ℝ_less_¬_eq_thm
$\vdash \forall\ x,\ y : \mathbb{R} \bullet x <_R y \Rightarrow \neg\ x = y$

z_ℝ_less_dense_thm

$\vdash \forall x, y : \mathbb{R} \bullet x <_R y \Rightarrow (\exists z : \mathbb{R} \bullet x <_R z \land z <_R y)$

z_ℝ_complete_thm

$\vdash \forall A : \mathbb{P}\ \mathbb{R}$
$\bullet \neg A = \{\} \land (\exists b : \mathbb{R} \bullet \forall x : \mathbb{R} \bullet x \in A \Rightarrow x \leq_R b)$
$\Rightarrow (\exists s : \mathbb{R}$
$\bullet (\forall x : \mathbb{R} \bullet x \in A \Rightarrow x \leq_R s)$
$\land (\forall b : \mathbb{R}$
$\bullet (\forall x : \mathbb{R} \bullet x \in A \Rightarrow x \leq_R b) \Rightarrow s \leq_R b))$

z_ℝ_plus_assoc_thm

$\vdash \forall x, y, z : \mathbb{R} \bullet x +_R y +_R z = x +_R (y +_R z)$

z_ℝ_plus_assoc_thm1

$\vdash \forall x, y, z : \mathbb{R} \bullet x +_R (y +_R z) = x +_R y +_R z$

z_ℝ_plus_comm_thm

$\vdash \forall x, y : \mathbb{R} \bullet x +_R y = y +_R x$

z_ℝ_plus_unit_thm

$\vdash \forall x : \mathbb{R} \bullet x +_R \mathit{real}\ 0 = x$

z_ℝ_plus_mono_thm

$\vdash \forall x, y, z : \mathbb{R} \bullet y <_R z \Rightarrow x +_R y <_R x +_R z$

z_ℝ_plus_mono_thm1

$\vdash \forall x, y, z : \mathbb{R} \bullet y <_R z \Rightarrow y +_R x <_R z +_R x$

z_ℝ_plus_mono_thm2

$\vdash \forall x, y, s, t : \mathbb{R}$
$\bullet x <_R y \land s <_R t \Rightarrow x +_R s <_R y +_R t$

z_ℝ_plus_0_thm

$\vdash \forall x : \mathbb{R} \bullet x +_R \mathit{real}\ 0 = x \land \mathit{real}\ 0 +_R x = x$

z_ℝ_plus_order_thm

$\vdash \forall x, y, z : \mathbb{R}$
$\bullet y +_R x = x +_R y$
$\land x +_R y +_R z = x +_R (y +_R z)$
$\land y +_R (x +_R z) = x +_R (y +_R z)$

z_ℝ_plus_minus_thm

$\vdash \forall x : \mathbb{R} \bullet x +_R \sim_R x = \mathit{real}\ 0 \land \sim_R x +_R x = \mathit{real}\ 0$

z_ℝ_eq_thm $\vdash \forall x, y : \mathbb{R} \bullet x = y \Leftrightarrow x +_R \sim_R y = \mathit{real}\ 0$

z_ℝ_minus_clauses

$\vdash \forall x, y : \mathbb{R}$
$\bullet \sim_R \sim_R x = x$
$\land x +_R \sim_R x = \mathit{real}\ 0$
$\land \sim_R x +_R x = \mathit{real}\ 0$
$\land \sim_R (x +_R y) = \sim_R x +_R \sim_R y$
$\land \sim_R \mathit{real}\ 0 = \mathit{real}\ 0$

z_ℝ_minus_eq_thm

$\vdash \forall x, y : \mathbb{R} \bullet \sim_R x = \sim_R y \Leftrightarrow x = y$

z_ℝ_plus_clauses

$\vdash \forall x, y, z : \mathbb{R}$
$\bullet (x +_R z = y +_R z \Leftrightarrow x = y)$
$\land (z +_R x = y +_R z \Leftrightarrow x = y)$
$\land (x +_R z = z +_R y \Leftrightarrow x = y)$
$\land (z +_R x = z +_R y \Leftrightarrow x = y)$
$\land (x +_R z = z \Leftrightarrow x = \mathit{real}\ 0)$
$\land (z +_R x = z \Leftrightarrow x = \mathit{real}\ 0)$
$\land (z = z +_R y \Leftrightarrow y = \mathit{real}\ 0)$

$\land (z = y +_R z \Leftrightarrow y = real\ 0)$
$\land x +_R real\ 0 = x$
$\land real\ 0 +_R x = x$
$\land \neg$
$real\ 1 = real\ 0$
$\land \neg$
$real\ 0 = real\ 1$

z_ℝ_less_clauses

$\vdash \forall x, y, z : \mathbb{R}$
$\bullet (x +_R z <_R y +_R z \Leftrightarrow x <_R y)$
$\land (z +_R x <_R y +_R z \Leftrightarrow x <_R y)$
$\land (x +_R z <_R z +_R y \Leftrightarrow x <_R y)$
$\land (z +_R x <_R z +_R y \Leftrightarrow x <_R y)$
$\land (x +_R z <_R z \Leftrightarrow x <_R real\ 0)$
$\land (z +_R x <_R z \Leftrightarrow x <_R real\ 0)$
$\land (x <_R z +_R x \Leftrightarrow real\ 0 <_R z)$
$\land (x <_R x +_R z \Leftrightarrow real\ 0 <_R z)$
$\land \neg$
$x <_R x$
$\land real\ 0 <_R real\ 1$
$\land \neg$
$real\ 1 <_R real\ 0$

z_ℝ_≤_clauses

$\vdash \forall x, y, z : \mathbb{R}$
$\bullet (x +_R z \leq_R y +_R z \Leftrightarrow x \leq_R y)$
$\land (z +_R x \leq_R y +_R z \Leftrightarrow x \leq_R y)$
$\land (x +_R z \leq_R z +_R y \Leftrightarrow x \leq_R y)$
$\land (z +_R x \leq_R z +_R y \Leftrightarrow x \leq_R y)$
$\land (x +_R z \leq_R z \Leftrightarrow x \leq_R real\ 0)$
$\land (z +_R x \leq_R z \Leftrightarrow x \leq_R real\ 0)$
$\land (x \leq_R z +_R x \Leftrightarrow real\ 0 \leq_R z)$
$\land (x \leq_R x +_R z \Leftrightarrow real\ 0 \leq_R z)$
$\land x \leq_R x$
$\land real\ 0 \leq_R real\ 1$
$\land \neg$
$real\ 1 \leq_R real\ 0$

z_ℝ_times_assoc_thm

$\vdash \forall x, y, z : \mathbb{R} \bullet x *_R y *_R z = x *_R (y *_R z)$

z_ℝ_times_comm_thm

$\vdash \forall x, y : \mathbb{R} \bullet x *_R y = y *_R x$

z_ℝ_times_unit_thm

$\vdash \forall x : \mathbb{R} \bullet x *_R real\ 1 = x$

z_ℝ_0_less_0_less_times_thm

$\vdash \forall x, y : \mathbb{R}$
$\bullet real\ 0 <_R x \land real\ 0 <_R y \Rightarrow real\ 0 <_R x *_R y$

z_ℝ_times_assoc_thm1

$\vdash \forall x, y, z : \mathbb{R} \bullet x *_R (y *_R z) = x *_R y *_R z$

z_ℝ_times_plus_distrib_thm

$\vdash \forall x, y, z : \mathbb{R}$
$\bullet x *_R (y +_R z) = x *_R y +_R x *_R z$
$\land (x +_R y) *_R z = x *_R z +_R y *_R z$

z_ℝ_times_order_thm
$\vdash \forall\ x,\ y,\ z : \mathbb{R}$
$\bullet\ y *_R x = x *_R y$
$\land\ x *_R y *_R z = x *_R (y *_R z)$
$\land\ y *_R (x *_R z) = x *_R (y *_R z)$

z_ℝ_times_clauses
$\vdash \forall\ x : \mathbb{R}$
$\bullet\ real\ 0 *_R x = real\ 0$
$\land\ x *_R real\ 0 = real\ 0$
$\land\ x *_R real\ 1 = x$
$\land\ real\ 1 *_R x = x$

z_ℝ_over_clauses
$\vdash (\forall\ y,\ z : \mathbb{R} \bullet \neg\ z = real\ 0 \Rightarrow y *_R z\ /_R\ z = y)$
$\land\ (\forall\ x,\ y,\ z : \mathbb{R}$
$\bullet \neg\ z = real\ 0 \Rightarrow x *_R y\ /_R\ z = x *_R (y\ /_R\ z))$

z_float_thm
$\vdash \forall\ m,\ p,\ e : \mathbb{Z}$
$\bullet\ \ulcorner Z'Float\ m\ p\ e \urcorner$
$= real\ m *_R real\ 10\ \hat{}_Z\ (e + \sim p)$

9.1.11 The Z Theory z_relations

9.1.11.1 Parents

z_sets

9.1.11.2 Children

z_functions

9.1.11.3 Global Variables

$(\_ \mapsto \_)[\mathbf{X}, \mathbf{Y}]$
$X \times Y \leftrightarrow X \times Y$

$\mathbf{ran}[\mathbf{X}, \mathbf{Y}]$ $(X \leftrightarrow Y) \leftrightarrow \mathbb{P}\ Y$

$\mathbf{dom}[\mathbf{X}, \mathbf{Y}]$ $(X \leftrightarrow Y) \leftrightarrow \mathbb{P}\ X$

$\mathbf{id\ X}$ $X \leftrightarrow X$

$(\_ \circ \_)[\mathbf{X}, \mathbf{Y}, \mathbf{Z}]$
$(Y \leftrightarrow Z) \times (X \leftrightarrow Y) \leftrightarrow X \leftrightarrow Z$

$(\_ \fatsemi \_)[\mathbf{X}, \mathbf{Y}, \mathbf{Z}]$
$(X \leftrightarrow Y) \times (Y \leftrightarrow Z) \leftrightarrow X \leftrightarrow Z$

$(\_ \rhd \_)[\mathbf{X}, \mathbf{Y}]$
$(X \leftrightarrow Y) \times \mathbb{P}\ Y \leftrightarrow X \leftrightarrow Y$

$(\_ \lhd \_)[\mathbf{X}, \mathbf{Y}]$
$\mathbb{P}\ X \times (X \leftrightarrow Y) \leftrightarrow X \leftrightarrow Y$

$(\_ \ntriangleright \_)[\mathbf{X}, \mathbf{Y}]$
$(X \leftrightarrow Y) \times \mathbb{P}\ Y \leftrightarrow X \leftrightarrow Y$

$(\_ \ntriangleleft \_)[\mathbf{X}, \mathbf{Y}]$
$\mathbb{P}\ X \times (X \leftrightarrow Y) \leftrightarrow X \leftrightarrow Y$

$(\_ ^\sim)[\mathbf{X}, \mathbf{Y}]$
$(X \leftrightarrow Y) \leftrightarrow Y \leftrightarrow X$

$(\_ \llparenthesis \_ \rrparenthesis)[\mathbf{X}, \mathbf{Y}]$
$(X \leftrightarrow Y) \times \mathbb{P}\ X \leftrightarrow \mathbb{P}\ Y$

$(\_ ^+)[\mathbf{X}]$ $(X \leftrightarrow X) \leftrightarrow X \leftrightarrow X$

$(\_ ^*)[\mathbf{X}]$ $(X \leftrightarrow X) \leftrightarrow X \leftrightarrow X$

$(\_ \oplus \_)[\mathbf{X}, \mathbf{Y}]$
$(X \leftrightarrow Y) \times (X \leftrightarrow Y) \leftrightarrow X \leftrightarrow Y$

9.1.11.4 Fixity

fun 10 leftassoc
$(\_ \mapsto \_)$

fun 40 leftassoc
$(\_ \circ \_)$ $(\_ \fatsemi \_)$

fun 50 leftassoc
$(\_ \oplus \_)$

fun 60 leftassoc

$(\_ \unrhd \_)(\_ \rhd \_)$

fun 65 rightassoc

$(\_ \unlhd \_)(\_ \lhd \_)$

fun 70 rightassoc

$(\_ \limg \_ \rimg)$ $(\_ ^{*})$ $(\_ ^{+})$ $(\_ ^{\sim})$

gen 70 rightassoc

$(id\ \_)$

9.1.11.5 Axioms

$\_ \mapsto \_$

$$\vdash [X, Y]((\_ \mapsto \_)[X, Y] \in X \times Y \to X \times Y \land (\forall x : X; y : Y \bullet (\_ \mapsto \_)[X, Y]\ (x, y) = (x, y)))$$

ran
dom

$$\vdash [X, Y]((dom[X, Y] \in (X \leftrightarrow Y) \to \mathbb{P}\ X \land ran[X, Y] \in (X \leftrightarrow Y) \to \mathbb{P}\ Y) \land (\forall R : X \leftrightarrow Y \bullet dom[X, Y]\ R = \{x : X; y : Y \mid x \mapsto y \in R \bullet x\} \land ran[X, Y]\ R = \{x : X; y : Y \mid x \mapsto y \in R \bullet y\}))$$

$\_ \circ \_$
$\_ \comp \_$

$$\vdash [X, Y, Z](((\_ \comp \_)[X, Y, Z] \in (X \leftrightarrow Y) \times (Y \leftrightarrow Z) \to X \leftrightarrow Z \land (\_ \circ \_)[X, Y, Z] \in (Y \leftrightarrow Z) \times (X \leftrightarrow Y) \to X \leftrightarrow Z) \land (\forall R : X \leftrightarrow Y; S : Y \leftrightarrow Z \bullet (\_ \comp \_)[X, Y, Z]\ (R, S) = (\_ \circ \_)[X, Y, Z]\ (S, R) \land (\_ \circ \_)[X, Y, Z]\ (S, R) = \{x : X; y : Y; z : Z \mid x \mapsto y \in R \land y \mapsto z \in S \bullet x \mapsto z\}))$$

$\_ \rhd \_$
$\_ \lhd \_$

$$\vdash [X, Y](((\_ \lhd \_)[X, Y] \in \mathbb{P}\ X \times (X \leftrightarrow Y) \to X \leftrightarrow Y \land (\_ \rhd \_)[X, Y] \in (X \leftrightarrow Y) \times \mathbb{P}\ Y \to X \leftrightarrow Y) \land (\forall S : \mathbb{P}\ X; R : X \leftrightarrow Y \bullet (\_ \lhd \_)[X, Y]\ (S, R) = \{x : X; y : Y \mid x \in S \land x \mapsto y \in R \bullet x \mapsto y\}) \land (\forall R : X \leftrightarrow Y; T : \mathbb{P}\ Y \bullet (\_ \rhd \_)[X, Y]\ (R, T)$$

$$= \{x : X;\ y : Y \mid x \mapsto y \in R \land y \in T \bullet x \mapsto y\}))$$

$\_ \rhd \_$

$\_ \lhd \_$ $\quad \vdash [X, Y](((\_ \lhd \_)[X, Y] \in \mathbb{P}\ X \times (X \leftrightarrow Y) \to X \leftrightarrow Y$
$\land (\_ \rhd \_)[X, Y] \in (X \leftrightarrow Y) \times \mathbb{P}\ Y \to X \leftrightarrow Y)$
$\land (\forall S : \mathbb{P}\ X;\ R : X \leftrightarrow Y \bullet (\_ \lhd \_)[X, Y]\ (S, R) = \{x : X;\ y : Y \mid x \notin S \land x \mapsto y \in R \bullet x \mapsto y\})$
$\land (\forall R : X \leftrightarrow Y;\ T : \mathbb{P}\ Y \bullet (\_ \rhd \_)[X, Y]\ (R, T) = \{x : X;\ y : Y \mid x \mapsto y \in R \land y \notin T \bullet x \mapsto y\}))$

$\_ ^{\sim}$ $\quad \vdash [X, Y]((\_ ^{\sim})[X, Y] \in (X \leftrightarrow Y) \to Y \leftrightarrow X$
$\land (\forall R : X \leftrightarrow Y \bullet (\_ ^{\sim})[X, Y]\ R = \{x : X;\ y : Y \mid x \mapsto y \in R \bullet y \mapsto x\}))$

$\_ (\!|\ \_\ |\!)$ $\quad \vdash [X, Y]((\_ (\!|\ \_\ |\!))[X, Y] \in (X \leftrightarrow Y) \times \mathbb{P}\ X \to \mathbb{P}\ Y$
$\land (\forall R : X \leftrightarrow Y;\ S : \mathbb{P}\ X \bullet (\_ (\!|\ \_\ |\!))[X, Y]\ (R, S) = \{x : X;\ y : Y \mid x \in S \land x \mapsto y \in R \bullet y\}))$

$\_ ^{+}$

$\_ ^{*}$ $\quad \vdash [X](\{(\_ ^{+})[X], (\_ ^{*})[X]\} \subseteq (X \leftrightarrow X) \to X \leftrightarrow X$
$\land (\forall R : X \leftrightarrow X \bullet (\_ ^{+})[X]\ R = \bigcap \{Q : X \leftrightarrow X \mid R \subseteq Q \land Q \fatsemi Q \subseteq Q\}$
$\land (\_ ^{*})[X]\ R = \bigcap \{Q : X \leftrightarrow X \mid id\ X \subseteq Q \land R \subseteq Q \land Q \fatsemi Q \subseteq Q\}))$

$\_ \oplus \_$ $\quad \vdash [X, Y]((\_ \oplus \_)[X, Y] \in (X \leftrightarrow Y) \times (X \leftrightarrow Y) \to X \leftrightarrow Y$
$\land (\forall f, g : X \leftrightarrow Y \bullet (\_ \oplus \_)[X, Y]\ (f, g) = dom\ g \lhd f \cup g))$

9.1.11.6 Definitions

id $\_$ $\quad \vdash [X](id\ X = \{x : X \bullet x \mapsto x\})$

9.1.11.7 Theorems

z_↔_thm $\vdash \forall X : \mathbb{U}; Y : \mathbb{U} \bullet X \leftrightarrow Y = \mathbb{P} (X \times Y)$

z_↦_thm $\vdash \forall x : \mathbb{U}; y : \mathbb{U} \bullet x \mapsto y = (x, y)$

z_dom_thm $\vdash \forall z : \mathbb{U}; R : \mathbb{U} \bullet z \in dom\ R \Leftrightarrow (\exists y : \mathbb{U} \bullet (z, y) \in R)$

z_ran_thm $\vdash \forall z : \mathbb{U}; R : \mathbb{U} \bullet z \in ran\ R \Leftrightarrow (\exists x : \mathbb{U} \bullet (x, z) \in R)$

z_id_thm $\vdash \forall X : \mathbb{U} \bullet id\ X = \{x : \mathbb{U} \mid x \in X \bullet (x, x)\}$

z_⨾_thm $\vdash \forall R : \mathbb{U}; S : \mathbb{U} \bullet R ⨾ S = S \circ R$

z_∘_thm $\vdash \forall x : \mathbb{U}; S : \mathbb{U}; R : \mathbb{U}$
$\bullet\ x \in S \circ R$
$\Leftrightarrow (\exists y : \mathbb{U} \bullet (x.1, y) \in R \land (y, x.2) \in S)$

z_◁_thm $\vdash \forall x : \mathbb{U}; S : \mathbb{U}; R : \mathbb{U} \bullet x \in S \lhd R \Leftrightarrow x.1 \in S \land x \in R$

z_▷_thm $\vdash \forall x : \mathbb{U}; R : \mathbb{U}; S : \mathbb{U} \bullet x \in R \rhd S \Leftrightarrow x \in R \land x.2 \in S$

z_⩤_thm $\vdash \forall x : \mathbb{U}; S : \mathbb{U}; R : \mathbb{U} \bullet x \in S ⩤ R \Leftrightarrow \neg\ x.1 \in S \land x \in R$

z_⩥_thm $\vdash \forall x : \mathbb{U}; R : \mathbb{U}; S : \mathbb{U} \bullet x \in R ⩥ S \Leftrightarrow x \in R \land \neg\ x.2 \in S$

z_rel_inv_thm
$\vdash \forall x : \mathbb{U}; R : \mathbb{U} \bullet x \in R^{\sim} \Leftrightarrow (x.2, x.1) \in R$

z_rel_image_thm
$\vdash \forall y : \mathbb{U}; R : \mathbb{U}; S : \mathbb{U}$
$\bullet\ y \in R (\!| S |\!) \Leftrightarrow (\exists x : \mathbb{U} \bullet x \in S \land (x, y) \in R)$

z_trans_closure_thm
$\vdash \forall R : \mathbb{U}$
$\bullet\ R^{+}$
$= \bigcap \{Q : \mathbb{U} \mid R \subseteq Q \land Q ⨾ Q \subseteq Q\}$

z_reflex_trans_closure_thm
$\vdash \forall R : \mathbb{U}$
$\bullet\ R^{*}$
$= \bigcap$
$\{Q : \mathbb{U}$
$\mid (id\ \_) \subseteq Q$
$\land R \subseteq Q$
$\land Q ⨾ Q \subseteq Q\}$

z_⊕_thm $\vdash \forall f : \mathbb{U}; g : \mathbb{U} \bullet f \oplus g = dom\ g ⩤ f \cup g$

z_↔_clauses $\vdash \forall X : \mathbb{U} \bullet X \leftrightarrow \{\} = \{\{\}\} \land \{\} \leftrightarrow X = \{\{\}\}$

z_dom_clauses
$\vdash \forall a : \mathbb{U}; b : \mathbb{U}$
$\bullet\ dom\ \mathbb{U} = \mathbb{U}$
$\land dom\ \{\} = \{\}$
$\land dom\ \{a \mapsto b\} = \{a\}$
$\land dom\ \{(a, b)\} = \{a\}$

z_ran_clauses
$\vdash \forall a : \mathbb{U}; b : \mathbb{U}$
$\bullet\ ran\ \mathbb{U} = \mathbb{U}$
$\land ran\ \{\} = \{\}$
$\land ran\ \{a \mapsto b\} = \{b\}$
$\land ran\ \{(a, b)\} = \{b\}$

z_id_clauses $\vdash id\ \{\} = \{\}$

z_⨾_clauses $\vdash \forall R : \mathbb{U} \bullet R ⨾ \{\} = \{\} \land \{\} ⨾ R = \{\} \land \mathbb{U} ⨾ \mathbb{U} = \mathbb{U}$

z_∘_clauses $\vdash \forall R : \mathbb{U} \bullet R \circ \{\} = \{\} \land \{\} \circ R = \{\} \land \mathbb{U} \circ \mathbb{U} = \mathbb{U}$

z_◁_clauses $\vdash \forall R : \mathbb{U}; S : \mathbb{U}$
$\bullet\ \mathbb{U} \lhd R = R \land \{\} \lhd R = \{\} \land S \lhd \{\} = \{\}$

z_▷_clauses $\vdash \forall R : \mathbb{U}; S : \mathbb{U}$

$\bullet\ R \rhd \mathbb{U} = R \land \{\} \rhd S = \{\} \land R \rhd \{\} = \{\}$

z_⩤_clauses $\vdash \forall\ R : \mathbb{U};\ S : \mathbb{U}$
$\bullet\ \mathbb{U} ⩤ R = \{\} \land \{\} ⩤ R = R \land S ⩤ \{\} = \{\}$

z_⩥_clauses $\vdash \forall\ R : \mathbb{U};\ S : \mathbb{U}$
$\bullet\ R ⩥ \mathbb{U} = \{\} \land \{\} ⩥ S = \{\} \land R ⩥ \{\} = R$

z_rel_inv_clauses
$\vdash \mathbb{U}\ ^{\sim} = \mathbb{U} \land \{\}\ ^{\sim} = \{\}$

z_rel_image_clauses
$\vdash \forall\ R : \mathbb{U};\ S : \mathbb{U} \bullet R \ (\!|\ \{\}\ |\!)\ = \{\} \land \{\}\ (\!|\ S\ |\!)\ = \{\}$

z_trans_closure_clauses
$\vdash \mathbb{U}\ ^{+} = \mathbb{U} \land \{\}\ ^{+} = \{\}$

z_reflex_closure_clauses
$\vdash \mathbb{U}\ ^{*} = \mathbb{U} \land \{\}\ ^{*} = (id\ \_)$

z_⊕_clauses $\vdash \forall\ f : \mathbb{U} \bullet f \oplus \{\} = f \land \{\} \oplus f = f \land f \oplus \mathbb{U} = \mathbb{U}$

9.1.12 The Z Theory z_sequences

9.1.12.1 Parents

z_numbers

9.1.12.2 Children

z_sequences1 *z_bags*

9.1.12.3 Global Variables

seq X	$\mathbb{P}\ (\mathbb{Z} \leftrightarrow X)$
seq$_1$ X	$\mathbb{P}\ (\mathbb{Z} \leftrightarrow X)$
iseq X	$\mathbb{P}\ (\mathbb{Z} \leftrightarrow X)$
(_ ⁀ _)[**X**]	$(\mathbb{Z} \leftrightarrow X) \times (\mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
head[**X**]	$(\mathbb{Z} \leftrightarrow X) \leftrightarrow X$
last[**X**]	$(\mathbb{Z} \leftrightarrow X) \leftrightarrow X$
tail[**X**]	$(\mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
front[**X**]	$(\mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
rev[**X**]	$(\mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
squash[**X**]	$(\mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
(_ ↿ _)[**X**]	$\mathbb{P}\ \mathbb{Z} \times (\mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
(_ ↾ _)[**X**]	$(\mathbb{Z} \leftrightarrow X) \times \mathbb{P}\ X \leftrightarrow \mathbb{Z} \leftrightarrow X$
⁀/[**X**]	$(\mathbb{Z} \leftrightarrow \mathbb{Z} \leftrightarrow X) \leftrightarrow \mathbb{Z} \leftrightarrow X$
(**disjoint** _)[**I**, **X**]	
	$\mathbb{P}\ (I \leftrightarrow \mathbb{P}\ X)$
(_ **partition** _)[**I**, **X**]	
	$(I \leftrightarrow \mathbb{P}\ X) \leftrightarrow \mathbb{P}\ X$

9.1.12.4 Fixity

fun 30 leftassoc

(_ ⁀ _)

fun 40 leftassoc

(_ ↾ _)

fun 45 rightassoc

(_ ↿ _)

gen 70 rightassoc

(*iseq* _)(*seq* _) (*seq$_1$* _)

rel (*disjoint* _) (_ *partition* _)

9.1.12.5 Axioms

_ ⁀ _ $\vdash [X]((\_ \frown \_)[X] \in seq\ X \times seq\ X \rightarrow seq\ X$
$\quad \land (\forall\ s,\ t : seq\ X$
$\quad\quad \bullet (\_ \frown \_)[X]\ (s,\ t)$
$\quad\quad\quad = s \cup \{n : dom\ t \bullet n + \#\ s \mapsto t\ n\}))$

head $\vdash [X](head[X] \in seq_1\ X \rightarrow X$
$\quad \land (\forall\ s : seq_1\ X \bullet head[X]\ s = s\ 1))$

last $\vdash [X](last[X] \in seq_1\ X \rightarrow X$
$\quad \land (\forall\ s : seq_1\ X \bullet last[X]\ s = s\ (\#\ s)))$

tail $\vdash [X](tail[X] \in seq_1\ X \rightarrow seq\ X$
$\quad \land (\forall\ s : seq_1\ X$
$\quad\quad \bullet tail[X]\ s = (\lambda\ n : 1\ ..\ \#\ s - 1 \bullet s\ (n + 1))))$

front $\vdash [X](front[X] \in seq_1\ X \rightarrow seq\ X$
$\quad \land (\forall\ s : seq_1\ X$
$\quad\quad \bullet front[X]\ s = (1\ ..\ \#\ s - 1) \lhd s))$

rev $\vdash [X](rev[X] \in seq\ X \rightarrow seq\ X$
$\quad \land (\forall\ s : seq\ X$
$\quad\quad \bullet rev[X]\ s = (\lambda\ n : dom\ s \bullet s\ (\#\ s - n + 1))))$

squash $\vdash [X](squash[X] \in (\mathbb{Z} ⇻ X) \rightarrow seq\ X$
$\quad \land (\forall\ f : \mathbb{Z} ⇻ X$
$\quad\quad \bullet squash[X]\ f$
$\quad\quad\quad = \{p : f$
$\quad\quad\quad\quad \bullet \#\ \{i : dom\ f \mid i \le p.1\} \mapsto p.2\}))$

_ ↿ _ $\vdash [X]((\_ \upharpoonleft \_)[X] \in \mathbb{P}\ \mathbb{Z} \times seq\ X \rightarrow seq\ X$
$\quad \land (\forall\ a : \mathbb{P}\ \mathbb{Z};\ s : seq\ X$
$\quad\quad \bullet (\_ \upharpoonleft \_)[X]\ (a,\ s) = squash\ (a \lhd s)))$

_ ↾ _ $\vdash [X]((\_ \upharpoonright \_)[X] \in seq\ X \times \mathbb{P}\ X \rightarrow seq\ X$
$\quad \land (\forall\ s : seq\ X;\ a : \mathbb{P}\ X$
$\quad\quad \bullet (\_ \upharpoonright \_)[X]\ (s,\ a) = squash\ (s \rhd a)))$

⁀/ $\vdash [X](\frown/[X] \in seq\ seq\ X \rightarrow seq\ X$
$\quad \land \frown/[X]\ \langle\rangle = \langle\rangle$
$\quad \land (\forall\ s : seq\ X \bullet \frown/[X]\ \langle s\rangle = s)$
$\quad \land (\forall\ q,\ r : seq\ seq\ X$
$\quad\quad \bullet \frown/[X]\ (q \frown r) = \frown/[X]\ q \frown \frown/[X]\ r))$

disjoint _ $\vdash [I,$
$\quad X]((disjoint\ \_)[I,\ X] \in \mathbb{P}\ (I \nrightarrow \mathbb{P}\ X)$
$\quad \land (\forall\ S : I \nrightarrow \mathbb{P}\ X$
$\quad\quad \bullet S \in (disjoint\ \_)[I,\ X]$
$\quad\quad\quad \Leftrightarrow (\forall\ i,\ j : dom\ S \mid i \ne j \bullet S\ i \cap S\ j = \varnothing)))$

_ partition _
$\vdash [I,$
$\quad X]((\_\ partition\ \_)[I,\ X] \in (I \nrightarrow \mathbb{P}\ X) \leftrightarrow \mathbb{P}\ X$
$\quad \land (\forall\ S : I \nrightarrow \mathbb{P}\ X;\ T : \mathbb{P}\ X$
$\quad\quad \bullet (S,\ T) \in (\_\ partition\ \_)[I,\ X]$
$\quad\quad\quad \Leftrightarrow disjoint\ S \land \bigcup\ \{i : dom\ S \bullet S\ i\} = T))$

9.1.12.6 Definitions

seq _ $\vdash [X](seq\ X = \{f : \mathbb{N} ⇻ X \mid dom\ f = 1\ ..\ \#\ f\})$

seq₁ _ $\vdash [X](seq_1\ X = \{f : seq\ X \mid \#\ f > 0\})$

iseq $\_$ $\vdash [X](iseq\ X = seq\ X \cap (\mathbb{N} \rightarrowtail\!\!\!\!\!\!\!+\ X))$

9.1.13 The Z Theory z_sequences1

9.1.13.1 Parents

z_sequences *z_numbers1*

9.1.13.2 Children

z_library

9.1.13.3 Theorems

z_seq_thm $\vdash \forall X : \mathbb{U} \bullet seq\ X = \bigcup \{n : \mathbb{N} \bullet 1 .. n \to X\}$

z_prim_seq_induction_thm

$\vdash \ulcorner \forall X\ p$
$\bullet p\ \ulcorner_Z\{\}\urcorner$
$\land (\forall x\ n\ s$
$\bullet x \in X \land n \in \ulcorner_Z\mathbb{N}\urcorner \land s \in \ulcorner_Z 1 .. n \to X\urcorner \land p\ s$
$\Rightarrow p\ \ulcorner_Z s \cup \{(n + 1, x)\}\urcorner)$
$\Rightarrow (\forall s \bullet s \in \ulcorner_Z seq\ X\urcorner \Rightarrow p\ s)\urcorner$

z_seq_thm1 $\vdash \forall X : \mathbb{U}; n : \mathbb{U}$
$\bullet seq\ X = \{s : \mathbb{U} \mid \exists n : \mathbb{N} \bullet s \in 1 .. n \to X\}$

z_size_seq_thm1

$\vdash \forall X : \mathbb{U}; s : \mathbb{U}; n : \mathbb{N}$
$\bullet s \in seq\ X \land \#\ s = n \Leftrightarrow s \in 1 .. n \to X$

z_size_seq_thm2

$\vdash \forall n : \mathbb{N}; s : (seq\ \_) \bullet \#\ s = n \Leftrightarrow dom\ s = 1 .. n$

z_size_seq_ℕ_thm

$\vdash \forall s : (seq\ \_) \bullet \#\ s \in \mathbb{N}$

z_singleton_seq_thm

$\vdash \forall x : \mathbb{U}$
$\bullet \langle x\rangle \in (seq\ \_)$
$\land dom\ \langle x\rangle = \{1\}$
$\land ran\ \langle x\rangle = \{x\}$
$\land \langle x\rangle\ 1 = x$

z_seq_u_thm $\vdash \forall X : \mathbb{U} \bullet \forall s : seq\ X \bullet s \in (seq\ \_)$

z_⁀_thm $\vdash \forall X, Y : \mathbb{U}$
$\bullet \forall s : seq\ X; t : seq\ Y$
$\bullet s \frown t = s \cup \{n : dom\ t \bullet n + \#\ s \mapsto t\ n\}$

z_⁀_∈_seq_thm

$\vdash \forall X, Y : \mathbb{U} \bullet \forall s : seq\ X; t : seq\ Y \bullet s \frown t \in (seq\ \_)$

z_⁀_∈_seq_thm1

$\vdash \forall s : (seq\ \_); t : (seq\ \_) \bullet s \frown t \in (seq\ \_)$

z_⁀_def_thm $\vdash \forall i : \mathbb{U}; t : (seq\ \_)$
$\bullet \{n : dom\ t$
$\bullet n + i \mapsto t\ n\}$
$= \{n : \mathbb{U}; x : \mathbb{U}$
$\mid (n, x) \in t$
$\bullet (n + i, x)\}$

z_⁀_singleton_thm

$\vdash [X](\forall\ s : seq\ X;\ x : X$
$\quad \bullet\ s \frown \langle x\rangle = s \cup \{(\#\ s + 1,\ x)\})$

z_⁀_singleton_thm1
$\vdash \forall\ s : (seq\ \_);\ x : \mathbb{U} \bullet s \frown \langle x\rangle = s \cup \{(\#\ s + 1,\ x)\}$

z_⟨⟩_thm $\vdash \langle\rangle = \{\}$

z_⟨⟩_seq_thm $\vdash \forall\ X : \mathbb{U} \bullet \langle\rangle \in seq\ X$

z_seq_induction_thm
$\vdash \ulcorner\forall\ X\ p$
$\quad \bullet\ p\ \ulcorner_Z\langle\rangle\urcorner$
$\qquad \wedge (\forall\ x\ s$
$\qquad \bullet\ x \in X \wedge s \in \ulcorner_Z seq\ X\urcorner \wedge p\ s \Rightarrow p\ \ulcorner_Z s \frown \langle x\rangle\urcorner)$
$\quad \Rightarrow (\forall\ s \bullet\ s \in \ulcorner_Z seq\ X\urcorner \Rightarrow p\ s)\urcorner$

z_⁀_⟨⟩_thm $\vdash \forall\ X : \mathbb{U} \bullet \forall\ s : seq\ X \bullet s \frown \langle\rangle = s$

z_⟨⟩_⁀_thm $\vdash \forall\ X : \mathbb{U} \bullet \forall\ s : seq\ X \bullet \langle\rangle \frown s = s$

z_dom_seq_thm
$\vdash \forall\ s : (seq\ \_) \bullet dom\ s = 1\ ..\ \#\ s$

z_dom_⁀_thm $\vdash \forall\ s : (seq\ \_);\ t : (seq\ \_)$
$\quad \bullet\ dom\ (s \frown t) = 1\ ..\ \#\ s + \#\ t$

z_seq_seq_x_thm
$\vdash \forall\ X : \mathbb{U};\ s : (seq\ \_) \bullet s \in seq\ X \Leftrightarrow ran\ s \subseteq X$

z_singleton_seq_x_thm
$\vdash \forall\ X : \mathbb{U} \bullet \forall\ x : \mathbb{U} \bullet \langle x\rangle \in seq\ X \Leftrightarrow x \in X$

z_⁀_seq_x_thm
$\vdash \forall\ X : \mathbb{U};\ s1,\ s2 : (seq\ \_)$
$\quad \bullet\ s1 \frown s2 \in seq\ X \Leftrightarrow s1 \in seq\ X \wedge s2 \in seq\ X$

z_size_⁀_thm $\vdash \forall\ s,\ t : (seq\ \_);\ x : \mathbb{U} \bullet \#\ (s \frown t) = \#\ s + \#\ t$

z_size_singleton_seq_thm
$\vdash \forall\ x : \mathbb{U} \bullet \#\ \langle x\rangle = 1$

z_seq_cases_thm
$\vdash \forall\ s : (seq\ \_)$
$\quad \bullet\ s = \langle\rangle \vee (\exists\ s1 : (seq\ \_);\ x : \mathbb{U} \bullet s = s1 \frown \langle x\rangle)$

z_¬_⁀_empty_thm
$\vdash \forall\ s : (seq\ \_);\ x : \mathbb{U} \bullet \neg\ s \frown \langle x\rangle = \langle\rangle$

z_⁀_one_one_thm
$\vdash \forall\ s : (seq\ \_);\ t : (seq\ \_);\ x,\ y : \mathbb{U}$
$\quad \bullet\ s \frown \langle x\rangle = t \frown \langle y\rangle \Leftrightarrow s = t \wedge x = y$

z_⁀_assoc_thm
$\vdash \forall\ s1,\ s2,\ s3 : (seq\ \_)$
$\quad \bullet\ s1 \frown s2 \frown s3 = s1 \frown (s2 \frown s3)$

z_⁀_assoc_thm1
$\vdash \forall\ s1,\ s2,\ s3 : (seq\ \_)$
$\quad \bullet\ s1 \frown (s2 \frown s3) = s1 \frown s2 \frown s3$

z_seq_induction_thm1
$\vdash \ulcorner\forall\ X\ p$
$\quad \bullet\ p\ \ulcorner_Z\langle\rangle\urcorner$
$\qquad \wedge (\forall\ x\ s$
$\qquad \bullet\ x \in X \wedge s \in \ulcorner_Z seq\ X\urcorner \wedge p\ s \Rightarrow p\ \ulcorner_Z\langle x\rangle \frown s\urcorner)$
$\quad \Rightarrow (\forall\ s \bullet\ s \in \ulcorner_Z seq\ X\urcorner \Rightarrow p\ s)\urcorner$

z_num_list_thm

$\vdash \ulcorner\forall\ l\ n$
$\bullet\ {}_{\mathrm{Z}}^{\ulcorner}\ulcorner Z'NumList\ (l,\ n)\urcorner\urcorner$
$=\ {}_{\mathrm{Z}}^{\ulcorner}\{i : \mathbb{U};\ x : \mathbb{U}$
$\mid (i,\ x) \in \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner$
$\bullet\ (i + \ulcorner Z'Int\ n\urcorner,\ x)\}\urcorner\urcorner$

z_seqd_∈_seq_thm

$\vdash \ulcorner\forall\ l\bullet\ {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner \in {}_{\mathrm{Z}}^{\ulcorner}(seq\ \_)\urcorner\urcorner$

z_seqd_⁀_thm $\vdash \ulcorner\forall\ a\ l\bullet\ {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ (Cons\ a\ l)\urcorner\urcorner = {}_{\mathrm{Z}}^{\ulcorner}\langle a\rangle \frown \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner\urcorner$

z_seqd_⁀_rw_thm

$\vdash \ulcorner\forall\ a\ b\ l$
$\bullet\ {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ (Cons\ a\ (Cons\ b\ l))\urcorner\urcorner$
$=\ {}_{\mathrm{Z}}^{\ulcorner}\langle a\rangle \frown (\langle b\rangle \frown \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner)\urcorner\urcorner$

z_∈_seq_app_eq_thm

$\vdash \forall\ s : (seq\ \_);\ m : \mathbb{U};\ x : \mathbb{U} \bullet (m,\ x) \in s \Rightarrow s\ m = x$

z_∈_seqd_app_eq_thm

$\vdash \ulcorner\forall\ l\ m\ x$
$\bullet\ {}_{\mathrm{Z}}^{\ulcorner}(m,\ x)\urcorner \in {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner \Rightarrow {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\ m\urcorner = x\urcorner$

z_size_seqd_thm

$\vdash \#\ \langle\rangle = 0$
$\wedge \ulcorner\forall\ a\ l$
$\bullet\ {}_{\mathrm{Z}}^{\ulcorner}\#\ \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ (Cons\ a\ l)\urcorner\urcorner = {}_{\mathrm{Z}}^{\ulcorner}1 + \#\ \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner\urcorner$

z_size_seqd_length_thm

$\vdash \ulcorner\forall\ l\bullet\ {}_{\mathrm{Z}}^{\ulcorner}\#\ \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner = {}_{\mathrm{Z}}^{\ulcorner}\ulcorner Z'Int\ (Length\ l)\urcorner\urcorner\urcorner$

z_dom_seqd_thm

$\vdash \ulcorner\forall\ l\bullet\ {}_{\mathrm{Z}}^{\ulcorner}dom\ \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner = {}_{\mathrm{Z}}^{\ulcorner}1\ ..\ \#\ \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner\urcorner$

z_ran_seqd_thm

$\vdash \ulcorner\forall\ l\bullet\ {}_{\mathrm{Z}}^{\ulcorner}ran\ \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner = {}_{\mathrm{Z}}^{\ulcorner}\ulcorner Z'Setd\ l\urcorner\urcorner\urcorner$

z_seqd_⁀_⟨⟩_clauses

$\vdash \ulcorner\forall\ l$
$\bullet\ {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner \frown \langle\rangle\urcorner = {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner$
$\wedge\ {}_{\mathrm{Z}}^{\ulcorner}\langle\rangle \frown \ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner = {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l\urcorner\urcorner\urcorner$

z_seqd_eq_thm

$\vdash \ulcorner\forall\ x\ y\ l1\ l2$
$\bullet\ {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ (Cons\ x\ l1)\urcorner\urcorner = {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ (Cons\ y\ l2)\urcorner\urcorner$
$\Leftrightarrow x = y \wedge {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l1\urcorner\urcorner = {}_{\mathrm{Z}}^{\ulcorner}\ulcorner\text{\$"}Z'\langle\rangle\text{"}\ l2\urcorner\urcorner\urcorner$

9.1.14 The Z Theory z_sets

9.1.14.1 Parents

z_language_ps

9.1.14.2 Children

z_relations

9.1.14.3 Global Variables

$\mathbf{X} \leftrightarrow \mathbf{Y}$	$\mathbb{P}\ (X \leftrightarrow Y)$
$\mathbf{X} \rightarrow \mathbf{Y}$	$\mathbb{P}\ (X \leftrightarrow Y)$
$(\_ \notin \_)[\mathbf{X}]$	$X \leftrightarrow \mathbb{P}\ X$
$(\_ \neq \_)[\mathbf{X}]$	$X \leftrightarrow X$
$\varnothing[\mathbf{X}]$	$\mathbb{P}\ X$
$(\_ \subset \_)[\mathbf{X}]$	$\mathbb{P}\ X \leftrightarrow \mathbb{P}\ X$
$\mathbb{P}_\mathbf{1}\ \mathbf{X}$	$\mathbb{P}\ (\mathbb{P}\ X)$
$(\_ \cup \_)[\mathbf{X}]$	$\mathbb{P}\ X \times \mathbb{P}\ X \leftrightarrow \mathbb{P}\ X$
$(\_ \cap \_)[\mathbf{X}]$	$\mathbb{P}\ X \times \mathbb{P}\ X \leftrightarrow \mathbb{P}\ X$
$(\_ \setminus \_)[\mathbf{X}]$	$\mathbb{P}\ X \times \mathbb{P}\ X \leftrightarrow \mathbb{P}\ X$
$(\_ \ominus \_)[\mathbf{X}]$	$\mathbb{P}\ X \times \mathbb{P}\ X \leftrightarrow \mathbb{P}\ X$
$\bigcup[\mathbf{X}]$	$\mathbb{P}\ (\mathbb{P}\ X) \leftrightarrow \mathbb{P}\ X$
$\bigcap[\mathbf{X}]$	$\mathbb{P}\ (\mathbb{P}\ X) \leftrightarrow \mathbb{P}\ X$
second$[\mathbf{X}, \mathbf{Y}]$	$X \times Y \leftrightarrow Y$
first$[\mathbf{X}, \mathbf{Y}]$	$X \times Y \leftrightarrow X$
(**if** _? **then** _! **else** _!)$[\mathbf{X}]$	$\mathbb{B} \times X \times X \leftrightarrow X$
$(\_ \overset{\oplus}{\oplus} \_)[\mathbf{X}]$	$X \times \mathbb{P}\ X \leftrightarrow X$
$(\mathbf{\Pi}\ \_?)$	$\mathbb{B} \leftrightarrow \mathbb{B}$
$(\ll \_! \gg)[\mathbf{X}]$	$X \leftrightarrow X$
$(\_\ \underline{\ \_\ }\ \_)[\mathbf{X}, \mathbf{Y}]$	$\mathbb{P}\ (X \times (X \leftrightarrow Y) \times Y)$

9.1.14.4 Fixity

fun 0 rightassoc

(*if* _? *then* _! *else* _!) $(\_ \overset{\oplus}{\oplus} \_)$
$(\ll \_! \gg)$ $(\Pi\ \_?)$

fun 25 leftassoc

$(\_ \ominus \_)$

fun 30 leftassoc

$(\_ \setminus \_)$ $(\_ \cup \_)$

fun 40 leftassoc
$(\_ \cap \_)$

gen 5 rightassoc
$(\_ \leftrightarrow \_)$ $(\_ \rightarrow \_)$

gen 70 rightassoc
$(\mathbb{P}_1\ \_)$

rel $(\_ \notin \_)\,(\_ \subset \_)(\_ \neq \_)(\_ \ \underline{\_}\ \_)$

9.1.14.5 Axioms

$\_ \notin \_$
$\_ \neq \_$ $\vdash [X](((\_ \neq \_)[X] \in X \leftrightarrow X$
$\quad \wedge (\_ \notin \_)[X] \in X \leftrightarrow \mathbb{P}\ X)$
$\quad \wedge (\forall\ x,\ y : X \bullet (x,\ y) \in (\_ \neq \_)[X] \Leftrightarrow \neg\ x = y)$
$\quad \wedge (\forall\ x : X;\ S : \mathbb{P}\ X$
$\quad\quad \bullet (x,\ S) \in (\_ \notin \_)[X] \Leftrightarrow \neg\ x \in S))$

$\_ \subset \_$ $\vdash [X]((\_ \subset \_)[X] \in \mathbb{P}\ X \leftrightarrow \mathbb{P}\ X$
$\quad \wedge (\forall\ S,\ T : \mathbb{P}\ X$
$\quad\quad \bullet (S,\ T) \in (\_ \subset \_)[X] \Leftrightarrow S \subseteq T \wedge S \neq T))$

$\_ \cup \_$
$\_ \cap \_$
$\_ \setminus \_$
$\_ \ominus \_$ $\vdash [X](\{(\_ \cup \_)[X],\ (\_ \cap \_)[X],\ (\_ \setminus \_)[X],\ (\_ \ominus \_)[X]\}$
$\quad\quad \subseteq \mathbb{P}\ X \times \mathbb{P}\ X \rightarrow \mathbb{P}\ X$
$\quad \wedge (\forall\ S,\ T : \mathbb{P}\ X$
$\quad\quad \bullet (\_ \cup \_)[X]\ (S,\ T) = \{x : X \mid x \in S \vee x \in T\}$
$\quad\quad\quad \wedge (\_ \cap \_)[X]\ (S,\ T) = \{x : X \mid x \in S \wedge x \in T\}$
$\quad\quad\quad \wedge (\_ \setminus \_)[X]\ (S,\ T) = \{x : X \mid x \in S \wedge x \notin T\}$
$\quad\quad\quad \wedge (\_ \ominus \_)[X]\ (S,\ T)$
$\quad\quad\quad\quad = \{x : X$
$\quad\quad\quad\quad\quad \mid \neg$
$\quad\quad\quad\quad (x \in S \Leftrightarrow x \in T)\}))$

$\bigcup$
$\bigcap$ $\vdash [X](\{\bigcup[X],\ \bigcap[X]\} \subseteq \mathbb{P}\ (\mathbb{P}\ X) \rightarrow \mathbb{P}\ X$
$\quad \wedge (\forall\ A : \mathbb{P}\ \mathbb{P}\ X$
$\quad\quad \bullet \bigcup[X]\ A = \{x : X \mid \exists\ S : A \bullet x \in S\}$
$\quad\quad\quad \wedge \bigcap[X]\ A = \{x : X \mid \forall\ S : A \bullet x \in S\}))$

second
first $\vdash [X,$
$\quad Y]((first[X,\ Y] \in X \times Y \rightarrow X$
$\quad \wedge second[X,\ Y] \in X \times Y \rightarrow Y)$
$\quad \wedge (\forall\ x : X;\ y : Y$
$\quad\quad \bullet first[X,\ Y]\ (x,\ y) = x$
$\quad\quad\quad \wedge second[X,\ Y]\ (x,\ y) = y))$

if _? then _! else _!
$\vdash [X]((if\ \_?\ then\ \_!\ else\ \_!)[X] \in \mathbb{B} \times X \times X \rightarrow X$
$\quad \wedge (\forall\ x,\ y : X$

$\bullet$ $(if\ \_?\ then\ \_!\ else\ \_!)[X]\ (true,\ x,\ y) = x$
$\land\ (if\ \_?\ then\ \_!\ else\ \_!)[X]\ (false,\ x,\ y)$
$= y))$

$\_ \overset{\oplus}{\oplus} \_$	$\vdash [X]((\_ \overset{\oplus}{\oplus} \_)[X] \in X \times \mathbb{P}\ X \to X \land (\_ \overset{\oplus}{\oplus} \_)[X] = first)$
Π _?	$\vdash (\Pi\ \_?) \in \mathbb{B} \to \mathbb{B} \land (\forall\ x : \mathbb{B} \bullet \Pi\ x \Leftrightarrow x)$
$\ll \_! \gg$	$\vdash [X]((\ll \_! \gg)[X] \in X \to X$ $\land (\forall\ x : X \bullet (\ll \_! \gg)[X]\ x = x))$
$\_ \underline{\ \_\ } \_$	$\vdash [X,$ $Y]((\_ \underline{\ \_\ } \_)[X,\ Y] \in \mathbb{P}\ (X \times \mathbb{P}\ (X \times Y) \times Y)$ $\land (\forall\ x : X;\ R : \mathbb{P}\ (X \times Y);\ y : Y$ $\bullet (x,\ R,\ y) \in (\_ \underline{\ \_\ } \_)[X,\ Y] \Leftrightarrow (x,\ y) \in R))$

9.1.14.6 Definitions

$\_ \leftrightarrow \_$	$\vdash [X,\ Y](X \leftrightarrow Y = \mathbb{P}\ (X \times Y))$
$\_ \to \_$	$\vdash [X,$ $Y](X \to Y$ $= \{f : X \leftrightarrow Y$ $\mid \forall\ x : X \bullet \exists_1\ y : Y \bullet (x,\ y) \in f\})$
$\varnothing$	$\vdash [X](\varnothing[X] = \{x : X \mid false\})$
$\mathbb{P}_1\ \_$	$\vdash [X](\mathbb{P}_1\ X = \{S : \mathbb{P}\ X \mid S \neq \varnothing\})$

9.1.14.7 Theorems

z_≠_thm	$\vdash \forall\ x : \mathbb{U};\ y : \mathbb{U} \bullet x \neq y \Leftrightarrow \neg\ x = y$
z_∉_thm	$\vdash \forall\ x : \mathbb{U};\ S : \mathbb{U} \bullet x \notin S \Leftrightarrow \neg\ x \in S$
z_∅_thm	$\vdash \forall\ x1 : \mathbb{U} \bullet \neg\ x1 \in \varnothing$
z_∅_thm1	$\vdash \varnothing = \{\}$
z_ℙ₁_thm	$\vdash \forall\ X : \mathbb{U} \bullet \mathbb{P}_1\ X = \{S : \mathbb{P}\ X \mid S \neq \varnothing\}$
z_∪_thm	$\vdash \forall\ z : \mathbb{U};\ s : \mathbb{U};\ t : \mathbb{U} \bullet z \in s \cup t \Leftrightarrow z \in s \lor z \in t$
z_∩_thm	$\vdash \forall\ z : \mathbb{U};\ s : \mathbb{U};\ t : \mathbb{U} \bullet z \in s \cap t \Leftrightarrow z \in s \land z \in t$
z_set_dif_thm	$\vdash \forall\ z : \mathbb{U};\ s : \mathbb{U};\ t : \mathbb{U} \bullet z \in s \setminus t \Leftrightarrow z \in s \land z \notin t$
z_⊖_thm	$\vdash \forall\ z : \mathbb{U};\ s : \mathbb{U};\ t : \mathbb{U} \bullet z \in s \ominus t \Leftrightarrow \neg\ (z \in s \Leftrightarrow z \in t)$
z_⊆_thm1	$\vdash \ulcorner\forall\ X$ $\bullet\ \ulcorner_Z(\_ \subseteq \_)[X]\urcorner \in \ulcorner_Z\mathbb{P}\ X \leftrightarrow \mathbb{P}\ X\urcorner$ $\land\ \ulcorner_Z\forall\ S,\ T : \mathbb{P}\ X$ $\bullet (S,\ T) \in (\_ \subseteq \_)[X]$ $\Leftrightarrow (\forall\ x : X \bullet x \in S \Rightarrow x \in T)\urcorner\urcorner$
z_⊆_thm	$\vdash \forall\ s : \mathbb{U};\ t : \mathbb{U} \bullet s \subseteq t \Leftrightarrow (\forall\ x : \mathbb{U} \bullet x \in s \Rightarrow x \in t)$
z_∈_ℙ_thm	$\vdash \forall\ s : \mathbb{U};\ t : \mathbb{U} \bullet s \in \mathbb{P}\ t \Leftrightarrow s \subseteq t$
z_⊂_thm	$\vdash \forall\ s : \mathbb{U};\ t : \mathbb{U} \bullet s \subset t \Leftrightarrow s \subseteq t \land s \neq t$
z_⋃_thm	$\vdash \forall\ z : \mathbb{U};\ a : \mathbb{U} \bullet z \in \bigcup\ a \Leftrightarrow (\exists\ S : \mathbb{U} \bullet S \in a \land z \in S)$
z_⋂_thm	$\vdash \forall\ z : \mathbb{U};\ a : \mathbb{U} \bullet z \in \bigcap\ a \Leftrightarrow (\forall\ S : \mathbb{U} \bullet S \in a \Rightarrow z \in S)$
z_first_thm	$\vdash \forall\ x : \mathbb{U} \bullet first\ x = x.1$
z_second_thm	$\vdash \forall\ x : \mathbb{U} \bullet second\ x = x.2$
z_if_thm	$\vdash \forall\ x,\ y : \mathbb{U}$ $\bullet\ if\ true\ then\ x\ else\ y = x$ $\land\ if\ false\ then\ x\ else\ y = y$
z_guillemets_thm	$\vdash \forall\ x : \mathbb{U} \bullet \ll x \gg = x$

z_underlining_brackets_thm
$\vdash \forall\ x : \mathbb{U};\ R : \mathbb{U};\ y : \mathbb{U} \bullet x\ \underline{\ R\ }\ y \Leftrightarrow (x,\ y) \in R$

z_∪_clauses $\vdash \forall\ a : \mathbb{U}$
$\bullet\ a \cup \{\} = a$
$\wedge \{\} \cup a = a$
$\wedge\ a \cup \mathbb{U} = \mathbb{U}$
$\wedge\ \mathbb{U} \cup a = \mathbb{U}$
$\wedge\ a \cup a = a$

z_∩_clauses $\vdash \forall\ a : \mathbb{U}$
$\bullet\ a \cap \{\} = \{\}$
$\wedge \{\} \cap a = \{\}$
$\wedge\ a \cap \mathbb{U} = a$
$\wedge\ \mathbb{U} \cap a = a$
$\wedge\ a \cap a = a$

z_set_dif_clauses
$\vdash \forall\ a : \mathbb{U}$
$\bullet\ a \setminus \{\} = a$
$\wedge \{\} \setminus a = \{\}$
$\wedge\ a \setminus \mathbb{U} = \{\}$
$\wedge\ a \setminus a = \{\}$

z_⊖_clauses $\vdash \forall\ a : \mathbb{U}$
$\bullet\ a \ominus \{\} = a$
$\wedge \{\} \ominus a = a$
$\wedge\ a \ominus \mathbb{U} = \mathbb{U} \setminus a$
$\wedge\ \mathbb{U} \ominus a = \mathbb{U} \setminus a$
$\wedge\ a \ominus a = \{\}$

z_⊆_clauses $\vdash \forall\ a : \mathbb{U} \bullet a \subseteq a \wedge \{\} \subseteq a \wedge a \subseteq \mathbb{U}$

z_⊂_clauses $\vdash \forall\ a : \mathbb{U} \bullet \neg\ a \subset a \wedge \neg\ a \subset \{\} \wedge \{\} \subset \mathbb{U}$

z_⋂_clauses $\vdash \bigcap \{\} = \mathbb{U} \wedge \bigcap \mathbb{U} = \{\}$

z_⋃_clauses $\vdash \bigcup \{\} = \{\} \wedge \bigcup \mathbb{U} = \mathbb{U}$

z_ℙ_clauses $\vdash \forall\ a : \mathbb{U} \bullet \mathbb{P}\ \{\} = \{\{\}\} \wedge \mathbb{P}\ \mathbb{U} = \mathbb{U} \wedge a \in \mathbb{P}\ a \wedge \{\} \in \mathbb{P}\ a$

z_$\mathbb{P}_1$_clauses
$\vdash \forall\ a : \mathbb{U}$
$\bullet\ \mathbb{P}_1\ \{\} = \{\} \wedge (a \in \mathbb{P}_1\ a \Leftrightarrow a \neq \{\}) \wedge \neg \{\} \in \mathbb{P}_1\ a$

z_×_clauses $\vdash \forall\ a : \mathbb{U} \bullet a \times \{\} = \{\} \wedge \{\} \times a = \{\} \wedge \mathbb{U} \times \mathbb{U} = \mathbb{U}$

z_sets_ext_clauses
$\vdash \forall\ s,\ t : \mathbb{U}$
$\bullet\ (s = t \Leftrightarrow (\forall\ x : \mathbb{U} \bullet x \in s \Leftrightarrow x \in t))$
$\wedge\ (s \subseteq t \Leftrightarrow (\forall\ x : \mathbb{U} \bullet x \in s \Rightarrow x \in t))$
$\wedge\ (s \subset t$
$\Leftrightarrow (\forall\ x : \mathbb{U} \bullet x \in s \Rightarrow x \in t)$
$\wedge\ (\exists\ y : \mathbb{U} \bullet y \in t \wedge \neg\ y \in s))$

9.2 Theory Related ML Values

This section contains various theory related ML values (e.g. the value of theorems bound to ML names, or special tactics of proof contexts associated with the theory). Where a theorem or definition is bound to an ML name the value of the theorem is to be found in the theory listing, only the ML name is given below.

9.2.1 Z Sets

SML

```
signature ZSets = sig
```

Description This provides the Z library sets material. It creates the theory *z_sets*.

SML

```
(* Proof Context: 'z_∈_set_lib *)
```

Description A component proof context for handling the membership of expressions created by Z set operations of the Z library.

Predicates and expressions treated by this proof context are constructs formed from:

```
⋂, ⋃, ∩, ∪, \, ⊖, ℙ₁, ∅
```

Contents

Rewriting:

Stripping theorems:

Stripping conclusions:

All three of the above have theorems concerning the membership (∈) of terms generated by the following operators:

```
⋂, ⋃, ∩, ∪, \, ⊖, ℙ₁, ∅
```

Stripping also contains the above in negated forms.

Rewriting canonicalisation:

𝕌 simplification has the definition of ↔ added.

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_sets*. It is intended to be used with proof context "'z_set_lang" and "'z_normal" It is not intended to be mixed with HOL proof contexts.

See Also *'z_sets_ext_lib*

SML

(∗ *Proof Context*: **′z_normal** ∗)

Description A component proof context for normalising certain constructs of the Z library. The normalisation is done to fix on, in each case, one of two possible equivalent representations of the same concept. These constructs are:

$x \neq y$	*normalised to* $\neg(x = y)$
$x \notin y$	*normalised to* $\neg(x \in y)$
$\varnothing$	*normalised to* $\{\}$
$x \in \mathbb{P}\ y$	*normalised to* $x \subseteq y$
if true then x else y	*normalised to x*
if false then x else y	*normalised to y*

Contents

Rewriting:

z_∈_ℙ_thm, z_∅_thm1, z_∉_thm, z_≠_thm, z_if_thm

Stripping theorems:

z_∈_ℙ_thm, z_∅_thm, z_∉_thm, z_≠_thm, z_if_thm
and these all pushed through ¬

Stripping conclusions:

z_∈_ℙ_thm, z_∅_thm, z_∉_thm, z_≠_thm, z_if_thm
and these all pushed through ¬

Rewriting canonicalisation:

𝕌 simplification has the definition of ↔ added.

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_sets*. It is intended to be used with proof contexts "′z_set_lib" or "′z_set_alg".

SML

(∗ *Proof Context*: ′**z_sets_alg** ∗)

Description A component proof context for handling algebraic reasoning of expressions created by Z set operations of the Z library.

Predicates and expressions treated by this proof context are constructs formed from:

$\in$, $\cap$, $\cup$, $\setminus$, $\ominus$, $\subseteq$, $\subset$, $\bigcap$, $\bigcup$, $\mathbb{P}$, $\mathbb{P}_1$, $\{D \mid false \bullet V\}$, $\times$

Contents

Rewriting:

z_∪_clauses, z_∩_clauses, z_set_dif_clauses, z_⊖_clauses,
z_⊆_clauses, z_⊂_clauses, z_⋃_clauses, z_⋂_clauses,
z_ℙ_clauses, z_ℙ₁_clauses, z_seta_false_conv,
z_×_clauses

Stripping theorems:

z_∪_clauses, z_∩_clauses, z_set_dif_clauses, z_⊖_clauses,
z_⊆_clauses, z_⊂_clauses, z_⋃_clauses, z_⋂_clauses,
z_ℙ_clauses, z_ℙ₁_clauses, z_seta_false_conv,
z_×_clauses
as necessary converted to membership statements by ∈_C,
And all of this pushed through ¬

Stripping conclusions:

z_∪_clauses, z_∩_clauses, z_set_dif_clauses, z_⊖_clauses,
z_⊆_clauses, z_⊂_clauses, z_⋃_clauses, z_⋂_clauses,
z_ℙ_clauses, z_ℙ₁_clauses, z_seta_false_conv,
z_×_clauses
as necessary converted to membership statements by ∈_C,
And all of this pushed through ¬

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_sets*. It is intended to usable with proof context "′z_∈_set_lib", and always with "′z_normal". The proof context ensures that its simplifications will be attempted before more general rules concerned membership of set operators are used.

It is not intended to be mixed with HOL proof contexts.

SML
(∗ *Proof Context*: **′z_sets_ext_lib** ∗)

Description An aggressive component proof context for handling the manipulation of Z set expressions, by breaking them into predicate calculus, within the Z library.

Predicates treated by this proof context are constructs formed from:

⊆, ⊂

Contents

Rewriting:

z_⊆_conv, *z_⊂_thm*, *z_setd_⊆_conv*

Stripping theorems:

z_⊆_conv, *z_⊂_thm*, *z_setd_⊆_conv*,
plus these all pushed in through ¬

Stripping conclusions:

z_⊆_conv, *z_⊂_thm*, *z_setd_⊆_conv*,
plus these all pushed in through ¬

In all of the above *z_setd_⊆_conv*, which does the conversion:

{*x1*,*x2*,...} ⊆ *y* −−−> *x1* ∈ *y* ∧ *x2* ∈ *y* ∧ ...

is used, where applicable, in preference to *z_⊆_conv*, which, in the simplest cases, does the conversion:

p ⊆ *q* −−−> ∀ *x1* • *x1* ∈ *p* ⇒ *xx1* ∈ *q*

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_sets*. It is intended to always be used in conjunction with "′z_set_lib" and "′z_set_ext_lang". If used with "′z_sets_alg" then the simplification in that proof context will take precedence over the extensionality effects of this proof context.

It is not intended to be mixed with HOL proof contexts.

See Also *′z_∈_set_lib*

SML
val **mk_z_if** : (*TERM* ∗ *TERM* ∗ *TERM*) −> *TERM*;
val **dest_z_if** : *TERM* −> (*TERM* ∗ *TERM* ∗ *TERM*);
val **is_z_if** : *TERM* −> *bool*;

Description Constructor, destructor and discriminator functions for Z conditional terms.

Errors
78003 ?0 is not a Z conditional term
78004 ?0 and ?1 do not have the same types
78005 ?0 is not of type ⌜:*BOOL*⌝

SML
```
val mk_z_⊆ : (TERM * TERM) -> TERM;
val dest_z_⊆ : TERM -> (TERM * TERM);
val is_z_⊆ : TERM -> bool;
```

Description Constructor, destructor and discriminator functions for Z subset terms.

Errors
```
78006 ?0 is not of the form ⌜Z a ⊆ s⌝
78004 ?0 and ?1 do not have the same types
78007 ?0 does not have a Z set type
```

SML
```
val z_seta_false_conv : CONV;
```

Description Simplifies a Z set abstraction whose predicate is false.

Conversion

$$\frac{}{\vdash \{\ D \mid false \bullet P\} = \{\}} \quad \begin{array}{l} z\_seta\_false\_conv \\ ⌜_Z\{\ D \mid false \bullet P\}⌝ \end{array}$$

Errors
```
78002 ?0 is not of the form: ⌜Z{D | false • P}⌝
```

SML
```
val z_⊆_conv : CONV;
```

Description Use *z_⊆_thm* in combination with knowledge about tuples. Given as input an equality of the form $v \subseteq w$ then:

If w is of type *ty SET* where *ty* is not a tuple type:

Conversion

$$\frac{}{\vdash (v \subseteq w) \Leftrightarrow (\forall\ xn : \mathbb{U} \bullet xn \in v \Rightarrow xn \in w)} \quad \begin{array}{l} z\_\subseteq\_conv \\ ⌜_Z v \subseteq w⌝ \end{array}$$

where *xn* is the first variable in the list *x1*, *x2*,... that doesn't appear in v or w (free or bound).

If w is of type *ty SET* where *ty* is an n-tuple type, or binding type, then sufficient *x_i* will be introduced, instead of just *xn*, to allow *xn* to be replaced by a construct of bindings and tuples of the *x_i*, such that no *x_i* has a binding or tuple type and appears exactly once in the construct.

Example
```
z_⊆_conv ⌜Z p ⊆ r × [a,b:X] × x2⌝ =
⊢ p ⊆ r × [a, b : X] × x2
    ⇔ (∀ x1 : 𝕌; x3 : 𝕌; x4 : 𝕌; x5 : 𝕌
      • (x1, (a ≙ x3, b ≙ x4), x5) ∈ p
        ⇒ (x1, (a ≙ x3, b ≙ x4), x5) ∈ r × [a, b : X] × x2)
```

Notice how the introduced universal quantification "skips" *x2* which is present in the input term.

See Also *z_⊆_thm*, *z_∈_ℙ_conv*.

Errors
```
78001 ?0 is not of the form ⌜Z v ⊆ w⌝
```

SML
```
val z_⊕⊕_def : THM;
val z'Π_def : THM;
val z'if_def : THM;
val z'guillemets_def : THM;
val z'underlining_brackets_def : THM;
```

Description These are the ML bindings of the definitions of built-in global variables that support the use of the ProofPower-Z language.

SML
```
val z_≠_def : THM;
val z_∉_def : THM;
val z_∅_def : THM;
val z_⊂_def : THM;
val z_ℙ1_def : THM;
val z_∪_def : THM;
val z_∩_def : THM;
val z_setdif_def : THM;
val z_⊖_def : THM;
val z_⋃_def : THM;
val z_⋂_def : THM;
val z_first_def : THM;
val z_second_def : THM;
val z_↔_def : THM;
val z_→_def : THM;
```

Description These are the ML bindings of the definitions of the theory *z_sets*.

SML
```
val z_≠_thm: THM;
val z_∉_thm: THM;
val z_∅_thm: THM;
val z_⊆_thm: THM;
val z_⊆_thm1: THM;
val z_⊂_thm: THM;
val z_∈_ℙ_thm: THM;
val z_ℙ₁_thm: THM;
val z_∪_thm: THM;
val z_∩_thm: THM;
val z_set_dif_thm: THM;
val z_⊖_thm: THM;
val z_⋃_thm: THM;
val z_⋂_thm: THM;
val z_first_thm: THM;
val z_second_thm: THM;
val z_∪_clauses: THM;
val z_∩_clauses: THM;
val z_set_dif_clauses: THM;
val z_⊖_clauses: THM;
val z_⊆_clauses: THM;
val z_⊂_clauses: THM;
val z_⋃_clauses: THM;
val z_⋂_clauses: THM;
val z_ℙ_clauses: THM;
val z_ℙ₁_clauses: THM;
val z_×_clauses: THM;
val z_if_thm: THM;
val z_guillemets_thm: THM;
val z_underlining_brackets_thm: THM;
val z_sets_ext_clauses: THM;
```

Description These are the ML bindings of the theorems of the theory *z_sets*.

9.2.2 Z Relations

SML
```
signature ZRelations = sig
```

Description This provides the basic proof support for the Z library relations. It creates the theory *z_relations*.

SML

```
(* Proof Context: 'z_∈_rel *)
```

Description A component proof context for handling the membership of Z relations created by Z library operations.

Predicates treated by this proof context are constructs formed from:

$\mapsto$, $\oplus$, $\_^{+}$, $\_^{*}$, $\_^{\sim}$, $\_ (\!| \_ |\!)$, $⩥$, $⩤$, $\rhd$, $\lhd$,
o, $⨾$, id, ran, dom, $\leftrightarrow$

Contents

Rewriting:

$z\_\mapsto\_thm$

Stripping theorems:

Stripping conclusions:

All three of the above also have theorems concerning the membership of terms generated by the following operators:

$\oplus$, $\_^{+}$, $\_^{*}$, $\_^{\sim}$, $\_ (\!| \_ |\!)$, $⩥$, $⩤$, $\rhd$, $\lhd$,
o, $⨾$, id, ran, dom, $\leftrightarrow$

Stripping also contains the above in negated forms.

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_relations*. It is intended to be used with proof contexts "z_sets_ext" and "z_sets_alg". It is not intended to be mixed with HOL proof contexts.

SML

(∗ *Proof Context*: ′**z_rel_alg** ∗)

Description A component proof context for the simplification of Z relations created by Z library operations.

Predicates treated by this proof context are constructs formed from:

$\oplus$, $\_^{+}$, $\_^{*}$, $\_^{\sim}$, $\_ (\!| \_ |\!)$, $\rhd\!\!\!>$, $<\!\!\!\lhd$, $\rhd$, $\lhd$,
o, $\fatsemi$, id, ran, dom, $\leftrightarrow$

Contents

Rewriting:

z_↔_clauses, *z_dom_clauses*, *z_ran_clauses*,*z_id_clauses*,
z_⨾_clauses, *z_∘_clauses*, *z_◁_clauses*, *z_▷_clauses*,
z_⩤_clauses, *z_⩥_clauses*, *z_rel_inv_clauses*, *z_rel_image_clauses*,
z_trans_closure_clauses, *z_reflex_closure_clauses*,
z_⊕_clauses

Stripping theorems:

z_↔_clauses, *z_dom_clauses*, *z_ran_clauses*,*z_id_clauses*,
z_⨾_clauses, *z_∘_clauses*, *z_◁_clauses*, *z_▷_clauses*,
z_⩤_clauses, *z_⩥_clauses*, *z_rel_inv_clauses*, *z_rel_image_clauses*,
z_trans_closure_clauses, *z_reflex_closure_clauses*,
z_⊕_clauses
Expressed as memberships, as necessary, using ∈_*C*
All also pushed through ¬

Stripping conclusions:

z_↔_clauses, *z_dom_clauses*, *z_ran_clauses*,*z_id_clauses*,
z_⨾_clauses, *z_∘_clauses*, *z_◁_clauses*, *z_▷_clauses*,
z_⩤_clauses, *z_⩥_clauses*, *z_rel_inv_clauses*, *z_rel_image_clauses*,
z_trans_closure_clauses, *z_reflex_closure_clauses*,
z_⊕_clauses
Expressed as memberships, as necessary, using ∈_*C*
All also pushed through ¬

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_relations*. It is intended to be used with proof contexts "z_sets_ext" and "z_sets_alg". There are clashes of effects if merged with "z_∈_rel", resolved in favour of "z_∈_rel", though the resulting merge has its uses. It is not intended to be mixed with HOL proof contexts.

SML

(∗ *Proof Context*: **′z_tuples** ∗)

Description A component proof context for handling the manipulation of Z tuples and cartesian products within the Z language and library.

Expressions and predicates treated by this proof context are constructs formed from:

(membership of) ×, *equations of tuple displays*,
selection from tuple displays, *first*, *second*, ↦

Contents

Rewriting:

z_∈_×_conv,
z_tuple_eq_conv, $z\_sel_t\_conv$,
z_second_thm, *z_first_thm*

Stripping theorems:

z_∈_×_conv,
z_tuple_eq_conv, ∈_C $z\_sel_t\_conv$,
$z\_sel_t\_conv$ *(where component of tuple is boolean)*,
plus these all pushed in through ¬

Stripping conclusions:

z_∈_×_conv,
z_tuple_eq_conv, ∈_C $z\_sel_t\_conv$,
$z\_sel_t\_conv$ *(where component of tuple is boolean)*,
plus these all pushed in through ¬

Stripping also contains the above in negated forms.

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *_basic_prove_conv*, and no existence prover (1-tuples and 2-tuples are handled in proof context "z_predicates").

Usage Notes It requires theory *z_relations*. It is intended to be used with proof contexts "z_sets_ext" and "z_sets_alg". It should not be used with "′z_tuples_lang". It is not intended to be mixed with HOL proof contexts.

SML

(∗ *Proof Context*: **′z_elementwise_eq** ∗)

Description A aggressive component proof context for forcing the elementwise comparison of any two items of tuple or binding types.

Predicates and expressions treated by this proof context are:

$x = y$ *where x has a tuple type*
$x = y$ *where x has the type of a bidning display*

Contents

Rewriting:

z_binding_eq_conv3, *z_tuple_eq_conv1*

Stripping theorems:

z_binding_eq_conv3, *z_tuple_eq_conv1*
plus these all pushed in through ¬

Stripping conclusions:

z_binding_eq_conv3, *z_tuple_eq_conv1*,
plus these all pushed in through ¬

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_relations*. It is intended to be used with proof context "z_language". It is not intended to be mixed with HOL proof contexts.

SML

(∗ *Proof Context*: **z_language** ∗)

Description A mild complete proof context for reasoning in the Z language. It will also do some minor peices of Z Library reasoning - in particular, it "understands" maplets and ⊆.

It consists of the merge of the proof contexts:

"*z_predicates*",
"′*z_∈_set_lang*",
"′*z_bindings*",
"′*z_schemas*",
"′*z_tuples*"

Usage Notes It requires theory *z_relations* (rather than *z_language_ps* as one might expect). This is because we wish to provide a proof context that can be added to to provide Library reasoning facilities. This means that we cannot use the Z language proof context "′z_*tuples_lang*″, *asthisisincompatiblewith*"′*z*_tuples", its library extension. This is why this proof context understands maplets, which are Z Library contructs.

SML
```
(∗ Proof Context: z_language_ext ∗)
```

Description An aggressive complete proof context for reasoning in the Z language. It uses the extensionality of sets, and will also decompose any equality of objects of schema or tuple type into a pairwise equality clause. It will also do some minor peices of Z Library reasoning - in particular, it "understands" maplets and ⊆.

It consists of the merge of the proof contexts:

```
"z_predicates",
"'z_∈_set_lang",
"'z_sets_ext_lang",
"'z_bindings",
"'z_schemas",
"'z_tuples",
"'z_elementwise_eq"
```

Usage Notes It requires theory *z_relations* (rather than *z_language_ps* as one might expect). This is because we wish to provide a proof context that can be added to to provide Library reasoning facilities. This means that we cannot use the Z language proof context "'z_*tuples_lang''*, *asthisisincompatiblewith*"'*z*_tuples", its library extension. This is why this proof context understands maplets, which are Z Library contructs.

SML
```
(∗ Proof Context: z_sets_ext ∗)
```

Description An aggressive complete proof context for handling the manipulation of Z set expressions, by breaking them into predicate calculus.

It consists of the merge of the proof contexts:

```
"z_language_ext",
"'z_∈_set_lib",
"'z_sets_ext_lib",
"'z_normal"
```

Usage Notes It requires theory *z_relations*.

It is not intended to be mixed with HOL proof contexts or "z_sets_alg", which offers an alternative approach to reasoning about sets.

SML
```
(∗ Proof Context: z_sets_alg ∗)
```

Description A mild complete proof context for handling the manipulation of Z set expressions, by algebraic reasoning and knowledge of the set membership of the set operators.

It consists of the merge of the proof contexts:

```
"z_language",
"'z_∈_set_lib",
"'z_sets_alg",
"'z_normal"
```

Usage Notes It requires theory *z_relations*. The proof context ensures that its simplifications will be attempted before more general rules concerned membership of set operators are used (including extensionality rules).

It is not intended to be mixed with HOL proof contexts.

SML

```
(* Proof Context: z_rel_ext *)
```

Description An aggressive complete proof context for reasoning about Z relations. When stripping or rewriting it attempts to reduce any predicate concerning relational constructs to predicate calculus. As a side effect set constructs are also reduced to predicate calculus. The proof context is a merge of:

```
z_sets_ext − extensional reasoning about sets
'z_∈_rel − membership of relational constructs
'z_rel_alg − simplifications of relational constructs
```

It requires the theory "z_relations".

SML

```
val z_↔_thm: THM;
val z_↦_thm: THM;
val z_dom_thm: THM;
val z_ran_thm: THM;
val z_id_thm: THM;
val z_⨾_thm: THM;
val z_∘_thm: THM;
val z_◁_thm: THM;
val z_▷_thm: THM;
val z_⩤_thm: THM;
val z_⩥_thm: THM;
val z_rel_inv_thm: THM;
val z_rel_image_thm: THM;
val z_trans_closure_thm: THM;
val z_reflex_trans_closure_thm: THM;
val z_⊕_thm: THM;
val z_↔_clauses: THM;
val z_dom_clauses: THM;
val z_ran_clauses: THM;
val z_id_clauses: THM;
val z_⨾_clauses: THM;
val z_∘_clauses: THM;
val z_◁_clauses: THM;
val z_▷_clauses: THM;
val z_⩤_clauses: THM;
val z_⩥_clauses: THM;
val z_rel_inv_clauses: THM;
val z_rel_image_clauses: THM;
val z_trans_closure_clauses: THM;
val z_reflex_closure_clauses: THM;
val z_⊕_clauses: THM;
```

Description The ML bindings of the theorems (other than consistency ones) in theory *z-_relations*.

SML

val **z_binding_eq_conv3** : *CONV*;

Description A conversion for eliminating equations of bindings to an elementwise equality clause. In general this does:

Conversion

$$\frac{}{\vdash (b_1 = b_2) \Leftrightarrow (b_1.s_1 = b_2.s_1) \land (b_1.s_2 = b_2.s_2) \land ...} \quad \begin{array}{l} z\_binding\_eq\_conv3 \\ ⌜_Z b_1 = b_2⌝ \end{array}$$

However, it will expand on either side θ-terms into binding displays, and also use $z\_sel_s\_conv$ on selections from binding displays (whether from θ-terms or otherwise).

Errors

42013 ?0 is not of the form ⌜Z binding = binding⌝

SML

val **z_sel$_t$_conv** : *CONV*;

Description This conversion carries out the selection from a tuple display.

Conversion

$$\frac{}{\vdash (t_1,...,t_i,...,t_n).i = t_i} \quad \begin{array}{l} z\_sel_t\_conv \\ ⌜_Z (t_1,...,t_i,...,t_n).i⌝ \end{array}$$

$x \mapsto y$ will be treated as a 2-tuple.

See Also $z\_sel_t\_lang\_conv$

Errors

47185 ?0 is not a Z tuple selection
42006 ?0 is not of the form ⌜Z (x,...).i⌝

SML

val **z_tuple_eq_conv** : *CONV*;

Description A conversion for eliminating tuples over equality.

Conversion

$$\frac{}{\vdash (t_1,t_2,...) = (u_1,u_2,...) \Leftrightarrow ((t_1 = u_1) \land (t_2 = u_2) \land ...)} \quad \begin{array}{l} z\_tuple\_eq\_conv \\ ⌜_Z (t_1,t_2,...) = (u_1,u_2,...)⌝ \end{array}$$

$x \mapsto y$ will be treated as a 2-tuple.

See Also *z_tuple_lang_eq_conv*

Errors

42003 ?0 is not of the form: ⌜Z (x1,...) = (y1,...)⌝

SML

```
val z_tuple_eq_conv1 : CONV;
```

Description A conversion for eliminating tuples over equality to an elementwise equality clause.

Conversion

$$\frac{}{\vdash (t1 = t2) \Leftrightarrow (t1.1 = t2.1 \land ...)} \quad \begin{array}{l} z\_tuple\_eq\_conv \\ ⓩt1 = t2⌝ \end{array}$$

This will then use $z\_sel_t\_conv$ to eliminate explicit tuples. $x \mapsto y$ will be treated as a 2-tuple.

See Also *z_tuple_lang_eq_conv*

Errors

83001 ?0 is not of the form: ⓩtuple1 = tuple2⌝

SML

```
val z_tuple_intro_conv : CONV;
```

Description This conversion carries out the elimination of a tuple display of tuple selections from the same tuple.

Conversion

$$\frac{}{\vdash (t.1,...,t.n) = t} \quad \begin{array}{l} z\_tuple\_intro\_conv \\ ⓩ(t.1,...,t.n)⌝ \end{array}$$

where n is the arity of t. $x \mapsto y$ will be treated as a 2-tuple.

See Also *z_tuple_lang_intro_conv*

Errors

42005 ?0 is not of the form: ⓩ(t.1,...,t.n)⌝

SML

```
val z_↦_def : THM;
val z_dom_def : THM;
val z_ran_def : THM;
val z_id_def : THM;
val z_⨾_def : THM;
val z_∘_def : THM;
val z_◁_def : THM;
val z_▷_def : THM;
val z_⩤_def : THM;
val z_⩥_def : THM;
val z_rel_inv_def : THM;
val z_rel_image_def : THM;
val z_tc_def : THM;
val z_rtc_def : THM;
val z_⊕_def : THM;
```

Description These are the definitions of the theory *z_relations*.

9.2.3 Z Functions

SML

```
signature ZFunctions = sig
```

Description This provides the basic proof support for the Z library functions. It creates the theory *z_functions*.

SML

```
(∗ Proof Context: ′z_∈_fun ∗)
```

Description A component proof context for handling the membership of Z functions created by Z library operations. Expressions and predicates treated by this proof context are constructs formed from:

$⤀, ↠, ⤖, ⤔, ↣, ⇸, →$

Contents

Rewriting:

Stripping theorems:

Stripping conclusions:

All three of the above also have theorems concerning the membership of terms generated by the following operators:

$⤀, ↠, ⤖, ⤔, ↣, ⇸, →$

Stripping also contains the above in negated forms.

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_sets*. It is intended to be used with proof context "′z_∈_rel". It is not intended to be mixed with HOL proof contexts.

SML

(∗ *Proof Context*: ′**z_fun_alg** ∗)

Description A component proof context for handling the simplification of Z functions created by Z library operations. Expressions and predicates treated by this proof context are constructs formed from:

$\nrightarrow$, →, ⤔, ↣, ⤀, ↠, ⤖

Contents

Rewriting:

z_⇸_clauses, *z_→_clauses*, *z_⤔_clauses*,
z_↣_clauses, *z_↠_clauses*, *z_⤀_clauses*, *z_⤖_clauses*

Stripping theorems:

z_⇸_clauses, *z_→_clauses*, *z_⤔_clauses*,
z_↣_clauses, *z_↠_clauses*, *z_⤀_clauses*, *z_⤖_clauses*
Expressed as membership statements as necessary, using ∈_*C*.
All also pushed through ¬.

Stripping conclusions:

z_⇸_clauses, *z_→_clauses*, *z_⤔_clauses*,
z_↣_clauses, *z_↠_clauses*, *z_⤀_clauses*, *z_⤖_clauses*
Expressed as membership statements as necessary, using ∈_*C*.
All also pushed through ¬.

Rewriting canonicalisation:

Automatic proof procedures are respectively *z_basic_prove_tac*, *z_basic_prove_conv*, and no existence prover.

Usage Notes It requires theory *z_sets*. The proof context ensures that its simplifications will be attempted before more general rules concerned membership of set operators are used (including extensionality rules).

It is not intended to be mixed with HOL proof contexts.

SML

(∗ *Proof Context*: **z_fun_ext** ∗)

Description An aggressive complete proof context for reasoning about Z functions. When stripping or rewriting it attempts to reduce any predicate concerning function constructs to predicate calculus. As a side effect relational and set constructs are also reduced to predicate calculus. The proof context is a merge of:

z_rel_ext − extensional reasoning about relations (and sets)
′z_∈_fun − membership of function constructs
′z_fun_alg − simplifications of function constructs

It requires the theory "z_functions".

SML

val **z_⇸_def** : *THM*;	*val* **z_⤔_def** : *THM*;
val **z_↣_def** : *THM*;	*val* **z_⤀_def** : *THM*;
val **z_↠_def** : *THM*;	*val* **z_⤖_def** : *THM*;

Description These are the ML bindings of the defining theorems in the theory *z_functions*.

SML
```
val z_⇸_thm : THM;                          val z_⇸_thm1 : THM;
val z_→_thm : THM;                          val z_→_app_thm : THM;
val z_⤔_thm : THM;                          val z_↣_thm : THM;
val z_⤀_thm : THM;                          val z_↠_thm : THM;
val z_⤗_thm : THM;                          val z_∈_first_thm : THM;
val z_∈_second_thm : THM;                   val z_→_app_∈_rel_thm : THM;
val z_→_app_eq_⇔_∈_rel_thm : THM;           val z_→_∈_rel_⇔_app_eq_thm : THM;
val z_⇸_clauses : THM;                      val z_→_clauses : THM;
val z_⤔_clauses : THM;                      val z_↣_clauses : THM;
val z_⤀_clauses : THM;                      val z_↠_clauses : THM;
val z_⤗_clauses : THM;                      val z_fun_app_clauses : THM;
val z_fun_∈_clauses : THM;                  val z_fun_dom_clauses : THM;
val z_fun_ran_clauses : THM;
```

Description These are the ML bindings of the theorems in the theory *z_functions*.

9.2.4 Z Numbers

SML
```
signature ZNumbers = sig
```

Description This provides the basic proof support for the Z library relations. It creates the theory *z_numbers*.

SML

(∗ *Proof Context*: ′**z_numbers** ∗)

Description A component proof context for handling the basic arithmetic operations for Z.

Expressions and predicates treated by this proof context are constructs formed from:

$+, *, -, abs, div, mod, \mathbb{Z}, \leq, <, \geq, >, =, \mathbb{N}$

Contents

Rewriting:

z_plus_conv, *z_times_conv*, *z_subtract_minus_conv*
z_abs_conv, *z_div_conv*, *z_mod_conv*
z_ℤ_eq_conv, *z_≤_conv*, *z_less_conv*
z_≥_≤_conv, *z_greater_less_conv*, *z_∈_ℕ_conv*
z_plus_clauses, *z_minus_clauses*, *z_≤_clauses*
z_less_clauses, *z_¬_≤_thm*, *z_¬_less_thm*,
z_∈_ℕ1_thm, *simple_z_dot_conv*, *z_∈_dot_dot_conv*

Stripping theorems:

z_ℤ_eq_conv, *z_≤_conv*, *z_less_conv*
z_≥_≤_conv, *z_greater_less_conv*, *z_∈_ℕ_conv*
z_plus_clauses, *z_minus_clauses*, *z_≤_clauses*
z_less_clauses, *z_∈_ℕ1_thm*, *z_∈_dot_dot_conv*
and all the above pushed through ¬

z_¬_≤_thm, *z_¬_less_thm*, *z_≤_conv*, *z_less_conv*

Stripping conclusions: as for stripping theorems.

Rewriting canonicalisation: blank.

𝕌-simplification:

$\vdash \mathbb{Z} = \mathbb{U}$

Automatic proof procedures: *z_basic_prove_tac*, *z_basic_prove_conv*.

Automatic existence prover: blank.

See Also Proof context ′*z_numbers1*

SML

(∗ *Proof Context*: ′**z_numbers1** ∗)

Description A component proof context for handling the basic arithmetic operations for Z. It is distinct from ′*z_numbers* by its normalising all inequalities to ≤.

Expressions and predicates treated by this proof context are constructs formed from:

+, ∗, −, *abs*, *div*, *mod*, ℤ, ≤, <, ≥, >, =, ℕ

Contents

Rewriting:

z_plus_conv, *z_times_conv*, *z_subtract_minus_conv*
z_abs_conv, *z_div_conv*, *z_mod_conv*
z_ℤ_eq_conv, *z_≤_conv*, *z_less_conv*
z_≥_≤_conv, *z_greater_less_conv*, *z_∈_ℕ_conv*
z_plus_clauses, *z_minus_clauses*, *z_≤_clauses*
z_less_clauses, *z_¬_less_thm*,
z_∈_ℕ₁_thm, *z_simple_dot_dot_conv*, *z_∈_dot_dot_conv*,
conv_rule (*ONCE_MAP_C eq_sym_conv*) *z_¬_≤_thm*

The final conversion to < to ≤ will only occur if no other rewriting applies.

Stripping theorems:

z_ℤ_eq_conv, *z_≤_conv*, *z_less_conv*
z_≥_≤_conv, *z_greater_less_conv*, *z_∈_ℕ_conv*
z_plus_clauses, *z_minus_clauses*, *z_≤_clauses*
z_less_clauses, *z_∈_ℕ₁_thm*, *z_∈_dot_dot_conv*
and all the above pushed through ¬

z_¬_less_thm, *z_≤_conv*, *z_less_conv*,
conv_rule (*ONCE_MAP_C eq_sym_conv*) *z_¬_≤_thm*

Stripping conclusions: as for stripping theorems.

Rewriting canonicalisation: blank.

𝕌-simplification:

⊢ ℤ = 𝕌

Automatic proof procedures: *z_basic_prove_tac*, *z_basic_prove_conv*.

Automatic existence prover: blank.

SML

```
val dest_z_≤ : TERM -> TERM * TERM;
val dest_z_≥ : TERM -> TERM * TERM;
val dest_z_abs : TERM -> TERM;
val dest_z_div : TERM -> TERM * TERM;
val dest_z_greater : TERM -> TERM * TERM;
val dest_z_less : TERM -> TERM * TERM;
val dest_z_minus : TERM -> TERM;
val dest_z_mod : TERM -> TERM * TERM;
val dest_z_plus : TERM -> TERM * TERM;
val dest_z_signed_int : TERM -> INTEGER;
val dest_z_subtract : TERM -> TERM * TERM;
val dest_z_times : TERM -> TERM * TERM;
```

Description These are derived destructor functions for the Z basic arithmetic operations. An optionally signed integer literal, *signed_int*, is taken to be either a numeric literal or the result of applying (∼_) to a numeric literal. The other constructors correspond directly to the arithmetic operations of the theory *z_numbers* with alphabetic names assigned to give a valid ML name as needed (*greater* :>, *less* :<, *minus*: ∼, *plus* : +, *subtract* : −, *times* : ∗).

As usual, there are also corresponding discriminator (*is_* ...) and constructor functions (*mk_* ...). For programming convenience, *dest_z_signed_int* returns *0* and *is_z_signed_int* returns *true* when applied to ∼*0*, but *mk_z_signed_int* cannot be used to construct such a term.

Errors

```
86101 ?0 is not of the form ⌜Z i ≤ j⌝
86102 ?0 is not of the form ⌜Z i ≥ j⌝
86103 ?0 is not of the form ⌜Z abs i⌝
86104 ?0 is not of the form ⌜Z i div j⌝
86105 ?0 is not of the form ⌜Z i > j⌝
86106 ?0 is not of the form ⌜Z i < j⌝
86107 ?0 is not of the form ⌜Z ∼i⌝
86108 ?0 is not of the form ⌜Z i mod j⌝
86109 ?0 is not of the form ⌜Z i + j⌝
86110 ?0 is not an optionally signed integer literal
86111 ?0 is not of the form ⌜Z i − j⌝
86112 ?0 is not of the form ⌜Z i ∗ j⌝
```

SML

```
val is_z_≤ : TERM -> bool;
val is_z_≥ : TERM -> bool;
val is_z_abs : TERM -> bool;
val is_z_div : TERM -> bool;
val is_z_greater : TERM -> bool;
val is_z_less : TERM -> bool;
val is_z_minus : TERM -> bool;
val is_z_mod : TERM -> bool;
val is_z_plus : TERM -> bool;
val is_z_signed_int : TERM -> bool;
val is_z_subtract : TERM -> bool;
val is_z_times : TERM -> bool;
```

Description These are derived discriminator functions for the Z basic arithmetic operations. See the documentation for the destructor functions (*dest_z_plus* etc.) for more information.

SML

```
val mk_z_≤ : TERM * TERM -> TERM;
val mk_z_≥ : TERM * TERM -> TERM;
val mk_z_abs : TERM -> TERM;
val mk_z_div : TERM * TERM -> TERM;
val mk_z_greater : TERM * TERM -> TERM;
val mk_z_less : TERM * TERM -> TERM;
val mk_z_minus : TERM -> TERM;
val mk_z_mod : TERM * TERM -> TERM;
val mk_z_plus : TERM * TERM -> TERM;
val mk_z_signed_int : INTEGER -> TERM;
val mk_z_subtract : TERM * TERM -> TERM;
val mk_z_times : TERM * TERM -> TERM;
```

Description These are derived constructor functions for the Z basic arithmetic operations. See the documentation for the destructor functions (*dest_z_plus* etc.) for more information.

Errors

86201 ?0 does not have type ℤ

SML

```
val z_cov_induction_tac : TERM -> TACTIC
```

Description A course of values induction tactic for a subset of the integers. To prove $j \leq x \Rightarrow t$, it suffices to prove $t[i/x]$ on the assumptions that $j \leq i$ and $\forall k \bullet j \leq k \land k < i \Rightarrow t[k/x]$.

(Course of values induction is sometimes called complete induction.) The term argument must appear free in the conclusion of the goal. It must also appear once, and only once, in the assumptions, in an assumption of the form $j \leq x$.

Tactic

$$\frac{\{\ \Gamma,\ j \leq x\}\ t[x]}{\{\ \Gamma,\ j \leq x\}\ t[j/x]\ ;} \qquad z\_cov\_induction\_tac\ ⌜_Z x⌝$$

strip $\{j \leq i,\ ⌜_Z \forall k \bullet\ j \leq k \land k < x \Rightarrow t[k]⌝,\ \Gamma\}\ t[x]$

See Also *z_ℤ_cases_thm*, *z_intro_∀_tac*, *z_ℕ_induction_tac*,

z_ℤ_induction_tac, *z_≤_induction_tac*

Errors As for *z_≤_induction_tac*.

SML

```
val z_simple_dot_dot_conv : CONV;
val z_∈_dot_dot_conv : CONV;
```

Description The first of these two conversions simplifies certain *dots* terms, the second, given a membership of a *dodts* expression, first tries the simplifications, and whether or not that succeeds, expands the membership.

Conversion

$$\frac{}{\vdash (x \mathrel{..} x) = \{x\}} \qquad z\_simple\_dot\_dot\_conv \ \ulcorner_Z x \mathrel{..} x\urcorner$$

and

Conversion

$$\frac{}{\vdash (n1 \mathrel{..} n2) = \{\}} \qquad z\_simple\_dot\_dot\_conv \ \ulcorner_Z n1 \mathrel{..} n2\urcorner$$

where *n1* is a numeric literal less than the numeric literal *n2*.

Conversion

$$\frac{}{\vdash x \in y \mathrel{..} y \Leftrightarrow x = y} \qquad z\_\in\_dot\_dot\_conv \ \ulcorner_Z x \in y \mathrel{..} y\urcorner$$

Conversion

$$\frac{}{\vdash x \in n1 \mathrel{..} n2 \Leftrightarrow false} \qquad z\_\in\_dot\_dot\_conv \ \ulcorner_Z x \in n1 \mathrel{..} n2\urcorner$$

where *n1* is a numeric literal less than the numeric literal *n2*.

Conversion

$$\frac{}{\vdash x \in low \mathrel{..} high \Leftrightarrow low \leq x \land x \leq high} \qquad z\_\in\_dot\_dot\_conv \ \ulcorner_Z x \in low \mathrel{..} high\urcorner$$

See Also *z_dot_dot_conv*

Errors

86001 *?0 is not of the form:* $\ulcorner_Z low \mathrel{..} high\urcorner$ *where the expresion can be simplified*

86002 *?0 is not of the form:* $\ulcorner_Z x \in low \mathrel{..} high\urcorner$

SML

```
val z_∈_ℕ_thm : THM;                     val z_¬_≤_thm : THM;
val z_¬_ℕ_thm : THM;                     val z_¬_less_thm : THM;
val z_≤_≤_0_thm : THM;                   val z_≤_antisym_thm : THM;
val z_≤_cases_thm : THM;                 val z_≤_clauses : THM;
val z_≤_induction_thm : THM;             val z_≤_less_trans_thm : THM;
val z_≤_plus_ℕ_thm : THM;                val z_≤_refl_thm : THM;
val z_≤_trans_thm : THM;                 val z_ℕ_¬_minus_thm : THM;
val z_ℕ_¬_plus1_thm : THM;               val z_ℕ_abs_minus_thm : THM;
val z_ℕ_cases_thm : THM;                 val z_ℕ_induction_thm : THM;
val z_ℕ_plus1_thm : THM;                 val z_ℕ_plus_thm : THM;
val z_ℕ_thm : THM;                       val z_ℕ_times_thm : THM;
val z_ℤ_cases_thm : THM;                 val z_ℤ_cases_thm1 : THM;
val z_ℤ_eq_thm : THM;                    val z_ℤ_induction_thm : THM;
val z_0_ℕ_thm : THM;                     val z_abs_ℕ_thm : THM;
val z_abs_eq_0_thm : THM;                val z_abs_minus_thm : THM;
val z_abs_plus_thm : THM;                val z_abs_thm : THM;
val z_abs_times_thm : THM;               val z_cov_induction_thm : THM;
val z_div_mod_unique_thm : THM;          val z_int_homomorphism_thm : THM;
val z_less_≤_trans_thm : THM;            val z_less_clauses : THM;
val z_less_irrefl_thm : THM;             val z_less_trans_thm : THM;
val z_minus_ℕ_≤_thm : THM;               val z_minus_clauses : THM;
val z_minus_thm : THM;                   val z_minus_times_thm : THM;
val z_plus0_thm : THM;                   val z_plus_assoc_thm : THM;
val z_plus_assoc_thm1 : THM;             val z_plus_clauses : THM;
val z_plus_comm_thm : THM;               val z_plus_cyclic_group_thm : THM;
val z_plus_minus_thm : THM;              val z_plus_order_thm : THM;
val z_times0_thm : THM;                  val z_times1_thm : THM;
val z_times_assoc_thm : THM;             val z_times_assoc_thm1 : THM;
val z_times_clauses : THM;               val z_times_comm_thm : THM;
val z_times_eq_0_thm : THM;              val z_times_order_thm : THM;
val z_times_plus_distrib_thm : THM;      val z_≤_less_eq_thm : THM;
val z_𝔽_thm : THM;                       val z_𝔽1_thm : THM;
val z_𝔽_empty_thm : THM;                     val z_∈_ℕ1_thm : THM;
```

Description These are the ML value bindings for the theorems saved in the theory *z_numbers*.

SML
```
val z_≤_induction_tac : TERM -> TACTIC
```

Description An induction tactic for a subset of the integers. To prove $j \leq x \Rightarrow t$, it suffices to prove $t[j/x]$ and to prove $t[x + 1/x]$ on the assumptions t and $j \leq x$. The term argument must be a variable of type ⌜:ℤ⌝ and must appear free in the conclusion of the goal. It must also appear once, and only once in the assumptions, in an assumption of the form $j \leq x$.

Tactic

$$\frac{\{\ \Gamma,\ j \leq x\}\ t[x]}{\{\ \Gamma,\ j \leq x\}\ t[j/x]\ ;\ strip\ \{t[x],\ j \leq x,\ \Gamma\}\ t[x+1]}\quad z\_\leq\_induction\_tac\ ⓩx⌝$$

See Also *z_ℤ_cases_thm*, *z_intro_∀_tac*, *z_ℕ_induction_tac*,

z_ℤ_induction_tac, *z_cov_induction_tac*

Errors

86401 ?0 is not a variable of type ⌜:ℤ⌝
86402 A term of the form ⓩj ≤ i⌝ where i is the induction variable could not be found in the assumptions
86403 ?0 appears free in more than one assumption of the goal
86404 ?0 does not appear free in the conclusions of the goal

SML
```
val z_ℕ_induction_tac : TACTIC
```

Description This tactic implements induction over the natural numbers in Z: to prove $x \in \mathbb{N} \Rightarrow t$, it suffices to prove $t[0/x]$ and to prove $t[x + 1/x]$ on the assumption that t. The conclusion of the goal must have the form $x \in \mathbb{N} \Rightarrow t$.

Tactic

$$\frac{\{\ \Gamma\ \}\ x \in \mathbb{N} \Rightarrow t}{\{\ \Gamma\ \}\ t[0/x]\ ;\ strip\{t,\ \Gamma\}\ t[x+1/x]}\quad z\_\mathbb{N}\_induction\_tac$$

See Also *z_ℤ_cases_thm*, *z_intro_∀_tac*, *z_ℤ_induction_tac*,

z_≤_induction_tac, *z_cov_induction_tac*

Errors As for *gen_induction_tac1*.

SML
```
val z_ℕ_plus_conv : CONV;              val z_ℕ_times_conv : CONV;

val z_subtract_minus_conv : CONV;      val z_greater_less_conv : CONV
val z_≥_≤_conv : CONV;                 val z_∈_ℕ_conv : CONV;

val z_plus_conv : CONV;                val z_times_conv : CONV
val z_abs_conv : CONV;                 val z_div_conv : CONV
val z_mod_conv : CONV;                 val z_≤_conv : CONV
val z_less_conv : CONV                 val z_ℤ_eq_conv : CONV
```

Description These conversions are used to perform evaluation of arithmetic expressions involving numeric literal operands. The normal interface to the conversion is via the proof context *′z_numbers* and other proof contexts which contain it.

The first block above gives conversions to evaluate expressions of the form *i op j* where *i* and *j* are numeric literals and *op* is one of + or ∗. The second block gives conversions to transform terms of the form $i - j$, $i > j$, $i > j$ and $i \in \mathbb{N}$ into $i + \sim j$, $j < i$, $j \leq i$ and $0 \leq i$ respectively. The third block give conversions which evaluate expressions of the form *i op j* or *abs i*, where *op* is one of +, ∗, *div*, *mod*, ≤, <, or =, and where *i* and *j* are signed integer literals (i.e., either numeric literals or of the form ∼*k*, where *k* is a numeric literal). Thus the second block of conversions transform expressions of the form $i - j$, $i > j$, $i \geq j$ and $i \in \mathbb{N}$ into a form which can be evaluated by the conversions in the third block if *i* and *j* are signed literals.

Errors
```
86301 ?0 is not of the form ?1 where ⓩi⌝ and ⓩj⌝ are numeric literals
86302 ?0 is not of the form ?1
86303 ?0 is not of the form ?1 where ⓩi⌝ and ⓩj⌝ are optionally signed
      numeric literals
```

SML
```
val z_ℤ_def : THM;                     val z_ℕ_def : THM;
val z_arith_def : THM;                 val z_inequality_def : THM;
val z_ℕ1_def : THM;                    val z_succ_def : THM;
val z_iter_def : THM;                  val z_dot_dot_def : THM;
val z_𝔽_def : THM;                     val z_𝔽1_def : THM;
val z_hash_def : THM;                  val z_⇸⇸_def : THM;
val z_⤀_def : THM;                     val z_min_def : THM;
val z_max_def : THM;                   val z'int_def : THM
```

Description These are the ML bindings of the definitions of the theory *z_numbers*.

SML
```
val z_ℤ_induction_tac : TERM -> TACTIC
```

Description An induction-like tactic for the integers, based on the fact that any subset of the integers containing *1* and closed under negation and addition must contain every integer.

Tactic
```
              { Γ } t
 ==========================================    z_ℤ_induction_tac ⌜x⌝
          { Γ } t[1/x] ;
     strip{t[i/x], Γ} t[~i/x] ;
  strip{t[i/x] ∧ t[j/x], Γ} t[i+j/x]
```

See Also *z_ℤ_cases_thm*, *z_intro_∀_tac*, *z_ℕ_induction_tac*,

z_≤_induction_tac, *z_cov_induction_tac*

Errors As for *gen_induction_tac*.

9.2.5 Z Arithmetic Proof Support

SML
```
signature ZArithmeticTools = sig
```

Description This is the signature of a structure containing arithmetic and an automatic linear arithmetic prover for the integers in Z.

SML
```
(* Proof Context: z_lin_arith *)
(* Proof Context: z_lin_arith1 *)
```

Description "*z_lin_arith*" is a proof context whose main purpose is to supply a decision procedure for problems in linear arithmetic in Z. "*z_lin_arith1*" differs from it only by using "*'z_numbers1*". The proof context provides an interface to the proof context *'ℤ_lin_arith* which provides the analogous facilities for the HOL integers.

Contents The proof context is the result of merging *z_predicates*, *'z_numbers(1)* and *'z_lin_arith*.

Examples *PC_T1*"*z_lin_arith*"*prove_tac*[] will prove any of the following goals:

```
([], ⌜∀a, b, c:ℤ•a ≤ b ∧ (a + b < c + a) ⇒ a < c⌝)
([], ⌜∀a, b, c:ℤ•a ≥ b ∧ ¬ b < c ⇒ a ≥ c⌝)
([], ⌜∀a, b, c:ℤ•a + 2*b < 2*a ⇒ b + b < a⌝)
([], ⌜∀ x, y:ℤ• ¬ (2*x + y = 4 ∧ 4*x + 2*y = 7)⌝)
```

In the following example, an induction reduces the problem to linear arithmetic, and then the automatic proof tactic in *z_lin_arith* completes the proof.

```
set_goal([], ⌜∀b:ℕ•(b + 1)*(b + 1) > 0⌝);
a(PC_T1 "z_library" REPEAT strip_tac);
a( z_≤_induction_tac⌜b⌝ THEN PC_T1 "z_lin_arith" asm_prove_tac[]);
pop_thm();
```

See Also *'z_lin_arith*

Errors The errors reported by the automatic proof tactic are as for *'ℤ_lin_arith*.

SML
```
(∗ Proof Context: 'z_lin_arith ∗)
(∗ Proof Context: 'z_lin_arith1 ∗)
```

Description "'*z_lin_arith*" is a component proof context whose purpose is to supply a decision procedure for problems in linear arithmetic for the integers in Z. "'*z_lin_arith1*" i a copy, only differing in using "'*z_numbers1*".

Contents The rewriting, theorem stripping and conclusion stripping components are those from '*z_numbers* augmented with the following transformations:

```
⌜Z(a ⊕⊕ ℤ) = b⌝      →    ⌜Z a ≤ b ∧ b ≤ a⌝
⌜Z a ≤ b⌝            →    ⌜ML⌜TRY_C z_ℤ_conv ⌜z_ℤ a⌝⌝ ≤ ML⌜TRY_C z_ℤ_conv ⌜b⌝⌝⌝
⌜Z a < b⌝            →    ⌜ML⌜TRY_C z_ℤ_conv ⌜a⌝⌝ < ML⌜TRY_C z_ℤ_conv ⌜b⌝⌝⌝
⌜Z x ∈ ℕ⌝            →    ⌜0 ≤ x⌝
```

(where all variables are of type ℤ). The effect of the above scheme is to transformed any Z equation or inequality in ≤ or < into an equivalent inequality over the HOL integers. The automatic proof procedures for the proof context are (slight adaptations of) the ones in the proof context *ℤ_lin_arith*, which can then automatically prove the transformed form if it is a theorem of linear arithmetic.

The automatic proof components is an interface to the one for '*ℤ_lin_arith*. Other components are as for '*z_numbers*.

Examples A typical use of the proof context would be to solve problems containing a mixture of (linear) arithmetic and set theory.

For example, *MERGE_PCS_T1*["*z_sets_ext*", "'*z_lin_arith*"]*prove_tac*[]will prove any of the following goals:

```
([], ⌜Z ∀m:ℤ• {i:ℤ | m ≤ i ∧ i < m+3} = {m, m+1, m+2}⌝)
([], ⌜Z {i, j : ℤ | 30∗i = 105∗j} = {i, j : ℤ | 2∗i = 7∗j}⌝)
([], ⌜Z {i : ℤ | 5∗i = 6∗i} = {0}⌝)
```

See Also *z_lin_arith*, *z_numbers*, '*z_numbers*

Errors The errors reported by the automatic proof tactic are as for '*ℤ_lin_arith*.

SML
```
val z_anf_conv : CONV;
```

Description *z_anf_conv* is a conversion which proves theorems of the form ⊢ $t1 = t2$ where $t1$ is a Z expression formed from atoms of type ℤ and $t2$ is in what we may call additive normal form, i.e. it has the form: $t_1 + t_2 + ...$, where the t_i have the form $s_1 * s_2 * ...$where the s_i are atoms. Here, by atom we mean an expression which is not of the form $t_1 + t_2 + ...$or $s_1 * s_2 * ...$.

The summands t_i and, within them, the factors s_j are given in increasing order with respect to the ordering on terms analogous to that given by the function *z_term_order*, q.v. Arithmetic computation is carried out on atoms to ensure that at most one of the summands is a numeric literal and that, within each summand, at most one factor is a numeric literal. Any literal appears at the beginning of its factor or summand and addition of *0* or multiplication by *1* is simplified out. Any signs are moved to the first factor in each summand.

The conversion fails with error number 106010 if there no changes can be made to the term.

Errors
```
106010 ?0 is not of type ⌜:ℤ⌝ or is already in additive normal form
```

SML

```
val z_ℤ_conv : CONV;
val ℤ_z_conv : CONV;
```

Description In the theory *z_arithmetic_tools*, isomorphisms, *z_ℤ* and *ℤ_z*, are defined between the Z integers and the HOL integers. These may be used to translate an arithmetic problem in Z into one in HOL. These conversions implement this translation and its inverse.

The operators handled by the conversions are +, ∗, ∼ and −.

The translation to HOL is carried out according to the following scheme:

```
z_ℤ⌜Z t1 + t2⌝  →   ⌜z_ℤ t1 + z_ℤ  t2⌝
z_ℤ⌜Z t1 ∗ t2⌝  →   ⌜z_ℤ t1 ∗ z_ℤ  t2⌝
z_ℤ⌜Z ∼t1⌝      →   ⌜∼ z_ℤ t1⌝
z_ℤ⌜Z ℤ_z t1⌝   →   ⌜t1⌝
z_ℤ⌜Z Z'Int c⌝  →   ⌜ℕℤ c⌝
```

z_ℤ_conv implements the above scheme recursively to translate the result of applying *z_ℤ* to a Z arithmetic expression into HOL.

ℤ_z_conv is the analogue of *z_ℤ_conv*, performing the translation of *ℤ_z* applied to a HOL integer arithmetic expression into Z.

Uses Tactic programming.

See Also *z_anf_conv*, *z_lin_arith*, *'z_lin_arith*

Errors

```
106001 ?0 is not of the form ⌜z_ℤ ⌜Z t⌝⌝ where ⌜Z t⌝ is constructed from +, ∼, −, ∗ or
        integer constants
106002 ?0 is not of the form ⌜ℤ_z ⌜t⌝⌝ where ⌜t⌝ is constructed from +, ∼, −, ∗ or
        integer constants
```

SML

```
val z_ℤ_def : THM;
val ℤ_z_def : THM;
val z_ℤ_plus_thm : THM;
val z_ℤ_times_thm : THM;
val z_ℤ_subtract_thm : THM;
val z_ℤ_minus_thm : THM;
val ℤ_z_plus_thm : THM;
val ℤ_z_times_thm : THM;
val ℤ_z_subtract_thm : THM;
val ℤ_z_minus_thm : THM;
val z_ℤ_one_one_thm : THM;
val ℤ_z_one_one_thm : THM;
val z_≤_ℤ_≤_thm : THM;
val z_less_ℤ_less_thm : THM;
```

Description These are the Standard ML bindings for the theorems saved in the theory *z_arithmetic_tools* which provides isomorphisms between the ring of integers in HOL and the ring of integers in Z. The main purpose of this theory is to allow proof procedures for HOL integers to be adapted to work with Z. The most common way of using these isomorphisms is via the proof context *z_lin_arith*, q.v.

9.2.6 Z Sequences

SML

signature **ZSequences** = *sig*

Description This provides the basic proof support for the Z library sequences. It creates the theory *z_sequences*.

SML

val **z_seq_def** : *THM*;
val **z_seq₁_def** : *THM*;
val **z_iseq_def** : *THM*;
val **z_⁀_def** : *THM*;
val **z_head_def** : *THM*;
val **z_last_def** : *THM*;
val **z_tail_def** : *THM*;
val **z_front_def** : *THM*;
val **z_rev_def** : *THM*;
val **z_squash_def** : *THM*;
val **z_↿_def** : *THM*;
val **z_↾_def** : *THM*;
val **z_⁀/_def** : *THM*;
val **z_disjoint_def** : *THM*;
val **z_partition_def** : *THM*;

Description These are the ML bindings of the definitions of the theory of *z_sequences*.

9.2.7 Z Finiteness and Sequences

SML

signature **ZFunctions1** = *sig*

Description This provides additional proof support for the Z library functions. It creates the theory *z_functions1*.

SML

signature **ZNumbers1** = *sig*

Description This provides additional proof support for the Z library functions. It creates the theory *z_functions1*.

SML

signature **ZSequences1** = *sig*

Description This provides additional proof support for the Z library sequences. It creates the theory *z_sequences1*.

SML

```
val z_dot_dot_clauses : THM;
val z_dot_dot_plus_thm : THM;
val z_less_cases_thm : THM;
val z_≤_≤_plus1_thm : THM;
val z_dot_dot_diff_thm : THM;
val z_dot_dot_∪_thm : THM;
val z_dot_dot_∩_thm : THM;
val z_empty_𝔽_thm : THM;
val z_𝔽_∪_singleton_thm : THM;
val z_𝔽_thm1 : THM;
val z_𝔽_induction_thm : THM;
val z_𝔽_size_thm : THM;
val z_𝔽_size_thm1 : THM;
val z_⊆_𝔽_thm : THM;
val z_size_empty_thm : THM;
val z_size_singleton_thm : THM;
val z_size_dot_dot_thm : THM;
val z_size_⤀_thm : THM;
val z_size_seq_thm : THM;
val z_size_∪_singleton_thm : THM;
val z_𝔽_∩_thm : THM;
val z_𝔽_diff_thm : THM;
val z_size_∪_thm : THM;
val z_⋃_𝔽_thm : THM;
val z_size_diff_thm : THM;
val z_size_ℕ_thm : THM;
val z_size_mono_thm : THM;
val z_size_∪_≤_thm : THM;
val z_size_eq_thm : THM;
val z_size_0_thm : THM;
val z_size_1_thm : THM;
val z_size_2_thm : THM;
val z_size_pair_thm : THM;
val z_size_×_thm : THM;
val z_size_≤_1_thm : THM;
val z_size_dot_dot_thm1 : THM;
val z_pigeon_hole_thm : THM;
val z_𝔽_ℙ_thm : THM;
val z_div_thm : THM;
val z_mod_thm : THM;
val z_abs_pos_thm : THM;
val z_abs_neg_thm : THM;
val z_abs_≤_times_thm : THM;
val z_abs_0_less_thm : THM;
val z_0_less_times_thm : THM;
val z_times_less_0_thm : THM;
val z_∈_succ_thm :THM;
val z_succ⁰_thm :THM;
val z_succⁿ_thm :THM;
val z_succ^minus_n_thm :THM;
```

Description These are the ML bindings of the theorems in the theory *z_numbers1*.

SML

```
val z_dot_dot_conv : CONV;
```

Description This conversion expands a range between two integer literals into a set display:

Example

```
z_dot_dot_conv ⓩ1 .. 5⌝ gives
        ⊢ 1 .. 5 = {1, 2, 3, 4, 5}
```

Errors

```
107002?0 is not of the form ⓩa .. b⌝ where ⓩa⌝ and ⓩb⌝ are integer literals
```

SML
```
val z_seqd_app_conv : CONV;
val z_size_seqd_conv : CONV;
val z_seqd_eq_conv : CONV;
```

Description Conversions for sequence displays.

z_seqd_app_conv applies to terms of the form sm, where s is a sequence display and m is a numeric literal.

z_size_seqd_conv

Description applies to terms of the form $\#s$, where s is a sequence display.

z_seqd_eq_conv

Description applies to terms of the form $s\_1 = s\_2$, where s_1 and s_2 are sequence displays.

Errors
```
107011 ?0 is not of the form ⌜⟨t1, ...⟩ m⌝
107012 ?0 is not a positive integer literal
107013 ?0 is not a valid index for the sequence ?1
107020 ?0 is not of the form ⌜⟨t1, ...⟩ = ⟨u1, ...⟩⌝
107021 ?0 is not of the form ⌜#⟨t1, ...⟩⌝
```

SML
```
val z_seq_induction_tac : TERM -> TACTIC;
val z_seq_induction_tac1 : TERM -> TACTIC;
```

Description Induction tactics for Z sequences. To prove $s \in seq\ A \Rightarrow t$, it suffices to prove $t[\langle\rangle/s]$ and to prove $t[s \frown \langle x\rangle/s]$(or $t[\langle x\rangle \frown s/s]$) on the assumptions t, $s \in seq\ A$ and $x \in A$. The term argument must be a variable of the same type as a Z sequence and must appear free in the conclusion of the goal. It must also appear once, and only once, in an assumption of the form $s \in seq\ A$.

Tactic

$$\frac{\{\ \Gamma,\ s \in seq\ A\}\ t[s]}{\{\ \Gamma\ \}\ t[\langle\rangle/s]\ ;\quad strip\ \{t,\ s \in seq\ A,\ x \in A,\ \Gamma\}\ t[s \frown \langle x\rangle/s]} \qquad z\_seq\_induction\_tac\ ⓩs⌝$$

Tactic

$$\frac{\{\ \Gamma,\ s \in seq\ A\}\ t[s]}{\{\ \Gamma\ \}\ t[\langle\rangle/s]\ ;\quad strip\ \{t,\ s \in seq\ A,\ x \in A,\ \Gamma\}\ t[\langle x\rangle \frown s/s]} \qquad z\_seq\_induction\_tac1\ ⓩs⌝$$

Errors
```
107031 A term of the form ⓩs ∈ seq A⌝ where s is the induction variable
       could not be found in the assumptions
107032 ?0 is not a variable
```

SML

```
val z_size_dot_dot_conv : CONV;
```

Description This conversion will calculate the size of a range between two integer literals, including the empty range case when the end of the range is less than the start.

Example

```
z_size_dot_dot_conv ⌜Z# (1 .. 5) ⌝ gives
        ⊢ # (1 .. 5) = 5

z_size_dot_dot_conv ⌜Z# (10 .. 1) ⌝ gives
        ⊢ # (10 .. 1) = 0
```

Errors

```
107001?0 is not of the form ⌜Z#(a .. b)⌝ where ⌜Za⌝ and ⌜Zb⌝ are integer literals
```

SML

```
val z_¬_⁀_empty_thm : THM;            val z_dom_⁀_thm : THM;
val z_dom_seqd_thm : THM;             val z_dom_seq_thm : THM;
val z_⁀_assoc_thm1 : THM;             val z_⁀_assoc_thm : THM;
val z_⁀_def_thm : THM;                val z_⁀_one_one_thm : THM;
val z_⁀_∈_seq_thm1 : THM;             val z_⁀_∈_seq_thm : THM;
val z_⁀_seq_x_thm : THM;              val z_⁀_singleton_thm1 : THM;
val z_⁀_singleton_thm : THM;          val z_⟨⟩_⁀_thm : THM;
val z_⁀_⟨⟩_thm : THM;                 val z_⁀_thm : THM;
val z_prim_seq_induction_thm : THM;   val z_ran_seqd_thm : THM;
val z_∈_seq_app_eq_thm : THM;         val z_seq_cases_thm : THM;
val z_∈_seqd_app_eq_thm : THM;        val z_seqd_⁀_⟨⟩_clauses : THM;
val z_seqd_⁀_rw_thm : THM;            val z_seqd_⁀_thm : THM;
val z_seqd_∈_seq_thm : THM;           val z_seq_induction_thm1 : THM;
val z_seq_induction_thm : THM;        val z_seq_seq_x_thm : THM;
val z_seq_thm1 : THM;                 val z_⟨⟩_seq_thm : THM;
val z_seq_thm : THM;                  val z_seq_u_thm : THM;
val z_singleton_seq_thm : THM;        val z_singleton_seq_x_thm : THM;
val z_size_⁀_thm : THM;               val z_size_seqd_length_thm : THM;
val z_size_seqd_thm : THM;            val z_size_seq_ℕ_thm : THM;
val z_size_seq_thm1 : THM;            val z_size_seq_thm2 : THM;
val z_size_singleton_seq_thm : THM;   val z_⟨⟩_thm : THM;
val z_seqd_eq_thm : THM;
```

Description These are the ML bindings of the theorems in the theory *z_sequences1*.

SML

```
val z_𝔽_induction_tac : TERM -> TACTIC;
```

Description An induction tactic for Z finite sets. To prove $s \in \mathbb{F}\ A \Rightarrow t$, it suffices to prove $t[\{\}/s]$ and to prove $t[s \cup \{x\}/s]$ on the assumptions t, $s \in \mathbb{F}\ A$, $x \in A$ and $\neg x \in s$. The term argument must be a variable of the same type as a Z set and must appear free in the conclusion of the goal. It must also appear once, and only once, in an assumption of the form $s \in \mathbb{F}\ A$.

Tactic

$$\frac{\{\ \Gamma,\ s \in \mathbb{F}\ A\}\ t[s]}{\{\ \Gamma\ \}\ t[\{\}/s]\ ;\quad strip\ \{t,\ s \in \mathbb{F}\ A,\ x \in A,\ \neg x \in s,\ \Gamma\}\ t[s \cup \{x\}/s]} \quad z\_\mathbb{F}\_induction\_tac\ ⓩs⌝$$

Errors

107033 A term of the form ⓩs ∈ 𝔽 A⌝ where s is the induction variable could not be found in the assumptions

SML

```
val z_◁_→_thm : THM;
val z_∈_→_thm : THM;
val z_↣_ran_eq_⤖_thm : THM;
val z_⇸_thm2 : THM;
val z_⤔_thm1 : THM;
val z_↣_thm1 : THM;
val z_ran_∪_thm : THM;
val z_∪_↣_thm : THM;
val z_∪_⤖_thm : THM;
val z_∘_↠_thm : THM;
val z_∘_⤖_thm : THM;
val z_id_thm1 : THM;
val z_simple_swap_⤖_thm : THM;
val z_⤖_trans_thm : THM;
val z_dom_f_→_f_thm : THM;
val z_dom_f_↣_f_thm : THM;
val z_∩_↔_thm : THM;
val z_→_ran_thm : THM;
val z_∩_↣_thm : THM;
val z_∩_↠_thm : THM;
val z_→_diff_singleton_thm : THM;
val z_singleton_app_thm : THM;
val z_→_empty_thm : THM;
val z_⊕_↦_app_thm1 : THM;
val z_⊕_↦_∈_→_thm : THM;
val z_ran_◁_thm : THM;
val z_→_ran_eq_↠_thm : THM;
val z_ran_mono_thm : THM;
val z_↠_thm1 : THM;
val z_→_dom_thm : THM;
val z_∪_↔_thm : THM;
val z_∪_→_thm : THM;
val z_∪_↠_thm : THM;
val z_∘_→_thm : THM;
val z_∘_↣_thm : THM;
val z_rel_inv_⤖_thm : THM;
val z_id_⤖_thm : THM;
val z_swap_⤖_thm : THM;
val z_dom_f_↔_f_thm : THM;
val z_dom_f_↠_f_thm : THM;
val z_dom_f_⤖_f_thm : THM;
val z_↔_ran_thm : THM;
val z_∩_→_thm : THM;
val z_↠_ran_thm : THM;
val z_∩_⤖_thm : THM;
val z_⤖_diff_singleton_thm : THM;
val z_empty_↠_thm : THM;
val z_⊕_↦_app_thm : THM;
val z_dom_⊕_↦_thm : THM;
```

Description These are the ML bindings of the theorems in the theory *z_functions1*.

9.2.8 Z Bags

SML

```
signature ZBags = sig
```

Description This provides the basic proof support for the Z library bags. It creates the theory *z_bags*.

SML

```
signature ZLibrary = sig
```

Description This provides a "marker" theory, indicating the "top" of the Z library theories. It creates the theory *z_library*.

As a side effect, loading this structure will set the current theory to *z_library*, the current proof context to "z_library", and tidies the subgoal package and proof context stacks.

SML

```
(* Proof Context: z_library *)
```

Description A mild complete proof context for handling the manipulation of Z language and library expressions and predicates. Its contents are chosen to be "uncontroversial". That is, any effect is considered to be "almost always the correct thing".

It consists of the merge of the proof contexts:

```
"z_sets_alg", − simplification of set contructs, and Z language
"'z_rel_alg", − simplification of relational contructs
"'z_fun_alg", − simplification of function contructs
"'z_numbers" − simplification of numeric contructs
```

Usage Notes It requires theory *z_bags*.

It is not intended to be mixed with HOL proof contexts or "z_library_ext", which offers an aggressive approach.

SML

```
(* Proof Context: z_library_ext *)
```

Description A agressive complete proof context for handling the manipulation of Z language and library expressions and predicates. Its purpose is to strip or rewrite its input into the Z predicate calculus.

It consists of the merge of the proof contexts:

```
"z_fun_ext", − extensional reasoning about functions (and
        realtions and sets)
"'z_numbers" − simplification of numeric contructs
```

Usage Notes It requires theory *z_bags*.

It is not intended to be mixed with HOL proof contexts or "z_library_ext", which offers an aggressive approach.

SML
```
(∗ Proof Context: z_library1 ∗)
```

Description A mild complete proof context for handling the manipulation of Z language and library expressions and predicates. Its contents are chosen to be "uncontroversial". That is, any effect is considered to be "almost always the correct thing".

It differs from *z_library* only in using *z_numbers1*.

It consists of the merge of the proof contexts:

```
"z_sets_alg", − simplification of set contructs, and Z language
"'z_rel_alg", − simplification of relational contructs
"'z_fun_alg", − simplification of function contructs
"'z_numbers1" − simplification of numeric contructs
```

Usage Notes It requires theory *z_bags*.

It is not intended to be mixed with HOL proof contexts or "z_library_ext", which offers an aggressive approach.

SML
```
(∗ Proof Context: z_library1_ext ∗)
```

Description A agressive complete proof context for handling the manipulation of Z language and library expressions and predicates. Its purpose is to strip or rewrite its input into the Z predicate calculus.

It differs from *z_library* only in using *z_numbers1*.

It consists of the merge of the proof contexts:

```
"z_fun_ext", − extensional reasoning about functions (and
        realtions and sets)
"'z_numbers1" − simplification of numeric contructs
```

Usage Notes It requires theory *z_bags*.

It is not intended to be mixed with HOL proof contexts or "z_library_ext", which offers an aggressive approach.

SML
```
val z_bag_def : THM;        val z_count_def : THM;
val z_in_def : THM; val z_⊎_def : THM;
val z_items_def : THM;
```

Description These are the definitions of the Z bag theory.

9.2.9 Z Reals

SML

(∗ *Proof Context*: **′z_reals** ∗)

Description A component proof context for handling the basic arithmetic operations for real numbers in Z.

Expressions and predicates treated by this proof context are constructs formed from:

$+_R$, $*_R$, $-_R$, $/_R$, $\leq_R$, $<_R$, $\geq_R$, $>_R$, $\hat{}_Z$, $=$

Contents

Rewriting:

z_ℝ_plus_conv, *z_ℝ_times_conv*, *z_ℝ_subtract_conv*
z_ℝ_abs_conv, *z_ℝ_div_conv*, *z_ℝ_mod_conv*
z_ℝ_eq_conv, *z_ℝ_≤_conv*, *z_ℝ_less_conv*
z_ℝ_≥_conv, *z_ℝ_greater_conv*,
z_ℝ_plus_clauses, *z_ℝ_minus_clauses*, *z_ℝ_≤_clauses*
z_ℝ_less_clauses, *z_ℝ_lit_norm_conv*

Stripping theorems:

z_ℝ_eq_conv, *z_ℝ_≤_conv*, *z_ℝ_less_conv*
z_ℝ_≥_conv, *z_ℝ_greater_conv*,
z_ℝ_plus_clauses, *z_ℝ_minus_clauses*, *z_ℝ_≤_clauses*
z_ℝ_less_clauses,
and all the above pushed through ¬

Stripping conclusions: as for stripping theorems.

Rewriting canonicalisation: blank.

Automatic proof procedures: *z_basic_prove_tac*, *z_basic_prove_conv*.

Automatic existence prover: blank.

SML

(∗ *Proof Context*: **z_ℝ_lin_arith** ∗)

Description This is a component proof context whose main purpose is to supply a decision procedure for problems in linear arithmetic for the real numbers in Z.

Contents The rewriting components converts Z real arithmetic expressions into equivalent HOL ones and the automatic proof tactic then uses the HOL linear arithmetic proof context to attempt the proof.

SML
```
val dest_z_ℝ_≤ : TERM -> TERM * TERM;
val dest_z_ℝ_≥ : TERM -> TERM * TERM;
val dest_z_ℝ_ℤ_exp : TERM -> TERM * TERM;
val dest_z_ℝ_abs : TERM -> TERM;
val dest_z_ℝ_frac : TERM -> TERM * TERM;
val dest_z_ℝ_greater : TERM -> TERM * TERM;
val dest_z_ℝ_less : TERM -> TERM * TERM;
val dest_z_ℝ_minus : TERM -> TERM;
val dest_z_ℝ_over : TERM -> TERM * TERM;
val dest_z_ℝ_plus : TERM -> TERM * TERM;
val dest_z_real : TERM -> TERM;
val dest_z_ℝ_subtract : TERM -> TERM * TERM;
val dest_z_ℝ_times : TERM -> TERM * TERM;
```

Description These are derived destructor functions for the Z basic arithmetic operations. An optionally signed integer literal, *signed_int*, is taken to be either a numeric literal or the result of applying (∼_) to a numeric literal. The other constructors correspond directly to the arithmetic operations of the theory *z_numbers* with alphabetic names assigned to give a valid ML name as needed (*greater* :>, *less* :<, *minus*: ∼, *plus* : +, *subtract* : −, *times* : ∗).

As usual, there are also corresponding discriminator (*is_* ...) and constructor functions (*mk_* ...).

Errors
```
117101 ?0 is not of the form ⓩx ≤_R y⌝
117102 ?0 is not of the form ⓩx ≥_R y⌝
117103 ?0 is not of the form ⓩabs_R x⌝
117104 ?0 is not of the form ⓩx /_R y⌝
117105 ?0 is not of the form ⓩx >_R y⌝
117106 ?0 is not of the form ⓩx <_R y⌝
117107 ?0 is not of the form ⓩ∼_R x⌝
117109 ?0 is not of the form ⓩx +_R y⌝
117110 ?0 is not of the form ⓩx /_Z y⌝
117111 ?0 is not of the form ⓩx −_R y⌝
117112 ?0 is not of the form ⓩx ∗_R y⌝
117113 ?0 is not of the form ⓩreal x⌝
```

SML
```
val is_z_ℝ_≤ : TERM -> bool;
val is_z_ℝ_≥ : TERM -> bool;
val is_z_ℝ_ℤ_exp : TERM -> bool;
val is_z_ℝ_abs : TERM -> bool;
val is_z_ℝ_frac : TERM -> bool;
val is_z_ℝ_greater : TERM -> bool;
val is_z_ℝ_less : TERM -> bool;
val is_z_ℝ_minus : TERM -> bool;
val is_z_ℝ_over : TERM -> bool;
val is_z_ℝ_plus : TERM -> bool;
val is_z_real : TERM -> bool;
val is_z_ℝ_subtract : TERM -> bool;
val is_z_ℝ_times : TERM -> bool;
```

Description These are derived discriminator functions for the Z basic arithmetic operations. See the documentation for the destructor functions (*dest_z_plus* etc.) for more information.

SML
```
val mk_z_ℝ_≤ : TERM * TERM -> TERM;
val mk_z_ℝ_≥ : TERM * TERM -> TERM;
val mk_z_ℝ_ℤ_exp : TERM * TERM -> TERM;
val mk_z_ℝ_abs : TERM -> TERM;
val mk_z_ℝ_frac : TERM * TERM -> TERM;
val mk_z_ℝ_greater : TERM * TERM -> TERM;
val mk_z_ℝ_less : TERM * TERM -> TERM;
val mk_z_ℝ_over : TERM * TERM -> TERM;
val mk_z_ℝ_minus : TERM -> TERM;
val mk_z_ℝ_plus : TERM * TERM -> TERM;
val mk_z_real : TERM -> TERM;
val mk_z_ℝ_subtract : TERM * TERM -> TERM;
val mk_z_ℝ_times : TERM * TERM -> TERM;
```

Description These are derived constructor functions for the Z basic arithmetic operations. See the documentation for the destructor functions (*dest_z_plus* etc.) for more information.

Errors
```
117201 ?0 does not have type ℝ
```

SML
```
val z_float_conv : CONV;
```

Description The conversion *z_float_conv* converts a floating point literal into a normalised real literal form.

Errors
```
117006 ?0 is not a Z floating point literal
```

SML
```
val z_ℝ_complete_thm : THM;
val z_ℝ_unbounded_above_thm : THM;
val z_ℝ_unbounded_below_thm : THM;
val z_ℝ_less_antisym_thm : THM;
val z_ℝ_less_cases_thm : THM;
val z_ℝ_less_clauses : THM;
val z_ℝ_less_dense_thm : THM;
val z_ℝ_less_irrefl_thm : THM;
val z_ℝ_less_thm : THM;
val z_ℝ_less_trans_thm : THM;
```

Description These are ML bindings for the theorems that characterise the ordering relation $\_ <_R \_$ on the real numbers.

SML

```
val z_ℝ_eq_≤_thm : THM;
val z_ℝ_eq_thm : THM;
val z_ℝ_less_≤_trans_thm : THM;
val z_ℝ_less_¬_eq_thm : THM;
val z_ℝ_≤_¬_less_thm : THM;
val z_ℝ_≤_antisym_thm : THM;
val z_ℝ_≤_cases_thm : THM;
val z_ℝ_≤_clauses : THM;
val z_ℝ_≤_less_cases_thm : THM;
val z_ℝ_≤_less_trans_thm : THM;
val z_ℝ_≤_refl_thm : THM;
val z_ℝ_≤_thm : THM;
val z_ℝ_≤_trans_thm : THM;
val z_ℝ_¬_≤_less_thm : THM;
val z_ℝ_¬_less_≤_thm : THM;
val z_ℝ_0_less_0_less_times_thm : THM;

val z_ℝ_greater_thm : THM;
val z_ℝ_≥_thm : THM;
```

Description These are ML bindings for theorems that deal with the equality and ordering relations.

SML

```
val z_ℝ_eval_conv : CONV;          val Z_ℝ_EVAL_C : CONV -> CONV;
```

Description *z_ℝ_eval_conv* computes theorems of the form ⊢ *t1* = *t2* where *t1* is an expression made up from rational literals (see *z_ℝ_plus_conv*) using real addition, subtraction, multiplication, division, reciprocal, absolute value and unary negation. *t2* will be an optionally signed rational literal in normal form. The conversion fails if the expression cannot be evaluated (e.g., because it contains variables).

z_ℝ_EVAL_C conv is similar to ℝ_*eval_conv* but it also applies *conv* to any subterm that cannot be evaluated using the conversions for the arithmetic operations listed above. E.g., *z_ℝ_EVAL_C z_ℝ_ℤ_exp_conv* will evaluate expressions involving the usual arithmetic operations and also exponentiation of rational literals by natural number literals.

Errors

```
117020 ?0 cannot be evaluated
```

SML

```
val z_ℝ_lin_arith_prove_conv : THM list -> CONV;
val z_ℝ_lin_arith_prove_tac : THM list -> TACTIC;
```

Description This conversion and tactic implement the linear arithmetic decision procedure for real numbers. The usual interface to these is via the proof context *z_reals*, q.v.

SML

```
val z_ℝ_minus_clauses : THM;
val z_ℝ_minus_eq_thm : THM;
val z_ℝ_minus_thm : THM;
val z_ℝ_plus_0_thm : THM;
val z_ℝ_plus_assoc_thm : THM;
val z_ℝ_plus_assoc_thm1 : THM;
val z_ℝ_plus_clauses : THM;
val z_ℝ_plus_comm_thm : THM;
val z_ℝ_plus_minus_thm : THM;
val z_ℝ_plus_mono_thm : THM;
val z_ℝ_plus_mono_thm1 : THM;
val z_ℝ_plus_mono_thm2 : THM;
val z_ℝ_plus_order_thm : THM;
val z_ℝ_plus_thm : THM;
val z_ℝ_plus_unit_thm : THM;
val z_ℝ_subtract_thm : THM;
```

Description ML bindings for theorems about addition, unary minus and subtraction for the real numbers.

SML

```
val z_ℝ_real_ℕℝ_thm : THM;
val z_ℝ_real_0_thm : THM;
val z_float_thm : THM;
```

Description ML bindings for theorems concerning Z integer and floating point real literals.

SML

```
val z_ℝ_times_assoc_thm : THM;
val z_ℝ_times_assoc_thm1 : THM;
val z_ℝ_times_clauses : THM;
val z_ℝ_times_comm_thm : THM;
val z_ℝ_times_order_thm : THM;
val z_ℝ_times_plus_distrib_thm : THM;
val z_ℝ_times_thm : THM;
val z_ℝ_times_unit_thm : THM;
val z_ℝ_over_thm : THM;
val z_ℝ_over_clauses : THM;
```

Description ML bindings for theorems about multiplication and division of real numbers.

SML

```
val z_ℝ_≤_conv : CONV;           (* _ ≤_R _ *)
val z_ℝ_eq_conv : CONV;          (* _ = _ *)
val z_ℝ_less_conv : CONV;        (* _ <_R _ *)
val z_ℝ_minus_conv : CONV;             (* ~_R *)
val z_ℝ_over_conv : CONV;              (* _ /_R _ *)
val z_ℝ_plus_conv : CONV;              (* _ +_R _ *)
val z_ℝ_times_conv : CONV;             (* _ *_R _ *)
val z_ℝ_ℤ_exp_conv : CONV;             (* _ ^_Z _ *)
val z_ℝ_abs_conv : CONV;         (* abs_Z _ *)

val z_ℝ_greater_conv : CONV;           (* _ >_R _ *)
val z_ℝ_≥_conv : CONV;           (* _ ≥_R _ *)
val z_ℝ_subtract_conv : CONV;    (* _ −_R _ *)

val z_ℝ_lit_norm_conv : CONV;

val z_ℝ_lit_conv : CONV;         val z_ℝ_lit_conv1 : CONV;
```

Description These are conversions for carrying out real arithmetic computation. The first and second blocks of conversions deal with expressions of the form *c op d*, where *c* and *d* are real literal expressions (see below) and where *op* is the operator given in the ML comment alongside the conversion above. The conversions in the first block actually carry out the computation to give a theorem $c\ op\ d = e$ or $c\ op\ d \Leftrightarrow v$ where *e* and *v* are a real literal expression or a truth value as appropriate.

The conversions in the second block rewrite their argument in terms of the operators supported by the conversions in the first block.

The conversion *z_ℝ_lit_norm_conv* normalises real literal expressions, i.e., expressions of either of the forms *real i* or $i\ /_Z\ j$, where *i* and *j* are optionally signed integer literals. The conversion puts the result in a normal form, where the sign if any is moved to the outside, where *real* is used whenever possible and where if the form $i\ /_Z\ j$ has to be used, *i* and it j are taken to be coprime. This conversion fails if its argument cannot be normalised or is already in the normal form.

The final two conversions *z_ℝ_lit_conv* and *z_ℝ_lit_conv1* convert to and from Z and HOL real literal expressions.

Errors

117001 ?0 is not a Z real fraction with integer literal operands
117002 ?0 is not an HOL real fraction with literal operands
117003 ?0 is not of the form ?1 where x and y are real literal expressions
117004 ?0 is not of the form ?1 where x is a real literal expression
117005 ?0 is not of the form x ^_Z i where x is a real literal expression and i is an integer literal

SML

```
val z_ℝ_ℤ_exp_def : THM;
val z_ℝ_≥_def : THM;
val z_ℝ_≤_def : THM;
val z_ℝ_abs_def : THM;
val z_ℝ_def : THM;
val z_ℝ_dot_dot_def : THM;
val z_ℝ_frac_def : THM;
val z_ℝ_greater_def : THM;
val z_ℝ_less_def : THM;
val z_ℝ_minus_def : THM;
val z_ℝ_over_def : THM;
val z_ℝ_plus_def : THM;
val z_ℝ_real_def : THM;
val z_ℝ_subtract_def : THM;
val z_ℝ_times_def : THM;
val z_ℝ_lb_def : THM;
val z_ℝ_glb_def : THM;
val z_ℝ_ub_def : THM;
val z_ℝ_lub_def : THM;
```

Description ML bindings for the definitions in the theory of real numbers.

REFERENCES

[1] Michael J.C. Gordon and Tom F. Melham, editors. *Introduction to HOL.* Cambridge University Press, 1993.

[2] Michael J.C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF. Lecture Notes in Computer Science. Vol. 78.* Springer-Verlag, 1979.

[3] DS/FMU/IED/USR001. *ProofPower Document Preparation.* Lemma 1 Ltd.

[4] DS/FMU/IED/USR004. *ProofPower Tutorial Manual.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[5] DS/FMU/IED/USR007. *ProofPower Installation and Operation.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[6] DS/FMU/IED/USR011. *ProofPower Z Tutorial.* Lemma 1 Ltd., `http://www.lemma-one.com`.

INDEX

KEYWORD INDEX