The HOL record_proof **Library**

W. Wong

University of Cambridge, Computer Laboratory
New Museums Site, Pembroke Street
Cambridge, CB2 3QG, England.

July 1993

Contents

Chapter 1

The record_proof Library

This library provides a user interface to the proof recording feature of HOL. This feature allows the user to record and output a proof to a file. This proof can then be checked by an independent system, such as a proof checker.

1.1 Proofs in HOL

A *proof* is a finite sequence of *inferences* in a deductive system. Each inference is a pair $(L, (\Gamma, t))$ where (Γ, t) is known as a *sequent* and L is a (possibly empty) list of sequents $(\Gamma_1, t_1) \ldots (\Gamma_n, t_n)$. In practice, a particular deductive system is usually specified by a number of schematic *rules of inference* written in the form

$$\frac{\Gamma_1 \vdash t_1 \quad \ldots \quad \Gamma_n \vdash t_n}{\Gamma \vdash t} \tag{1.1}$$

The sequents above the line are called the *hypotheses* of the rule and the sequent below the line is called its *conclusion*. Each inference step in the sequence of inferences forming a proof must be satisfied by one of the inference rules of the deductive system. There are eight primitive rules of inference in HOL. They are described in detail in Section 10.3.1 of *DESCRIPTION*.

In HOL, rules of inference are represented by ML functions. More complex inference can be derived by combining the primitive inference rules. For example, the rule of *symmetry of equality*([`SYM`]) can be specified as

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}. \tag{1.2}$$

This can be derived using the primitive rules as follows:

1.	$\Gamma \vdash t_1 = t_2$	[Hypothesis]
2.	$\vdash t_1 = t_1$	[Reflexivity]
3.	$\Gamma \vdash t_2 = t_1$	[Substitution of 1 into 2]

This style of presenting a proof is known as *Hilbert* style. Each line is a single step in the sequence of inferences. The first column is the line number. The middle column is the resulting theorem(s) of the inference. The right-hand column is known as the *justification* which tells which rule of inference is applied in each step.

Derived rules are also represented by ML functions. They are implemented in terms of the primitive rules. A theorem prover in which all proofs are fully expanded into primitive inferences is known as *fully-expansive*[1]. The advantage of this type of theorem provers is the consistence of the proof is guaranteed since every primitive inference step is actually performed. However, this is very expensive in terms of both time and space for any sizable proof. To improve the efficiency of HOL, some of the simple derived rules, such as `SYM`, are not fully expanded, but are implemented directly in ML.

These rules, including the primitive rules and derived rules that are implemented directly in ML, will be referred to as *basic inference rules* or simply *basic rules* hereinbelow. When recording a proof, all inference steps in which a basic inference rule is applied should be included so that any error resulting from bugs in the implementation of the inference rules can be caught.

Simple proofs can be carried out in HOL by calling the inference rules in sequence. However, these inference steps are too small for any sizable proof. Another more powerful way of carrying out proof, known as *goal-directed* or *tactical* proof is often used. In this proof style, a term in the same form as the required theorem is set up as a goal, tactics are used to reduce the goal to simpler subgoals recursively until all the subgoal are resolved. In such a proof, the user does not call the inference rules directly. However, a correct sequence of inferences is calculated and performed by the system behind the scene automatically to derive the theorem. These proofs can be checked using these sequence of inferences.

A proof in HOL as described above is carried out within an environment. It consists of a type structure Ω and a signature under the type structure Σ_Ω. The type structure Ω is a set of type constants, each of which is a pair (ν, n) where ν is the name and n is known as the arity. Type constants include both the atomic types and the type operators. For example, the name of the atomic type $:bool$ is the string `bool` and its arity is 0, and the name of the type operator *list* is *list* and its arity is 1. The signature Σ_Ω is a set of constants, each of which is a pair $(\mathtt{c}, \sigma)$ where `c` is the name and σ is its type and all the type constants occur in the σs must be in Ω. This provides a context against which the well-typedness of terms can be checked.

1.2 The Proof file format

A recorded proof is saved in a disk file in a format known as `prf` format which is specified in [2]. Proof files in this format are primarily for automatic checker. It follows the Hilbert style of proofs as described in the previous section. It is a linear model which simplifies both the generation and the checking of proofs.

The proof file format `prf` is in two levels: the *core* level allows proofs using only primitive inference rules to be written into the file, and the *extended* levels allows all basic inference rules.

A checker accepting the core level file will be very simple, so it may be verified formally. Another program can be developed to expand the inference step involving derived rules into a sequence of primitive steps before being send to the core checker.

The syntax of the proof file format `prf` is similar to LISP S-expression. Objects, such as lines, theorems, terms and so on, are enclosed in a pair of matching parentheses. The first atom in an object is a tag specifying what kind of object it is. A file in this format begins with a format expression which identifies the name, version and level of the format it conforms to. This is followed by an environment expression. The remainder of the file consists of one or more proofs. Each proof expression begins with the `PROOF` tag identifying the expression as a proof. This is followed by the name of the proof and a list of theorems. These theorems are the goals of the proof. A checker checking the proof may stop processing the remaining proof lines after it has found all the theorems in the theorem fields of the proof lines. The last part of a proof expression is a sequence of line lines.

1.3 The User Interface

To a user, recording proof is a feature which can be enabled or disabled. Whatever the state the system is in, it performs proofs in the same way exception that the extra step of storing the proofs in a file is carried out only if the feature is enabled. The typical use of this feature will be that

1. the user carries out a proof in the usual manner;

2. when he/she is satisfied with the proof, the proof recording feature is enabled by loading the library `record_proof`, the proof is re-done once more and is saved in a proof file.

To help explain the user interface, a simple interactive session with HOLis shown below.

```
#load_library`record_proof`;;                                        1
Loading library record_proof ...
Updating search path
.......................................Updating help search path
.
Library record_proof loaded.
() : void

#new_theory`THY`;;
() : void

#new_proof_file`myproof.prf`;;
() : void

#prove_thm(`ADD_0`,
#  "!m. m + 0 = m",
#   INDUCT_TAC THEN ASM_REWRITE_TAC[ADD]);;
|- !m. m + 0 = m

#close_proof_file();;
() : void

#close_theory();;
() : void
```

After loading the library, a new theory named `THY` is created first for saving the theorem. A new proof file with the name `myproof.prf` is opened using the library function `new_proof_file`. This file becomes the current proof file. The name of the current proof file can be returned by calling the ML function `current_proof_file`. A theorem with the name `ADD_0` is proved and saved in the current theory using the system function `prove_thm`, and the proof is saved automatically in the current proof file. In fact, every call to the tactical proof functions `TAC_PROOF`, `PROVE`, `prove` and `prove_thm` will save a proof into the current proof file. When all the proofs are completed, the library function `close_proof_file` is called to close the proof file. Since the proof file is usually very large, it will be automatically compressed. The name of the compressed file is obtained by adding the suffix `.gz` to the name string passed to t`new_proof_file`.

To record forward proofs, one needs to use the library functions `begin_proof` and `end_proof` to enclose the proof procedures. Below is a simple example of forward proof:

```
#let th = SPEC_ALL ADD_SYM;;                                          2
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%536" : term

#begin_proof`proof1`;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#end_proof();;
() : void
```

Any inferences performed between the calls of `begin_proof` and `end_proof` will be saved in the current proof file. The string passed to the function `begin_proof` is the name of the proof. The function `current_proof` returns the name of the current proof if it is called within a forward proof. A null string returned indicates that no proof is in progress.

While one is developing the proof, one will not required the system to record and save the proof in a disk file. To disable the proof recording feature, the partial library `disable` is loaded instead of the whole library. This is done by the command:

```
load_library`record_proof:disable`;;
```

Usually, the input to the system is saved in a script file. It can then be loaded into the system to perform the proof in a batch processing fashion. By loading different part of the library as required, the same script file can be used to perform normal proofs and to generate proof files without any modification.

1.4 The Developer's Interface

This section describes a lower level interface to the proof recorder which consists of a small set of ML functions. These lower level functions provide a finer control to the proof recorder. They are useful for developing alternative user interfaces.

The process of recording proof and generating proof files can be divided into three stages:

1. recording inference steps;

2. generating a proof;

3. outputting to a text file.

In Stage 1, once the proof recorder is enabled by calling an ML function, every application of basic inference rule is recorded in an internal buffer. Each inference is represented by a ML object of type `step`. The recording can be temporarily suspended and resumed later. The current state of the recorder and the internal buffer can be accessed by calling ML functions. The ML functions available to the developer for managing the proof recorder are:

- `record_proof :bool -> void`
  When called with `true`, this function clears the internal buffer for storing proof steps and enables the recording of proof. When called with `false`, it just disables the feature without clearing the internal buffer.

- `is_recording_proof :void -> bool`
  This function returns the current state of the proof recorder. If it is enabled, `true` is returned.

- `get_steps :void -> step list`
  This function returns the proof in the internal buffer as a list of inference steps since the last time the proof recorder is enabled.

- `suspend_recording :void -> void`
  Disable the recorder temporarily without clearing the internal buffer. It should be re-enable using `resume_recording`.

- `resume_recording :void -> void`
  Re-enable the proof recorder without clearing the internal buffer.

In the second stage, the raw records of the inference steps are processed to reduce the duplicated information. The result is a list of proof lines which is represented by the ML type `line`. The only Ml function available for this stage is `MakeProof` which has type `step list -> line list`. It converts a list of inference steps to a list of proof lines.

The last stage is to convert the list of proof lines into text in the `prf` format and to output to a disc file. The ML functions available to the developer in this stage are:

- `write_proof_to : string -> string -> thm list -> line list -> void`
  This function outputs a proof into a proof file. The first string argument is the name of the output file, and the second is the name of the proof. If a file of the given name exists, it will be overwritten. The theorem list contains the theorem this proof supposes to derive. It may be null. In such case, a checker reading the proof should check all the lines. The last argument is a list of proof lines.

- `write_proof_add_to : string -> string -> thm list -> line list -> void`
  This function appends a proof into a proof file. The first string argument is the name of the output file. If no file of the given name exists, it will be created. If the named file exists and the version expression in it is not the same as the version the system is using, this function fails. The remaining arguments are interpreted in the same way as `write_proof_to`.

- `write_env :string -> void`
  This function outputs the current proof environment to the output port given as the argument.

- `write_line :string -> line -> void`
  This function converts a proof line from its internal representation to a text string and outputs it to the port given as the first argument.

- `write_thm_list :string -> thm list -> void`
  This function converts a list of theorems from its internal representation in to text in the `prf` format and outputs it to the port given as the first argument.

The implementation details of all these functions and internal data structures are described in [2].

Chapter 2

ML Functions in the record_proof Library

This chapter provides documentation on all the ML functions that are made available in HOL when the `res_quan` library is loaded. This documentation is also available online via the `help` facility.

begin_proof

```
begin_proof : string -> void
```

Synopsis

Begin recording a proof.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1]. A proof file may contain one or more of proofs.

The function `begin_proof` starts the proof recorder. All inferences performed by the system will be recorded until the function `end_proof` is called. It stores the proof into the current proof file. The argument to `begin_proof` is the name of the proof which will be written into the proof file.

The `begin_proof` and `end_proof` pair provides a simple user interface for recording forward proofs. For tactical proofs, use the functions `TAC_PROOF`, `PROVE`, `prove` and `prove_thm`. They will save the proof in the current proof file automatically.

Failure

Fails if a proof starts by `begin_proof` has not been terminated by `end_proof`.

Example

Below is an example showing the use of this function to record a simple proof:

```
#new_proof_file `myproof`;;
() : void

#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%536" : term

#begin_proof`proof1`;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#end_proof();;
() : void
```

See also

current_proof, current_proof_file, new_proof_file, close_proof_file, begin_proof, end_proof, TAC_PROOF, PROVE, prove, prove_thm

close_proof_file

```
close_proof_file : void -> void
```

Synopsis

Close the current proof file.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1]. A proof file may contain one or more of proofs.

The function `close_proof_file` closes the current proof file which has been opened by calling the function `new_proof_file`. Since the size of the proof file is very large, it will be automatically compressed. The name of the compressed file obtained by appending the suffix `.gz` to the original name.

Failure

Fails if not proof file is currently opened.

Example

Below is an example showing the use of this function to record a simple proof:

```
#new_theory`THY`;;
() : void

#new_proof_file`myproof`;;
() : void

#prove_thm(`ADD_0`,
#  "!m. m + 0 = m",
#   INDUCT_TAC THEN ASM_REWRITE_TAC[ADD]);;
|- !m. m + 0 = m

#close_proof_file ();;
() : void
```

See also

`current_proof`, `current_proof_file`, `new_proof_file`, `begin_proof`, `end_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`

current_proof

```
current_proof : void -> string
```

Synopsis

Get the name of the current proof.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1]. A proof file may contain one or more of proofs.

The function `current_proof` returns the name of the current proof. A null string indicates no proof is currently in progress.

Failure

Never fail.

Example

Below is an example showing the use of this function:

```
#new_proof_file `myproof`;;
() : void

#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%536" : term

#begin_proof`proof1`;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#current_proof ();;
`proof1` : string

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#end_proof();;
() : void

#current_proof();;
() : void
```

See also

current_proof_file, new_proof_file, close_proof_file, begin_proof, end_proof, TAC_PROOF, PROVE, prove, prove_thm

current_proof_file

```
current_proof_file : void -> string
```

Synopsis

Get the name of the current proof file.

Description

The function `current_proof_file` returns the name of the current proof file which has been opened by calling the function `new_proof_file`. A null string `‘‘` indicates that no current proof file is opened.

Failure

Never fail.

Example

Below is an example showing the use of this function:

```
#new_theory‘THY‘;;
() : void

#new_proof_file‘myproof‘;;
() : void

#prove_thm(‘ADD_0‘,
#  "!m. m + 0 = m",
#   INDUCT_TAC THEN ASM_REWRITE_TAC[ADD]);;
|- !m. m + 0 = m

#current_proof_file();;
‘myproof‘ : string

#close_proof_file ();;
() : void

#current_proof_file();;
‘‘ : string
```

See also

`current_proof`, `close_proof_file`, `new_proof_file`, `begin_proof`, `end_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`

end_proof

```
end_proof : void -> void
```

Synopsis

Terminate recording and store a proof.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1]. A proof file may contain one or more of proofs.

The function `end_proof` stops the proof recorder and saves the proof into the current proof file. A proof must have been started by the function `begin_proof`.

The `begin_proof` and `end_proof` pair provides a simple user interface for recording forward proofs. For tactical proofs, use the functions `TAC_PROOF`, `PROVE`, `prove` and `prove_thm`. They will save the proof in the current proof file automatically.

Failure

Fails if no proof has been started by `begin_proof`.

Example

Below is an example showing the use of this function to record a simple proof:

```
#new_proof_file `myproof`;;
() : void

#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%536" : term

#begin_proof`proof1`;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#end_proof();;
() : void
```

See also

`current_proof`, `current_proof_file`, `new_proof_file`, `close_proof_file`, `begin_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`

get_steps

```
get_steps : void -> step list
```

Synopsis

Get the proof steps since last time the proof recorder is enabled.

Description

A proof is a list of inference steps. After the proof recorder is enabled, every inference performed by the system is recorded and cumulated in an internal buffer. When a proof is completed, the raw records can then be processed and output to a disk file.

`get_steps` is a low level user function for managing the proof recorder. It returns the list of steps cumulated since last time the proof recorder is enabled, i.e. the function `record_proof` is called with the value `true`. The list of inference step needs to be processed to produce a proof line list which can be output to a file.

Failure

Never fail.

Example

Below is an example showing how to record an extremely simple proof in which the function `get_steps` is called to get the list of inference steps:

```
#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%536" : term

#record_proof true;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#record_proof false;;
() : void

#write_proof_to `ap_term.prf` `ap_term` [] (get_steps());;
() : void
```

The proof consists of two inference steps: the application of the two primitive inference rules `REFL` and `SUBST`. The function `write_proof_to` outputs the proof into a file names `ap_term.prf`.

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than the low level functions such as `get_steps` directly.

See also

`record_proof`, `is_recording_proof`, `RecordStep`, `suspend_recording`, `resume_recording`, `MakeProof`, `current_proof`, `current_proof_file`, `new_proof_file`, `close_proof_file`, `begin_proof`, `end_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`.

is_recording_proof

```
is_recording_proof : void -> bool
```

Synopsis
Get the current state of proof recorder.

Description
A proof is a list of inference steps. After the proof recorder is enabled, every inference performed by the system is recorded and cumulated in an internal buffer. When a proof is completed, the raw records can then be processed and output to a disk file.

`is_recording_proof` is a low level user function for managing the proof recorder. It returns `true` is the proof recording is currently enabled. Otherwise, it returns `false`.

Failure
Never fail.

Comments
This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than the low level functions such as `is_recording_proof` directly.

See also
`record_proof, RecordStep, get_steps, suspend_recording, resume_recording, current_proof, current_proof_file, new_proof_file, close_proof_file, begin_proof, end_proof, TAC_PROOF, PROVE, prove, prove_thm.`

MakeProof

```
MakeProof : step list -> line list
```

Synopsis
Convert a inference step list to a proof line list.

Description
A proof is a list of inference steps. After the proof recorder is enabled, every inference performed by the system is recorded and cumulated in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file.

`MakeProof` is the conversion function for transforming the step list into proof line list. The ML type `step` represents the inference steps. There are 52 constructors for the type `step`. Each constructor associates with a different kind of inferences. Each step contains the actual arguments of the call of the inference rule. As shown in the example below, the first step in the step list returned by `get_steps` is a substitution, i.e. the rule `SUBST`. This particular rule takes a list of theorem-term pairs, a term and a theorem as its argument.

The ML type `line` represents a proof line as in Hilbert's style proofs. Each line has a line number, a theorem which is the result of applying the inference rule, and a justification which indicates the inference rule to be applied together with its arguments. If an argument is a theorem, it is represented by a number. This number refers to the theorem in the line having the given line number.

Failure

Never fail.

Example

Below is an example showing how to record an extremely simple proof:

```
#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%385" : term

#record_proof true;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#get_steps();;
[SubstStep([(|- m + n = n + m, "GEN%VAR%385")],
           "SUC(m + n) = SUC GEN%VAR%385",
           |- SUC(m + n) = SUC(m + n));
 ReflStep"SUC(m + n)"]
: step list

#MakeProof it;;
[Line(-1, |- m + n = n + m, Hypothesis);
 Line(1, |- SUC(m + n) = SUC(m + n), Refl"SUC(m + n)");
 Line(2,
      |- SUC(m + n) = SUC(n + m),
      Subst([(-1, "GEN%VAR%385")], "SUC(m + n) = SUC GEN%VAR%385", 1))]
: line list
```

The proof consists of two inference steps: the application of the two primitive inference rules `REFL` and `SUBST`. The step list is in the reverse order of the inferences, i.e. the first element is the last inference. All arguments to the inference rule are presented in the steps explicitly. The line list generated by the function `MakeProof` is in the order of the proof. The last line, Line 2, corresponds to the substitution step. The theorems are replaced by line numbers. They indicate the line where the theorems are found.

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than this low level function directly.

See also

`get_steps`, `write_line`, `write_thm_list`, `write_env`, `write_proof_add_to`, `write_proof_to`, `write_last_proof`, `current_proof`, `current_proof_file`, `new_proof_file`, `close_proof_file`, `begin_proof`, `end_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`

new_proof_file

```
new_proof_file : string -> void
```

Synopsis

Create a new proof file.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1]. A proof file may contain one or more of proofs.

The function `new_proof_file` creates a new proof file with the name specified as its argument. This file becomes the current proof file. All subsequent calls of tactical proof functions `TAC_PROOF`, `PROVE`, `prove` and `prove_thm` will save the proof in this file. If forward inferences are required in the middle of a tactical proof, they should be make local to the above proof functions. Otherwise they will not be recorded.

If more extensive forward proof is performed, the functions `begin_proof` and `end_proof` can be used to record and store the proof into the current proof file.

If a file of the same name as the argument already exists, it will be overwritten. The current proof environment will be written to the file before the proofs.

Failure

Never fail.

Example

Below is an example showing the use of this function to record a simple proof:

```
#new_theory`THY`;;
() : void

#new_proof_file`myproof`;;
() : void

#prove_thm(`ADD_0`,
#  "!m. m + 0 = m",
#   INDUCT_TAC THEN ASM_REWRITE_TAC[ADD]);;
|- !m. m + 0 = m

#close_proof_file();;
() : void
```

See also

current_proof, current_proof_file, close_proof_file, begin_proof, end_proof, TAC_PROOF, PROVE, prove, prove_thm

PROVE

```
PROVE : ((term # tactic) -> thm)
```

Synopsis

Attempts to prove a boolean term using the supplied tactic and saves the proof.

Description

When applied to a term-tactic pair `(tm,tac)`, the function `PROVE` attempts to prove the goal `?- tm`, that is, the term `tm` with no assumptions, using the tactic `tac`. If `PROVE` succeeds, it saves the proof in the current proof file and returns the corresponding theorem `A |- tm`, where the assumption list `A` may not be empty if the tactic is invalid; `PROVE` has no inbuilt validity-checking.

Failure

Fails if the term is not of type `bool` (and so cannot possibly be the conclusion of a theorem), or if the tactic cannot solve the goal. In the latter case `PROVE` will list the unsolved goals to help the user.

See also

`current_proof`, `current_proof_file`, `begin_proof`, `end_proof`, `new_proof_file`, `close_proof_file`, `TAC_PROOF`, `prove`, `prove_thm`.

prove

```
prove : ((term # tactic) -> thm)
```

Synopsis

Attempts to prove a boolean term using the supplied tactic and saves the proof.

Description

When applied to a term-tactic pair `(tm,tac)`, the function `prove` attempts to prove the goal `?- tm`, that is, the term `tm` with no assumptions, using the tactic `tac`. If `prove` succeeds, it saves the proof in the current proof file and returns the corresponding theorem `A |- tm`, where the assumption list `A` may not be empty if the tactic is invalid; `prove` has no inbuilt validity-checking.

Failure

Fails if the term is not of type `bool` (and so cannot possibly be the conclusion of a theorem), or if the tactic cannot solve the goal.

Comments

The function `PROVE` provides almost identical functionality, and will also list unsolved goals if the tactic fails. It is therefore preferable for most purposes.

See also

`current_proof`, `current_proof_file`, `begin_proof`, `end_proof`, `new_proof_file`, `close_proof_file`, `PROVE`, `TAC_PROOF`, `prove_thm`.

prove_thm

```
prove_thm : ((string # term # tactic) -> thm)
```

Synopsis

Attempts to prove a boolean term using the supplied tactic, then save the proof and the theorem.

Description

When applied to a triple (s,tm,tac), giving the name to save the theorem under, the term to prove (with no assumptions) and the tactic to perform the proof, the function prove_thm attempts to prove the goal ?- tm, that is, the term tm with no assumptions, using the tactic tac. If prove_thm succeeds, it stores the proof in the current proof file, and then attempts to save the resulting theorem in the current theory segment, and if this succeeds, the saved theorem is returned.

Failure

Fails if the term is not of type bool (and so cannot possibly be the conclusion of a theorem), or if the tactic cannot solve the goal. In the latter case prove_thm will list the unsolved goals to help the user. In addition, prove_thm will fail if the theorem cannot be saved, e.g. because there is already a theorem of that name in the current theory segment, or if the resulting theorem has assumptions; clearly this can only happen if the tactic was invalid, so this gives some measure of validity checking.

See also

current_proof, current_proof_file, begin_proof, end_proof, new_proof_file, close_proof_file, PROVE, TAC_PROOF, prove_thm, VALID.

RecordStep

```
RecordStep : step -> void
```

Synopsis

Record a single inference step.

Description

A proof is a list of inference steps. After the proof recorder is enabled, every inference performed by the system is recorded and cumulated in an internal buffer. When a proof is completed, the raw records can then be processed and output to a disk file.

record_proof is a system function for recording a single proof step. The type step represents a basic inference step. It contains all the necessary information of each inference. There are currently 52 basic inferences which are being recorded. All other ML functions representing inference rules are implemented by these basic inferences. If the proof recorder is enabled when an inference is performed, RecordStep will add a step into the internal buffer.

Failure

Never fail.

Comments

This is a system function implementing the proof recorded. Users should not use this function directly. User functions are provided in the library `record_proof`. When new basic inference rule is implemented, this function should be called to record the inference step.

See also

`record_proof`, `is_recording_proof`, `get_steps`, `suspend_recording`, `resume_recording`.

record_proof

```
record_proof : bool -> void
```

Synopsis

Enable/disable proof recording.

Description

A proof is a list of inference steps. After the proof recorder is enabled, every inference performed by the system is recorded and cumulated in an internal buffer. When a proof is completed, the raw records can then be processed and output to a disk file.

`record_proof` is a low level user function for managing the proof recorder. It takes a single boolean as its argument. If this is `true`, it initialises and enables the proof recorder. If it is `false`, it disables the proof recorder without clearing the proof recorded since the last time this function is called with the value `true`.

The current state of the proof recorder can interrogated using the function `is_recording_proof`. The recorder can be temporarily disabled and later re-enabled without clearing the internal buffer.

Failure

Never fail.

Example

Below is an example showing how to record an extremely simple proof:

```
#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%536" : term

#record_proof true;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#record_proof false;;
() : void

#write_proof_to `ap_term.prf` `ap_term` [] (get_steps());;
() : void
```

The proof consists of two inference steps: the application of the two primitive inference rules REFL and SUBST. The function write_proof_to outputs the proof into a file names ap_term.prf.

Comments

This function is used to implement higher level user functions for recording proof in the library record_proof. It is much more convenient to use those functions than the low level functions such as record_proof directly.

See also

is_recording_proof, RecordStep, get_steps, suspend_recording, resume_recording, current_proof, current_proof_file, new_proof_file, close_proof_file, begin_proof, end_proof, TAC_PROOF, PROVE, prove, prove_thm.

resume_recording

`resume_recording : void -> void`

Synopsis

Suspend proof recording temporarily.

Description

A proof is a list of inference steps. After the proof recorder is enabled, every inference performed by the system is recorded and cumulated in an internal buffer. When a proof is completed, the raw records can then be processed and output to a disk file.

`resume_recording` is a low level user function for managing the proof recorder. It resumes the proof recorder without clearing the internal inference step buffer. The proof recorder should be in the suspended state. Otherwise, this function does nothing.

The current state of the proof recorder can interrogated using the function `is_recording_proof`. A value of `false` indicates the proof recorder is disabled.

Failure

Never fail.

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than the low level functions such as `resume_recording` directly.

See also

`record_proof`, `is_recording_proof`, `RecordStep`, `get_steps`, `suspend_recording`, `current_proof`, `current_proof_file`, `new_proof_file`, `close_proof_file`, `begin_proof`, `end_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`.

suspend_recording

`suspend_recording : void -> void`

Synopsis

Suspend proof recording temporarily.

Description

A proof is a list of inference steps. After the proof recorder is enabled, every inference performed by the system is recorded and cumulated in an internal buffer. When a proof is completed, the raw records can then be processed and output to a disk file.

`suspend_recording` is a low level user function for managing the proof recorder. It pauses the proof recorder until the function `resume_recording` is called. When the proof recorder is suspended, any inference carried out by the system will not be recorded. The internal buffer for storing the inference steps will not be cleared. When the proof recorder is resumed, the steps will be appended to what have been saved in the buffer.

The current state of the proof recorder can interrogated using the function `is_recording_proof`. A value of `false` indicates the proof recorder is disabled.

Failure

Never fail.

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than the low level functions such as `suspend_recording` directly.

See also

`record_proof, is_recording_proof, RecordStep, get_steps, resume_recording, current_proof, current_proof_file, new_proof_file, close_proof_file, begin_proof, end_proof, TAC_PROOF, PROVE, prove, prove_thm`

TAC_PROOF

```
TAC_PROOF : ((goal # tactic) -> thm)
```

Synopsis

Attempts to prove a goal using a given tactic and stores the proof in a file.

Description

When applied to a goal-tactic pair `(A ?- t,tac)`, the `TAC_PROOF` function attempts to prove the goal `A ?- t`, using the tactic `tac`. If it succeeds, it saves the proof in the current proof file and returns the theorem `A' |- t` corresponding to the goal, where the assumption list `A'` may be a proper superset of `A` unless the tactic is valid; there is no inbuilt validity checking.

Failure

Fails unless the goal has hypotheses and conclusions all of type `bool`, and the tactic can solve the goal.

See also

`current_proof`, `current_proof_file`, `begin_proof`, `end_proof`, `new_proof_file`, `close_proof_file`, `PROVE`, `prove`, `prove_thm`.

write_env

```
write_env : string -> void
```

Synopsis

Output the current proof environment to a port.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1]. A proof is performed in an environment. It consists of a set of type operators and a set of constants.

`write_env` is the output function for writing the current proof environment to a output port. The first argument is a string representing a port. It should be the value returned by the system function `openw`.

Failure

Never fail.

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than this low level function directly.

See also

`write_thm_list`, `write_line`, `write_proof_add_to`, `write_proof_to`, `write_last_proof`, `current_proof`, `current_proof_file`, `new_proof_file`, `close_proof_file`, `begin_proof`, `end_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`

write_line

```
write_line : string -> line -> void
```

Synopsis

Output a proof line to a port.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1].

`write_line` is the output function for writing a proof line to a output port. The first argument is a string representing a port. It should be the value returned by the system function `openw`.

Failure

Never fail.

Example

Below is an example showing the use of this function in writing the proof lines of an extremely simple proof:

```
#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%385" : term

#record_proof true;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#get_steps();;
[SubstStep([(|- m + n = n + m, "GEN%VAR%385")],
           "SUC(m + n) = SUC GEN%VAR%385",
           |- SUC(m + n) = SUC(m + n));
 ReflStep"SUC(m + n)"]
: step list

#MakeProof it;;
[Line(-1, |- m + n = n + m, Hypothesis);
 Line(1, |- SUC(m + n) = SUC(m + n), Refl"SUC(m + n)");
 Line(2,
      |- SUC(m + n) = SUC(n + m),
      Subst([(-1, "GEN%VAR%385")], "SUC(m + n) = SUC GEN%VAR%385", 1))]
: line list

#map (write_line port) it;;
() : void
```

`port` is assumed has been connected to a file that is opened for writing.

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than this low level function directly.

See also

`write_thm_list`, `write_env`, `write_proof_add_to`, `write_proof_to`, `write_last_proof`, `current_proof`, `current_proof_file`, `new_proof_file`, `close_proof_file`, `begin_proof`, `end_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`

write_proof_add_to

```
write_proof_add_to : string -> string -> thm list -> line list -> void
```

Synopsis

Output a proof to an existing proof file.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1].

`write_proof_add_to` is the output function for writing a proof to an existing proof file. The first argument is the name of the output file. The second is the name of the proof. The third is a list of theorems that are to be derived in the proof. The fourth is the list of proof lines.

If a file having the name as the first argument does not exist, it will be created. In this case, `write_proof_add_to` is identical to `write_proof_to`. If a file of the same name as the first argument already exists, its format should be compatible to the current format. The proof is appended to the file.

Failure

Fails if the named file exists and not in a compatible format.

Example

Below is an example showing the use of this function in writing the proof lines of an extremely simple proof:

```
#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%385" : term

#record_proof true;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#record_proof false;;
() : void

#write_proof_add_to `ap_term.prf` `ap_term` [] (MakeProof(get_steps()));;
() : void
```

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than this function directly.

See also

`write_line`, `write_thm_list`, `write_env`, `write_proof_to`, `write_last_proof`, `current_proof`, `current_proof_file`, `new_proof_file`, `close_proof_file`, `begin_proof`, `end_proof`, `TAC_PROOF`, `PROVE`, `prove`, `prove_thm`

write_proof_to

```
write_proof_to : string -> string -> thm list -> line list -> void
```

Synopsis

Output a proof to a proof file.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1].

`write_proof_to` is the output function for writing a proof to a file. The first argument is the name of the output file. The second is the name of the proof. The third is a list of theorems that are to be derived in the proof. The fourth is the list of proof lines.

If a file of the same name as the first argument already exists, it will be overwritten. The current proof environment will be written to the file before the proof.

Failure

Never fail.

Example

Below is an example showing the use of this function in writing the proof lines of an extremely simple proof:

```
#let th = SPEC_ALL ADD_SYM;;
Theorem ADD_SYM autoloading from theory `arithmetic` ...
ADD_SYM = |- !m n. m + n = n + m

th = |- m + n = n + m

#let v = genvar ":num";;
"GEN%VAR%385" : term

#record_proof true;;
() : void

#let th1 = (REFL "SUC(m + n)");;
th1 = |- SUC(m + n) = SUC(m + n)

#let th2 = SUBST [th,v] "SUC(m + n) = SUC ^v" th1;;
th2 = |- SUC(m + n) = SUC(n + m)

#record_proof false;;
() : void

#write_proof_to `ap_term.prf` `ap_term` [] (MakeProof(get_steps()));;
() : void
```

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than this function directly.

See also

`write_line, write_thm_list, write_env, write_proof_add_to, write_last_proof, current_proof, current_proof_file, new_proof_file, close_proof_file, begin_proof, end_proof, TAC_PROOF, PROVE, prove, prove_thm`

write_thm_list

```
write_thm_list : string -> thm list -> void
```

Synopsis

Output a proof line to a port.

Description

A proof is a list of inference steps. The proof recorder records every inference performed by the system in an internal buffer. When a proof is completed, the raw records are converted into a list of proof lines as in Hilbert's style proofs. These lines are then output to a disk file. This file is in the `prf` format specified in [1].

`write_thm_list` is the output function for writing a list of theorems to a output port. The first argument is a string representing a port. It should be the value returned by the system function `openw`.

Failure

Never fail.

Comments

This function is used to implement higher level user functions for recording proof in the library `record_proof`. It is much more convenient to use those functions than this low level function directly.

See also

`write_thm_list, write_env, write_proof_add_to, write_proof_to, write_last_proof, current_proof, current_proof_file, new_proof_file, close_proof_file, begin_proof, end_proof, TAC_PROOF, PROVE, prove, prove_thm`

References

[1] R. J. Boulton. On efficiency in theorem provers which fully expand proofs into primitive inferences. Technical Report 248, University of Cambridge Computer Laboratory, 1992.

[2] W. Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, ENGLAND, July 1993.

Index