Recording HOL Proofs

Wai Wong

30 July 1993

Abstract

This paper describes a text file format for recording HOL proofs. It is intended to become an interface between HOL and proof checkers. Modification to HOL88 has been carried out to incorporate a proof recorder to generate a proof file in this format. The usage of this new feature is explained by a simple example. A more substantial proof has been recorded, and benchmark data is presented here.

Contents

1 Motivation

Formal methods have been used in the development of many safety-critical systems in the form of formal specification and formal proof of correctness. Formal proofs are usually carried out by *theorem provers* or *proof assistants*. These systems are based on well-founded formal logic, and provide a programming environment for the user to discover, construct and perform a proof. The result of this process is usually a theorem which can be stored and used in subsequent proofs. HOL is one of the most popular theorem proving environments. The user interacts with the system by writing and evaluating ML programs which instruct the system to carry out proofs. A proof is a sequence of inferences. In the HOL system, it is transient in the sense that there is no object that exists as a proof once a theorem has been derived.

In some applications, it is desirable to check the sequence of inferences by an independent checker to assure its consistency[1]. The ML programs that a user develops while doing the proof are not adequate, and very often are too complicated for this purpose. The necessary condition for allowing the independent checking of a HOL proof is to record it in a form which is readable by the checker and consists of sufficient information to re-derive the theorem. Currently, research is being carried out to formalise the notion of higher-order logic proofs in HOL[5]. This will provide a theoretical foundation for a proof checker.

This report describes a proof file format `prf`, which acts as an interface between HOL and a proof checker, and an enhancement to HOL88 to record proofs and to generate proof files in this format. Section 2 reviews briefly the notion of a proof in HOL and the ways a proof is carried out. Section 3 gives a precise definition of the proof file format and an explanation of its semantics. The interface for recording and generating proof is explained in Section 5 and 6. Section 7 describes the modification to HOL88 for recording proofs and generating proof files. Section 8 shows a non-trivial example proof as a benchmark.

2 Proofs in HOL

A detailed description of the HOL logic and several tutorial examples of using the HOL system can be found in [3]. For the benefit of readers who are not familiar with HOL, an overview of the HOL deductive system and the theorem-proofing infrastructure is given in this section.

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)

[1] "The highest degree of assurance in the design can be obtained by providing all supporting proofs and checking them with a *proof checker*. A proof checker can be a relatively simple program and thus can itself be verified by Formal Proof", quoted from Part 2, 32.1.3 of [4].

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]

Figure 1: Derivation of the derived rule `SYM`

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

$$\frac{\Gamma_1 \vdash t_1 \quad \ldots \quad \Gamma_n \vdash t_n}{\Gamma \vdash t} \tag{1}$$

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 4.1 on Page 15.

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

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

This can be derived using the primitive rules as shown in Figure 1.

Derived rules are also represented by ML functions. Many of them are implemented in terms of the primitive rules, i.e., calling the ML functions representing 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 prover is that the soundness 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.

The primitive rules and the derived rules that are implemented directly in ML will be collectively referred to as *basic inference rules* or simply *basic rules* below. 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 automatically behind the scenes to derive the theorem. The proof can be checked using this 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 occurring in the σs must be in Ω. This provides a context against which the well-typedness of terms can be checked.

The style of presenting a proof in Figure 1 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. This Hilbert style of proof is used as the model of the proof file format described in the next section.

3 The Proof File Format: `prf`

The proof files written in the format described here are intended primarily for use by an automatic checker. Therefore, the following two considerations influence the design of the proof file format:

- the file should contain only ASCII text characters thus facilitating the transfer of the file between machines across networks, and ensuring that machines with different internal data representation will have less of a problem reading it;

- the format should be simple so that it can be parsed easily, and so that the parser is simple and it may be verified itself.

The `prf` proof file format 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* level allows all directly implemented inference rules.

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

The remainder of this section is divided into two parts giving the syntax and the semantics of the `prf` format, respectively. The syntax part defines the concrete syntax of files conforming to the `prf` format. The semantics part explains how the files are interpreted.

3.1 The Syntax

The syntax of the proof file format for HOL 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 may consist of one or more proofs. This section specifies the concrete syntax of the format, i.e., the sequence of characters which appears in the file. The meanings of the expressions will be explained in Section 3.2.

The syntax of the proof format is specified in an augmented BNF. The conventions used in this augmented BNF are:

- The terminal symbols are in upper case. The actual text strings of the terminals which appear in the proof file are given in Figure 2.
- The token `NUMBER` is a string of one or more digits (`0`...`9`) and optionally prefixed by a minus sign(`-`).
- The token `STRING` is a string of characters which denotes names of variables, constants, and so on. The characters allowed in different expressions will be detailed in Section 3.2 when the expressions are explained.
- The non-terminal symbols are in lower case.
- A non-terminal symbol whose name ends with the suffix '`_list`' stands for a list of similar elements. A list is enclosed by a pair of matching delimiters `LB` and `RB`. The default delimiters are the square brackets, e.g., '`[...]`'. A list may be empty, in which case, there will be nothing in between the square brackets. The production rule for `xxx` then specifies the syntax of each element of the list `xxx_list`. Since each element is enclosed in a matching pair of parentheses, no element separator is needed.

TERMINAL	STRING	TERMINAL	STRING
PROOF	PROOF	LINE	LINE
THM	THM	VERSION	VERSION
VAR	V	CONST	C
APP	A	ABS	L
TYVAR	v	TYOP	o
TYCONST	c	ENV	ENV
LP	(	RP	)
LC	{	RC	}
LB	[	RB	]

Figure 2: Input strings for terminals

- In the rules for `prim_justification` and `ext_justification`, the first terminal of each justification is its name. The input string for this is the same as the name but the case of the characters is ignored.
- The start symbol is `prf_file`.

The syntax of the HOL proof file format is as below:

```
prf_file ::= version environment proofs ;;

version ::= LP VERSION STRING RP ;;

environment ::= LP ENV STRING typeconst_list const_list RP ;;

typeconst ::= LC STRING NUMBER RC ;;

const ::= LC STRING type RC ;;

proofs ::= proof | proofs proof ;;

proof ::= LP PROOF STRING thm_list line_list RP ;;

line ::= LP LINE NUMBER LP justification RP thm RP ;;

thm ::= LP THM term_list term RP ;;

term ::= LP VAR STRING type RP
```

```
      |  LP CONST STRING type RP
      |  LP APP term term RP
      |  LP ABS term term RP ;;

type ::= LP TYVAR STRING RP
      |  LP TYCONST STRING RP
      |  LP TYOP STRING type_list RP ;;

justification ::= core_justification
      | ext_justification ;;

core_justification ::= HYPOTHESIS
     | ASSUME term
     | REFL term
     | SUBST number_term_list term NUMBER
     | BETACONV term
     | ABS term NUMBER
     | INSTTYPE type_type_list NUMBER
     | DISCH term NUMBER
     | MP NUMBER NUMBER
     | STOREDEFINITION STRING term
     | DEFINITION STRING STRING
     | DEFEXISTSRULE term
     | NEWAXIOM STRING term
     | AXIOM STRING STRING
     | THEOREM STRING STRING
     | NEWCONSTANT STRING type
     | NEWTYPE NUMBER STRING
     | NUMCONV term ;;

ext_justification ::= MKCOMB NUMBER NUMBER
     | MKABS NUMBER
     | ALPHA term term
     | ADDASSUM term NUMBER
     | SYM NUMBER
     | TRANS NUMBER NUMBER
     | IMPTRANS NUMBER NUMBER
     | APTERM term NUMBER
     | APTHM NUMBER term
     | EQMP NUMBER NUMBER
     | EQIMPRULER NUMBER
     | EQIMPRULEL NUMBER
```

```
     | SPEC term NUMBER
     | EQTINTRO NUMBER
     | GEN term NUMBER
     | ETACONV term
     | EXT NUMBER
     | EXISTS term term NUMBER
     | CHOOSE term NUMBER NUMBER
     | IMPANTISYMRULE NUMBER NUMBER
     | MKEXISTS NUMBER
     | SUBS NUMBER_list NUMBER
     | SUBSOCCS number_list_number_list NUMBER
     | SUBSTCONV number_term_list term term
     | CONJ NUMBER NUMBER
     | CONJUNCT1 NUMBER
     | CONJUNCT2 NUMBER
     | DISJ1 NUMBER term
     | DISJ2 term NUMBER
     | DISJCASES NUMBER NUMBER NUMBER
     | NOTINTRO NUMBER
     | NOTELIM NUMBER
     | CONTR term NUMBER
     | CCONTR term NUMBER
     | INST term_term_list NUMBER ;;

number_term ::= LC NUMBER term RC ;;

type_type ::= LC type type RC ;;

number_list_number  ::= LC NUMBER_list NUMBER RC ;;

term_term ::= LC term term RC ;;
```

Space between tokens is optional except where its absence will result in ambiguity. For example, a space is required between the tag `C` and the string `T` and between the tag `c` and the string `bool` in the constant expression below:

```
  (C T(c bool))
```

The space character(ASCII code 32), tab(9), carriage return(13) and line feed(10) are considered to be space. Consecutive spaces are treated as a single space.

3.2 The Semantics

The semantics of the proof file is described informally in this section in a bottom-up fashion starting with the rule for types.

3.2.1 Types

HOL types are represented in the proof file by type expressions. There are three kinds of types: type variables, type constants and type operators. A type variable is represented by a `TYVAR` expression. The string in this expression is the name of the variable. In HOL88, all type variable names begin with the character '`*`'. This should be checked when the proof is parsed. For example, the expression `(v **)` represents the HOL type $:**$.

An atomic type is represented by a `TYCONST` expression. The string in this expression is the name of the type. For example, the HOL built-in type $:bool$ is represented as `(c bool)`.

The `TYOP` expression represents HOL type operators. The string in this expression is the name of the type operator and the `type_list` is the list of arguments of the type operator. The names of the HOL built-in infix type operators $\rightarrow$, # and + are respectively `fun`, `prod` and `sum`. The examples below show some HOL types in the proof format:

`(o fun [(v *)(v *)])`	$:* \rightarrow *$
`(o prod [(c num)(c bool)])`	$:num\#bool$
`(o sum [(c num)(c bool)])`	$:num + bool$

The string denoting the name of a type operator must be alphanumeric as described in next subsection.

3.2.2 Terms

There are four kinds of terms: variables, constants, applications and abstractions. They are represented respectively by the four kinds of expressions: `VAR`, `CONST`, `APP` and `ABS`.

The string in a `VAR` expression is the name of the variable. In HOL88, the variable name can be either alphanumeric or symbolic. An alphanumeric name is a sequence of one or more characters consisting of letters, digits, underscores(`_`), primes(`'`) and percent signs(`%`) and begins with a letter. A symbolic name is a sequence of one or more characters which is an initial subsequence of one of the following character sequences:

```
**   ++   <--   <->   -->   ---   ><   >>
>=   <==   <=>   ===   ==>   \/   //   /\
!?   !!   !\   ?!   ?? ?\   :=   <>   <-   <<   ->   =>
```

The `type` field in a `VAR` expression specifies the type of the variable. Some examples of `VAR` expressions are:

`(V foo (v *))`	$foo : *$
`(V GEN%VAR%123 (c bool))`	$GEN\%VAR\%123 : bool$
`(V -- (v *))`	$-- : *$

The string in a `CONST` expression is the name of the constant. It has the same form as the name of the variables. In addition, a sequence of one or more digits is also allowed, and it denotes a numeric constant. The `type` field of a constant expression gives the type of the specific instance of the constant. For instance, the HOL constant CONS has the polymorphic type $: * \to (*)list \to (*)list$, but the type of the occurrence of CONS in the term $\mathsf{CONS}\,\mathsf{T}\,l$ is $: bool \to (bool)list \to (bool)list$ which is more specific. This is the type given in the `CONST` expression representing the above occurrence of CONS. Below are some examples of constant expressions:

`(C T (c bool))`	$\mathsf{T} : bool$
`(C 2 (c num))`	$2 : num$

The expression representing function application has two sub-terms: the first is the function and the second is its argument. No explicit type information is included since this can be deduced from the types of the sub-terms. For example, the term $\mathsf{SUC}\,2$ is represented by the expression

```
(A(C SUC(o fun[(c num)(c num)]))(C 2(c num)))
```

The abstraction expression represents λ-abstraction in HOL. The two sub-terms are the bound variable and the abstraction body, respectively. Like the function application expression, no explicit type information is included in an abstraction. For example, the expression

```
(L (V x(c num))
   (A (A (C = (o fun[(c num)(o fun[(c num)(c bool)])]))
         (V x(c num)))(C 0(c num))))
```

represents the abstraction $(\lambda x.x = 0)$.

3.2.3 Theorems

A HOL theorem consists of a list of hypotheses and a conclusion. This is represented by the `THM` expression. The `term_list` field is the list of hypotheses and the `term` field is the conclusion.

3.2.4 Lines

Each line in a proof represents a single inference. The `NUMBER` in a line acts as a label of the line. It can be either a positive or a negative integer, but not zero. The lines in a proof occur in ascending order of the line numbers. No two lines have the same line number. The `justification` field indicates which inference rule is used in this step. Many justifications refer to the resulting theorems of other lines using their line numbers. It is illegal to refer to a line whose number is larger than the current line. The `thm` field contains the result of applying the inference rule.

3.2.5 Justifications

The justification expression in a proof line represents the inference rule to be applied. The justifications are divided into two categories: the core level justifications which are those in the production rule `core_justification`, and extended level rules which are in the production rule `ext_justification`. The first group consists of the eight HOL primitive inference rules and miscellaneous functions for declaring new types, constants and for retrieving definitions and theorems from existing theories. The extended group contains other basic inference rules. These can only appear in a file conforming to the extended level.

Since the HOL deduction system is based entirely on the eight primitive rules and five axioms. Any valid theorem can be derived using only these primitive rules and axioms however tedious the process may be. By dividing the justifications into two levels, a simple checker accepting only the core level justifications can be developed. It may then be verified to ensure its correctness. A translator can be added as a front end of the core checker. It expands the derived rules into primitives.

In a `justification` expression, the first field is the name of the inference rule. The name is a sequence of letters, and their case is ignored. For example, `Subst` and `SUBST` indicate the same inference rule, namely substitution. The remaining fields are the arguments to the inference rule. If an argument is a theorem, it is not explicitly written, instead a number is given. This refers to the theorem in a line having the given number. Arguments of other types are written explicitly. For example, the inference rule `REFL` takes a term, it is explicitly written in the justification expression. The expression below represents the justification in Line 2 of the proof shown in Figure 4.

```
(Refl (A (C SUC(o fun[(c num)(c num)]))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))
(V m(c num)))(V n(c num)))))
```

Details of individual justifications are explained in the next section.

3.2.6 Proofs

A `PROOF` expression consists of

- a name which is an alphanumeric string;
- a list of theorems; and
- a sequence of lines.

The theorems in the `thm_list` field are the goals of the proof, i.e., the theorems it aims to derive. A proof checker may ignore all subsequent lines as soon as all theorems in this list have been found in the theorem list field of the proof lines. A null list may be interpreted as to check all the lines in the proof. The `line_list` field contains the list of proof lines. If a proof is checked and no error is found, the theorems in all lines are valid theorems.

3.2.7 Environment

The environment expression `ENV` specifies a context for the proof(s) in the proof file. It consists of three sub-expressions: the `STRING` is the name of the environment; the `typeconst_list` is a type structure whose elements are type constants as described in Section 3.2.8; and the `const_list` is a signature under the type structure. There are three built-in environments: `MIN`, `LOG` and `HOL`. The exact contents of these environments can be found in Appendix D. The following expression indicates that the built-in environment `HOL` is selected:

```
(ENV HOL [] [])
```

Anything appearing in the lists will be ignored.

3.2.8 Type Structure

A type constant is a pair whose first field is a string and whose second field is a number. The string is the name of the type constant. The number is its arity which must not be negative. For example, the expression representing the type constant *bool* and *list* are

```
{bool 0}
{list 1}
```

The type structure of the current environment contains all properly defined type constants. The well-formedness of a type appearing in a term is checked against this type structure when required. The type expression below is correct in syntax but is not well-formed under the built-in environment `HOL`:

```
(o list [(v *) (v **)])
```

This is because the type operator *list* has arity of 1. It is assumed that all type constants have unique name, i.e., the same name cannot appear more than once in the type structure of an environment.

3.2.9 Signature

A signature is a list of pairs. Each pair represents a constant. The first field is a string denoting the name of the constant. The second field is its generic type, i.e., if a constant is polymorphic, the type field is its most general type. For example, the constant representing equality(=) has the name `=` and the type $: \alpha \to \alpha \to bool$. The expression representing this constant is

```
{= (o fun [(v *) (o fun [(v *)(c bool)])])}
```

The well-typedness of an occurrence of a constant is checked against the signature of the current environment when required. It is well-typed if the particular occurrence is an instance of the generic constant in the signature.

3.2.10 Version

The string in the `VERSION` expression specifies the version and level of the format to which the file conforms. This string can be divided into three parts:

1. *format name* — the standard name of the format is `PRF FORMAT`;
2. *version* — the current version of the format is `1.0`;
3. *level* — the levels must be either `CORE` or `EXTENDED`.

Parts are separated by a single space. The proof checker reading a file may perform different action according to the version and level indicated in the file. These provides a means for backward compatibility in case the format changes in future.

3.2.11 Proof File

A proof file starts with an expression indicating the version and level it conforms to. This is followed by an environment expression, and then, one or more proofs. An example of a complete proof file of the proof in Figure 4 can be found in Appendix A.

4 Basic inference rules

This section describes all the justifications in more detail. They are arranged in three groups: primitive rules, miscellaneous functions and derived rules. Each groups is presented in a subsection. Justifications in each group are arranged in alphabetical order of their names. Each justification begins with the syntax rule enclosing in a box. This is followed by the inference rule in conventional form as in Equation (1) on page 4. Then, the ML function implementing this rule with its type is given, and it is followed by a more detailed description. For derived inference rules, a proof will be presented as well. This description of the justifications aims to provide all necessary information to proof checker developers.

4.1 Primitive rules

- **Abstraction**

  ```
  ABS term NUMBER
  ```

  $$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.t_1) = (\lambda x.t_2)}$$

  `ABS : term -> thm -> thm`

  The `term` argument must be a variable, and it is not free in the assumptions of the hypothesis Γ. The `NUMBER` refers to the theorem in a line having the given line number. This theorem must be an equation.

- **Assumption introduction**

  ```
  ASSUME term
  ```

  $$\frac{}{t \vdash t}$$

  `ASSUME : term -> thm`

  The term t must be of type : *bool*.

- **β-conversion**

  ```
  BETACONV term
  ```

  $$\frac{}{\vdash (\lambda x.t_1)t_2 = t_1[t_2/x]}$$

  `BETA_CONV : term -> thm`

  The `term` argument must be a β-redex in the form $(\lambda x.t_1)t_2$. The right-hand side of the resulting theorem is obtained by substitute t_2 for x in t_1 with suitable renaming of free variables in t_2 to avoid variable capture.

- **Discharging an assumption**

```
DISCH term NUMBER
```

$$\frac{\Gamma \vdash t_2}{\Gamma - \{t_1\} \vdash t_1 \supset t_2}$$

```
DISCH : term -> thm -> thm
```

The term t_1 must be of type : *bool*. The `NUMBER` refers to the theorem in the line having the given number. The expression $\Gamma - \{t_1\}$ denotes the set subtraction of $\{t_1\}$ from Γ. If t_1 is not in Γ, the result of the subtraction is Γ itself.

- **Type instantiation**

```
INSTTYPE type_type_list NUMBER
```

$$\frac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

```
INST_TYPE : (type # type)list -> thm -> thm
```

The first argument is a list of type pairs $[(\sigma_1, \alpha_1); \ldots; (\sigma_n, \alpha_n)]$ which specifies the simultaneous type substitutions to be made in the theorem referred to by `NUMBER`. The second fields α_i of the pairs must be type variables, and none of any α_i occurs in any assumption in Γ. All occurrences of α_i in t are replaced by the corresponding δ_i. Furthermore, if distinct variables in t become identified after the instantiation, they will be renamed.

- **Modus Ponens**

```
MP NUMBER NUMBER
```

$$\frac{\Gamma_1 \vdash t_1 \supset t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

```
MP : thm -> thm -> thm
```

The theorem referred to by the first `NUMBER` must be an implication. The theorem referred to by the second `NUMBER` must match the antecedent of the first theorem exactly.

- **Reflexivity**

```
REFL term
```

$$\overline{\vdash t = t}$$

```
REFL : term -> thm
```

The term t must be of type : *bool*.

- **Substitution**

```
SUBST NUMBER_term_list term NUMBER
```

$$\frac{\Gamma_1 \vdash t_1 = t_1' \quad \cdots \quad \Gamma_n \vdash t_n = t_n' \quad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t_1', \ldots, t_n'/t_1, \ldots, t_n]}$$

`SUBST : (thm # term)list -> term -> thm -> thm`

The first argument is a list of pairs whose first fields are equational theorems $t_i = t_i'$, and whose second fields are simple variables x_i. The second argument is a term of the form $t[x_1, \ldots, x_n]$. It should match the conclusion of the theorem referred to by `NUMBER`. The variables x_i in the term mark the places where substitutions using the theorems $\vdash t_i = t_i'$ are to be done. The type of x_i must be the same as of t_i. Free variables in t_i' may be renamed to avoid capture.

4.2 Miscellaneous functions

- **Retrieving an axiom**

```
AXIOM STRING STRING
```

`axiom : string -> string -> thm`

This justification indicates that the theorem in this inference step is an axiom whose name is the second string and which is stored in the theory specified as the first string. The well-typedness of the axiom should be checked.

- **Retrieving a definition**

```
DEFINITION STRING STRING
```

`definition : string -> string -> thm`

This justification indicates that the theorem in this inference step is a previously defined definition whose name is the second string and which is stored in the theory specified as the first string. The well-typedness of the theorem should be checked.

- **Create a definitional theorem**

```
DEFEXISTSRULE term
```

`DEF_EXISTS_RULE : term -> thm`

This justification introduces a new definitional theorem. The input term must be an equation and both sides must be of the same type.

- **Hypothesis**

```
HYPOTHESIS
```

This justification indicates that the theorem is one of the initial theorems of the proof. It should have been proved in a previous proof.

- **Introducing a new axiom**

```
NEWAXIOM STRING term
```

$$\frac{}{\vdash \forall x_1 \ldots x_n.\, t[x_1, \ldots, x_n]}$$

`new_axiom : term -> thm`

This justification introduces a new axiom. It is in the form given as the `term`. The `STRING` is the name of the axiom. All free variables in the input term are automatically generalized.

- **Introducing a new constant**

```
NEWCONSTANT STRING type
```

`new_constant : string -> type -> void`

This justification declares a new constant whose name is the given `STRING` and whose type is the given `type`. The name should be unique, i.e., not already the name of an existing constant, and the type should be well-formed. The current signature should be updated. The theorem in this inference step is not used. To satisfy the type checker, the theorem `TRUTH` is used as a dummy.

- **Introducing a new type**

```
NEWTYPE NUMBER STRING
```

`new_type : int -> string -> void`

This justification declares a new type construtor whose name is the given `STRING` and whose arity is the given `NUMBER`. The name should be unique, i.e., not already the name of an existing type constructor. The current type structure should be updated. The theorem in this inference step is not used. To satisfy the type checker, the theorem `TRUTH` is used as a dummy.

- **Definition of non-zero numbers**

```
NUMCONV term
```

$$\frac{}{\vdash n = \mathsf{SUC}\, m}$$

`num_CONV : term -> thm`

The input term must be a constant denoting a non-zero natural number. m is a numeric constant denoting the predecessor of n.

- **Storing a definitiion**

  `STOREDEFINITION STRING term`

  `store_definition : term -> thm`

  This justification introduces a new definition. In fact, making a new definition is a three-step process:

  1. a theorem asserting the existence of the definition is derived with the justification `DEFEXISTSRULE`;
  2. a new constant is declared with the justification `NEWCONSTANT`;
  3. the definition is saved in the current theory with the justification `STOREDEFINITION`.

  Since this file format allows only the four kinds of primitive terms, special syntactic status of constants, i.e., infix or binder, are not recognized. The input term must be an equation and both sides are of the same type.

- **Retrieving a theorem**

  `THEOREM STRING STRING`

  `theorem : string -> string -> thm`

  This justification indicates the theorem in this inference step has been derived previously. It has been stored in the theory, whose name is the first string, under the name specified as the second string. The well-typedness of the theorem should be checked.

4.3 Derived rules

There are 35 derived inference rules. The derivations of 14 of them use only the primitive rules. They are:

```
ADDASSUM     EQIMPRULEL  NOTELIM
ALPHA        EQIMPRULER  NOTINTRO
APTERM       EQMP        SPEC
APTHM        IMPTRANS    SYM
             MKCOMB      TRANS
```

The three derived substitution rules `SUBS`, `SUBSOCCS` and `SUBSTCONV` are simple variants of the primitive substitution rule `SUBST`. The difference is only in the argument which indicates which occurrence of a variable is to substituted. Therefore, no derivation is shown for them. Since the HOL logic considers terms are equal up to α-conversion. No derivation is shown for the rule `ALPHA` representing this conversion. The derivations of the remaining 17 rules use the derived rules listed above in addition to the primitive rules, thus, they are less tedious. Some of these derivations can be found in Section 22.3 of [3].

- **Adding an assumption**

```
ADDASSUM term NUMBER
```

$$\frac{\Gamma \vdash t}{\Gamma,\ t' \vdash t}$$

`ADD_ASSUM : term -> thm -> thm`

The term is the new assumption t' to be added to the theorem.

1.	$t' \vdash t'$	[ASSUME]
2.	$\Gamma \vdash t$	[Hypothesis]
3.	$\Gamma \vdash t' \supset t$	[DISCH 2]
4.	$\Gamma,\ t' \vdash t$	[MP 3,1]

- **α-conversion**

```
ALPHA term term
```

$$\overline{\vdash t_1 = t_2}$$

`ALPHA : term -> term -> thm`

The input terms t_1 and t_2 must be α-equivalent, otherwise, it fails.

- **Application of a term to a theorem**

```
APTERM term NUMBER
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t\, t_1 = t\, t_2}$$

`AP_TERM : term -> thm -> thm` The input term must have a function type whose domain is the type of the left-hand side (or right-hand side)[2] of the input theorem.

1.	$\Gamma \vdash t_1 = t_2$	[Hypothesis]
2.	$\vdash t\, t_1 = t\, t_1$	[REFL]
3.	$\Gamma \vdash t\, t_1 = t\, t_2$	[SUBST 1,2]

[2]The type of both sides must be the same.

- **Application of a theorem to a term**

```
APTHM NUMBER term
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_1\, t = t_2\, t}$$

`AP_THM : thm -> term -> thm` The input term must have the same type as the domain of the left-hand side (or the right-hand side)[3] of the input theorem.

1.	$\Gamma \vdash t_1 = t_2$	[Hypothesis]
2.	$\vdash t_1\, t = t_1\, t$	[`REFL`]
3.	$\Gamma \vdash t_1\, t = t_2\, t$	[`SUBST` 1,2]

- **Classical contradiction rule**

```
CCONTR term NUMBER
```

$$\frac{\Gamma,\ \neg t \vdash \mathsf{F}}{\Gamma \vdash t}$$

`CCONTR : term -> thm -> thm`

The theorem referred to by the `NUMBER` must have F as its conclusion. The input term t should be of type : *bool*, and the negation of it should occur in the assumption of the input theorem.

1.	$\vdash \neg = \lambda b.\, b \supset \mathsf{F}$	[Definition of $\neg$]
2.	$\vdash \neg t = (\lambda b.\, b \supset \mathsf{F})t$	[`AP_THM` 1]
3.	$\vdash (\lambda b.\, b \supset \mathsf{F})t = t \supset \mathsf{F}$	[`BETA_CONV`]
4.	$\vdash \neg t = t \supset \mathsf{F}$	[`TRANS` 2,3]
5.	$\Gamma,\ \neg t \vdash \mathsf{F}$	[Hypothesis]
6.	$\Gamma \vdash \neg t \supset \mathsf{F}$	[`DISCH` 5]
7.	$\Gamma \vdash (t \supset \mathsf{F}) \supset \mathsf{F}$	[`SUBST` 4,6]
8.	$t = \mathsf{F} \vdash t = \mathsf{F}$	[`ASSUME`]
9.	$\Gamma,\ t = \mathsf{F} \vdash (\mathsf{F} \supset \mathsf{F}) \supset \mathsf{F}$	[`SUBST` 8,7]
10.	$\mathsf{F} \vdash \mathsf{F}$	[`ASSUME`]
11.	$\vdash \mathsf{F} \supset \mathsf{F}$	[`DISCH` 10]
12.	$\Gamma,\ t = \mathsf{F} \vdash \mathsf{F}$	[`MP` 9,11]
13.	$\vdash \mathsf{F} = \forall b.\, b$	[Definition of F]
14.	$\Gamma,\ t = \mathsf{F} \vdash \forall b.\, b$	[`SUBST` 13,12]
15.	$\Gamma,\ t = \mathsf{F} \vdash t$	[`SPEC` 14]
16.	$\vdash \forall b.\, (b = \mathsf{T}) \vee (b = \mathsf{F})$	[Axiom `EXCLUDED_MIDDLE`]
17.	$\vdash (t = \mathsf{T}) \vee (t = \mathsf{F})$	[`SPEC` 16]
18.	$t = \mathsf{T} \vdash t = \mathsf{T}$	[`ASSUME`]
19.	$t = \mathsf{T} \vdash \mathsf{T} = t$	[`SYM` 18]
20.	$\vdash \mathsf{T}$	[Theorem `TRUTH`]
21.	$t = \mathsf{T} \vdash t$	[`EQ_MP` 19,20]
22.	$\Gamma \vdash t$	[`DISJ_CASES` 17,21,15]

- **$\exists$-elimination**

```
CHOOSE term NUMBER NUMBER
```

$$\frac{\Gamma_1 \vdash \exists x.\, t[x] \quad \Gamma_2,\ t[v] \vdash t'}{\Gamma_1 \cup \Gamma_2 \vdash t'}$$

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

The input `term` must be a variable v and its type must be the same as the existentially quantified variable x in the first theorem. $t[v]$ is a term occurring in the assumptions of the second theorem. It is the same as $t[x]$, the body of the first theorem, up to α-conversion. The variable v must not occur free in the conclusion of the first theorem, i.e., $\exists x.\, t[x]$, and neither can it occur free in Γ_2 or t'.

1.	$\vdash \exists = \lambda P.\, P(\varepsilon\ P)$	[Definition of $\exists$[3]]
2.	$\vdash \exists(\lambda x.\, t[x]) = (\lambda P.\, P(\varepsilon\ P))(\lambda x.\, t[x])$	[`AP_THM` 1]
3.	$\Gamma_1 \vdash \exists(\lambda x.\, t[x])$	[Hypothesis]
4.	$\Gamma_1 \vdash (\lambda P.\, P(\varepsilon\ P))(\lambda x.\, t[x])$	[`EQ_MP` 2,3]
5.	$\vdash (\lambda P.\, P(\varepsilon\ P))(\lambda x.\, t[x]) = (\lambda x.\, t[x])(\varepsilon(\lambda x.\, t[x]))$	[`BETA_CONV`]
6.	$\Gamma_1 \vdash (\lambda x.\, t[x])(\varepsilon(\lambda x.\, t[x])$	[`EQ_MP` 5,4]
7.	$\vdash (\lambda x.\, t[x])v = t[v]$	[`BETA_CONV`]
8.	$\vdash t[v] = (\lambda x.\, t[x])v$	[`SYM` 7]
9.	$\Gamma_2,\ t[v] \vdash t'$	[Hypothesis]
10.	$\Gamma_2 \vdash t[v] \supset t'$	[`DISCH` 9]
11.	$\Gamma_2 \vdash (\lambda x.\, t[x])v \supset t'$	[`SUBST` 8,10]
12.	$(\lambda x.\, t[x])v \vdash (\lambda x.\, t[x])v$	[`ASSUME`]
13.	$\Gamma_2,\ (\lambda x.\, t[x])v \vdash t'$	[`MP` 11,12]
14.	$\Gamma_1 \cup \Gamma_2 \vdash t'$	[`SELECT_ELIM` 6,13]

[3] with suitable type instantiation.

- **$\wedge$-introduction**

```
CONJ NUMBER NUMBER
```

$$\frac{\Gamma_1 \vdash t_1 \quad \Gamma_2 \vdash t_2}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2}$$

`CONJ : thm -> thm -> thm`

The two `NUMBER`s refer to two theorems which are to be combined by the $\wedge$ operator.

1.	$\vdash \wedge = \lambda b_1\, b_2.\, \forall b.\, (b_1 \supset (b_2 \supset b)) \supset b$	[Definition of $\wedge$]
2.	$\vdash \$\wedge t_1 = \$\wedge t_1$	[`REFL`]
3.	$\vdash (\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset (b_2 \supset b)) \supset b) t_1 =$ $\lambda b_2.\, \forall b.\, (t_1 \supset (b_2 \supset b)) \supset b$	[`BETA_CONV`]
4.	$\vdash \$\wedge t_1 = (\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset (b_2 \supset b)) \supset b) t_1$	[`SUBST` 1,2]
5.	$\vdash \$\wedge t_1 = \lambda b_2.\, \forall b.\, (t_1 \supset (b_2 \supset b)) \supset b$	[`SUBST` 3,4]
6.	$\vdash t_1 \wedge t_2 = t_1 \wedge t_2$	[`REFL`]
7.	$\vdash (\lambda b_2.\, \forall b.\, (t_1 \supset (b_2 \supset b)) \supset b) t_2$ $= \forall b.\, (t_1 \supset (t_2 \supset b)) \supset b)$	[`BETA_CONV`]
8.	$\vdash t_1 \wedge t_2 = (\lambda b_2.\, \forall b.\, (t_1 \supset (b_2 \supset b)) \supset b) t_2$	[`SUBST` 5,6]
9.	$\vdash t_1 \wedge t_2 = \forall b.\, (t_1 \supset (t_2 \supset b)) \supset b$	[`SUBST` 7,8]
10.	$t_1 \supset (t_2 \supset b) \vdash t_1 \supset (t_2 \supset b)$	[`ASSUME`]
11.	$\Gamma_1 \vdash t_1$	[Hypothesis]
12.	$\Gamma_1,\ t_1 \supset (t_2 \supset b) \vdash t_2 \supset b$	[`MP` 10,11]
13.	$\Gamma_2 \vdash t_2$	[Hypothesis]
14.	$\Gamma_1 \cup \Gamma_2,\ t_1 \supset (t_2 \supset b) \vdash b$	[`MP` 12,13]
15.	$\Gamma_1 \cup \Gamma_2 \vdash (t_1 \supset (t_2 \supset b)) \supset b$	[`DISCH` 14]
16.	$\Gamma_1 \cup \Gamma_2 \vdash \forall b.\, (t_1 \supset (t_2 \supset b)) \supset b$	[`GEN` 15]
17.	$\vdash \forall b.\, (t_1 \supset (t_2 \supset b)) \supset b = t_1 \wedge t_2$	[`SUBST` 9,6]
18.	$\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2$	[`SUBST` 17,16]

- **$\wedge$-elimination(left)**

`CONJUNCT1 NUMBER`

$$\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_1}$$

`CONJUNCT1 : thm -> thm`

Extract the left conjunct from the theorem referred to by `NUMBER`. The conclusion of the input theorem must be a conjunction.

1.	$\vdash \wedge = \lambda b_1\, b_2.\, \forall b.\, (b_1 \supset (b_2 \supset b)) \supset b$	[Definition of $\wedge$]
2.	$\vdash \$\wedge t_1 = \$\wedge t_1$	[`REFL`]
3.	$\vdash (\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset (b_2 \supset b)) \supset b) t_1$	
	$= \lambda b_2.\, \forall b.\, (t_1 \supset (b_2 \supset b)) \supset b$	[`BETA_CONV`]
4.	$\vdash \$\wedge t_1 = (\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset (b_2 \supset b)) \supset b) t_1$	[`SUBST` 1,2]
5.	$\vdash \$\wedge t_1 = \lambda b_2.\, \forall b.\, (t_1 \supset (b_2 \supset b)) \supset b$	[`SUBST` 3,4]
6.	$\vdash t_1 \wedge t_2 = t_1 \wedge t_2$	[`REFL`]
7.	$\vdash (\lambda b_2.\, \forall b.\, (t_1 \supset (b_2 \supset b)) \supset b) t_2$	
	$= \forall b.\, (t_1 \supset (t_2 \supset b)) \supset b)$	[`BETA_CONV`]
8.	$\vdash t_1 \wedge t_2 = (\lambda b_2.\, \forall b.\, (t_1 \supset (b_2 \supset b)) \supset b) t_2$	[`SUBST` 5,6]
9.	$\vdash t_1 \wedge t_2 = \forall b.\, (t_1 \supset (t_2 \supset b)) \supset b$	[`SUBST` 7,8]
10.	$\Gamma \vdash t_1 \wedge t_2$	[Hypothesis]
11.	$\Gamma \vdash \forall b.\, (t_1 \supset (t_2 \supset b)) \supset b$	[`SUBST` 9,10]
12.	$\Gamma \vdash (t_1 \supset (t_2 \supset t_1)) \supset t_1$	[`SPEC` 11]
13.	$t_1 \vdash t_1$	[`ASSUME`]
14.	$t_1 \vdash t_2 \supset t_1$	[`DISCH` 13]
15.	$\vdash t_1 \supset (t_2 \supset t_1)$	[`DISCH` 14]
16.	$\Gamma \vdash t_1$	[`MP` 12,15]

- **$\wedge$-elimination(right)**

```
CONJUNCT2 NUMBER
```

$$\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_2}$$

`CONJUNCT2 : thm -> thm`

Extract the left conjunct from the theorem referred to by `NUMBER`. The conclusion of the input theorem must be a conjunction.

1.	$\vdash \wedge = \lambda b_1\, b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b$	[Definition of $\wedge$]
2.	$\vdash \$\wedge t_1 = \$\wedge t_1$	[`REFL`]
3.	$\vdash (\lambda b_1\, b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b)t_1$ $= \lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b$	[`BETA_CONV`]
4.	$\vdash \$\wedge t_1 = (\lambda b_1\, b_2. \forall b. (b_1 \supset (b_2 \supset b)) \supset b)t_1$	[`SUBST` 1,2]
5.	$\vdash \$\wedge t_1 = \lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b$	[`SUBST` 3,4]
6.	$\vdash t_1 \wedge t_2 = t_1 \wedge t_2$	[`REFL`]
7.	$\vdash (\lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b)t_2$ $= \forall b. (t_1 \supset (t_2 \supset b)) \supset b)$	[`BETA_CONV`]
8.	$\vdash t_1 \wedge t_2 = (\lambda b_2. \forall b. (t_1 \supset (b_2 \supset b)) \supset b)t_2$	[`SUBST` 5,6]
9.	$\vdash t_1 \wedge t_2 = \forall b. (t_1 \supset (t_2 \supset b)) \supset b$	[`SUBST` 7,8]
10.	$\Gamma \vdash t_1 \wedge t_2$	[Hypothesis]
11.	$\Gamma \vdash \forall b. (t_1 \supset (t_2 \supset b)) \supset b$	[`SUBST` 9,10]
12.	$\Gamma \vdash (t1_1 \supset (t_2 \supset t_2)) \supset t_2$	[`SPEC` 11]
13.	$t_2 \vdash t_2$	[`ASSUME`]
14.	$\vdash t_2 \supset t_2$	[`DISCH` 13]
15.	$\vdash t_1 \supset (t_2 \supset t_2)$	[`DISCH` 14]
16.	$\Gamma \vdash t_2$	[`MP` 12,15]

- **Intuitionistic contradiction rule**

```
CONTR term NUMBER
```

$$\frac{\Gamma \vdash \mathsf{F}}{\Gamma \vdash t}$$

`CONTR : term -> thm -> thm`

The theorem referred to by `NUMBER` should have falsity (F) as its conclusion.

1.	$\vdash \forall t. \mathsf{F} \supset t$	[Theorem `FALSITY`]
2.	$\Gamma \vdash \mathsf{F}$	[Hypothesis]
3.	$\vdash \mathsf{F} \supset t$	[`SPEC` 1]
4.	$\Gamma \vdash t$	[`MP` 3,2]

- **Right $\vee$-introduction**

```
DISJ1 NUMBER term
```

$$\frac{\Gamma \vdash t_1}{\Gamma \vdash t_1 \vee t_2}$$

`DISJ1 : thm -> term -> thm`

The result of this inference rule is a disjunctive theorem whose second disjunct is the input term. This term must be of type `:bool`.

1.	$\vdash \vee = \lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b$	[Definition of $\vee$]
2.	$\vdash \$\vee t_1 = \$\vee t_1$	[REFL]
3.	$(\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b)t_1$ $= \lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b$	[BETA_CONV]
4.	$\vdash \$\vee t_1 = (\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b)t_1$	[SUBST 1,2]
5.	$\vdash \$\vee t_1 = \lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b$	[SUBST 3,4]
6.	$\vdash t_1 \vee t_2 = t_1 \vee t_2$	[REFL]
7.	$(\lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b)t_2$ $= \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[BETA_CONV]
8.	$\vdash t_1 \vee t_2 = (\lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b)t_2$	[SUBST 5,6]
9.	$\vdash t_1 \vee t_2 = \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[SUBST 7,8]
10.	$\Gamma \vdash t_1$	[Hypothesis]
11.	$t_1 \supset b \vdash t_1 \supset b$	[ASSUME]
12.	$\Gamma,\ t_1 \supset b \vdash b$	[MP 11,10]
13.	$\Gamma,\ t_1 \supset b \vdash (t_2 \supset b) \supset b$	[DISCH 12]
14.	$\Gamma \vdash (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[DISCH 13]
15.	$\Gamma \vdash \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[GEN 14]
16.	$\vdash \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b = t_1 \vee t_2$	[SUBST 9,6]
17.	$\Gamma \vdash t_1 \vee t_2$	[SUBST 16,15]

- **Left $\vee$-introduction**

```
DISJ2 term NUMBER
```

$$\frac{\Gamma \vdash t_2}{\Gamma \vdash t_1 \vee t_2}$$

`DISJ2 : term -> thm -> thm`

The result of this inference rule is a disjunctive theorem whose first disjunct is the input term. This term must be of type `:bool`.

1.	$\vdash \vee = \lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b$	[Definition of $\vee$]
2.	$\vdash \$\vee t_1 = \$\vee t_1$	[REFL]
3.	$(\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b)t_1$ $= \lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b$	[BETA_CONV]
4.	$\vdash \$\vee t_1 = (\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b)t_1$	[SUBST 1,2]
5.	$\vdash \$\vee t_1 = \lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b$	[SUBST 3,4]
6.	$\vdash t_1 \vee t_2 = t_1 \vee t_2$	[REFL]
7.	$(\lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b)t_2$ $= \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[BETA_CONV]
8.	$\vdash t_1 \vee t_2 = (\lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b)t_2$	[SUBST 5,6]
9.	$\vdash t_1 \vee t_2 = \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[SUBST 7,8]
10.	$\Gamma \vdash t_2$	[Hypothesis]
11.	$t_2 \supset b \vdash t_2 \supset b$	[ASSUME]
12.	$\Gamma,\ t_2 \supset b \vdash b$	[MP 11,10]
13.	$\Gamma,\ t_2 \supset b \vdash (t_2 \supset b) \supset b$	[DISCH 12]
14.	$\Gamma \vdash (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[DISCH 13]
15.	$\Gamma \vdash \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[GEN 14]
16.	$\vdash \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b = t_1 \vee t_2$	[SUBST 9,6]
17.	$\Gamma \vdash t_1 \vee t_2$	[SUBST 16,15]

- **$\vee$-elimination**

```
DISJCASES NUMBER NUMBER NUMBER
```

$$\frac{\Gamma \vdash t_1 \vee t_2 \quad \Gamma_1,\ t_1 \vdash t \quad \Gamma_2,\ t_2 \vdash t}{\Gamma \cup \Gamma_1 \cup \Gamma_2 \vdash t}$$

`DISJ_CASES : thm -> thm -> thm -> thm`

The theorem referred to by the first `NUMBER` must be a disjunction. The assumptions of the second and the third theorems must include the first and second disjunct of the first theorem, respectively.

1.	$\vdash \vee = \lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b$	[Definition of $\vee$]
2.	$\vdash \$\vee t_1 = \$\vee t_1$	[REFL]
3.	$(\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b) t_1 =$ $\lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b$	[BETA_CONV]
4.	$\vdash \$\vee t_1 = (\lambda b_1\, b_2.\, \forall b.\, (b_1 \supset b) \supset (b_2 \supset b) \supset b) t_1$	[SUBST 1,2]
5.	$\vdash \$\vee t_1 = \lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b$	[SUBST 3,4]
6.	$\vdash t_1 \vee t_2 = t_1 \vee t_2$	[REFL]
7.	$(\lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b) t_2$ $= \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[BETA_CONV]
8.	$\vdash t_1 \vee t_2 = (\lambda b_2.\, \forall b.\, (t_1 \supset b) \supset (b_2 \supset b) \supset b) t_2$	[SUBST 5,6]
9.	$\vdash t_1 \vee t_2 = \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[SUBST 7,8]
10.	$\Gamma \vdash t_1 \vee t_2$	[Hypothesis]
11.	$\Gamma \vdash \forall b.\, (t_1 \supset b) \supset (t_2 \supset b) \supset b$	[SUBST 9,10]
12.	$\Gamma \vdash (t_1 \supset t) \supset (t_2 \supset t) \supset t$	[SPEC 11]
13.	$\Gamma_1,\ t_1 \vdash t$	[Hypothesis]
14.	$\Gamma_1 \vdash t_1 \supset t$	[DISCH 13]
15.	$\Gamma \cup \Gamma_1 \vdash (t_2 \supset t) \supset t$	[MP 12,14]
16.	$\Gamma_2,\ t_2 \vdash t$	[Hypothesis]
17.	$\Gamma_2 \vdash t_2 \supset t$	[DISCH 16]
18.	$\Gamma \cup \Gamma_1 \cup \Gamma_2 \vdash t$	[MP 15,17]

- **Implication from equality**

```
EQIMPRULEL NUMBER
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 \supset t_1}$$

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

The theorem referred to by `NUMBER` must be an equation, and both sides of it must be of type `:bool`. The resulting theorem of this justification is the second theorem returned by the function `EQ_IMP_RULE`. It is an implication whose antecedent is the right-hand side of the hypothesis and whose conclusion is the left-hand side of the hypothesis.

1.	$\Gamma \vdash t_1 = t_2$	[Hypothesis]
2.	$t_2 \vdash t_2$	[ASSUME]
3.	$\Gamma \vdash t_2 = t_1$	[SYM 1]
4.	$\Gamma,\ t_2 \vdash t_1$	[SUBST 3,2]
5.	$\Gamma \vdash t_2 \supset t_1$	[DISCH 4]

- **Implication from equality(right)**

```
EQIMPRULER NUMBER
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_1 \supset t_2}$$

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

The theorem referred to by `NUMBER` must be an equation, and both sides of it must be of type `:bool`. The resulting theorem of this justification is the first theorem returned by the function `EQ_IMP_RULE`. It is an implication whose antecedent is the left-hand side of the hypothesis and whose conclusion is the right-hand side of the hypothesis .

1.	$\Gamma \vdash t_1 = t_2$	[Hypothesis]
2.	$t_1 \vdash t_1$	[`ASSUME`]
3.	$\Gamma,\ t_1 \vdash t_2$	[`SUBST` 1,2]
4.	$\Gamma \vdash t_1 \supset t_2$	[`DISCH` 3]

- **Modus Ponens for equality**

```
EQMP NUMBER NUMBER
```

$$\frac{\Gamma_1 \vdash t_1 = t_2 \quad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

`EQ_MP : thm -> thm -> thm`

The first theorem should be an equation. The second theorem should be identical to the left-hand side of the first. The resulting theorem is the right-hand side of the first theorem.

1.	$\Gamma_1 \vdash t_1 = t_2$	[Hypothesis]
2.	$\Gamma_2 \vdash t_1$	[Hypothesis]
3.	$\Gamma_1 \cup \Gamma_2 \vdash t_2$	[`SUBST` 1,2]

- **Equality-with-T introduction**

```
EQTINTRO NUMBER
```

$$\frac{\Gamma \vdash t}{\Gamma \vdash t = \mathsf{T}}$$

`EQT_INTRO : thm -> thm`

1.	$\vdash \forall b_1\, b_2.\, (b_1 \supset b_2) \supset (b_2 \supset b_1) \supset (b_1 = b_2)$	[IMP_ANTISYM_AX]
2.	$\vdash \forall b_2.\, (t \supset b_2) \supset (b_2 \supset t) \supset (t = b_2)$	[SPEC 1]
3.	$\vdash (t \supset \mathsf{T}) \supset (\mathsf{T} \supset t) \supset (t = \mathsf{T})$	[SPEC 2]
4.	$\vdash \mathsf{T}$	[TRUTH]
5.	$\vdash t \supset \mathsf{T}$	[DISCH 4]
6.	$\vdash (\mathsf{T} \supset t) \supset (t = \mathsf{T})$	[MP 3,5]
7.	$\Gamma \vdash t$	[Hypothesis]
8.	$\Gamma \vdash \mathsf{T} \supset t$	[DISCH 7]
9.	$\Gamma \vdash t = \mathsf{T}$	[MP 6,8]

- **η-conversion**

```
ETACONV term
```

$$\overline{\vdash (\lambda x'.\, t\, x') = t}$$

`ETA_CONV  : term -> thm`

The variable x' does not occur free in t. The input `term` is the same as the left-hand side of the resulting theorem.

1.	$\vdash \forall f.\, (\lambda x.\, f\, x) = f$	[ETA_AX[4]]
2.	$\vdash (\lambda x.\, t\, x) = t$	[SPEC 1]
3.	$\vdash (\lambda x'.\, t\, x') = (\lambda x.\, t\, x)$	[α-conversion[5]]
4.	$\vdash (\lambda x'.\, t\, x') = t$	[TRANS 3,2]

- **$\exists$-introduction**

```
EXISTS term term NUMBER
```

$$\frac{\Gamma \vdash t[t_2/x]}{\Gamma \vdash \exists x.\, t[x]}$$

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

The first term is an existentially quantified term which matches exactly the conclusion of the resulting theorem. The second term t_2 must be of the same type as the bound variable x. When substituting it into t for the bound variable x, it results in a theorem which is the same as the input theorem referred to by `NUMBER`.

[4] with appropriate type instantiation.

[5] Two terms are considered equal up to α-conversion.

1. $\vdash (\lambda x.\, t_1[x])t_2 = t_1[t_2]$ [BETA_CONV]
2. $\vdash t_1[t_2] = (\lambda x.\, t_1[x])t_2$ [SYM 1]
3. $\Gamma \vdash t_1[t_2]$ [Hypothesis]
4. $\Gamma \vdash (\lambda x.\, t_1[x])t_2$ [EQ_MP 2,3]
5. $\Gamma \vdash (\lambda x.\, t_1[x])(\varepsilon(\lambda x.\, t_1[x]))$ [SELECT_INTRO 4]
6. $\vdash \exists = \forall P.\, P(\varepsilon\ P)$ [Definition of $\exists$]
7. $\vdash \exists(\lambda x.\, t_1[x]) = (\lambda P.\, P(\varepsilon\ P))(\lambda x.\, t_1[x])$ [AP_THM 6]
8. $\vdash (\lambda P.\, P(\varepsilon\ P))(\lambda x.\, t_1[x]) = (\lambda x.\, t_1[x])(\varepsilon(\lambda x.\, t_1[x]))$ [BETA_CONV]
9. $\vdash \exists(\lambda x.\, t_1[x]) = (\lambda x.\, t_1[x])(\varepsilon(\lambda x.\, t_1[x]))$ [TRANS 7,8]
10. $\vdash (\lambda x.\, t_1[x])(\varepsilon(\lambda x.\, t_1[x])) = \exists(\lambda x.\, t_1[x])$ [SYM 9]
11. $\Gamma \vdash \exists(\lambda x.\, t_1[x])$ [EQ_MP 10,5]

- **extensionality**

```
EXT NUMBER
```

$$\frac{\Gamma \vdash \forall x.\, t_1\ x = t_2\ x}{\Gamma \vdash t_1 = t_2}$$

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

The variable x' in the proof below is a new variable which does not occur free anywhere in the input theorem.

1. $\Gamma \vdash \forall x.\, t_1\ x = t_2\ x$ [Hypothesis]
2. $\Gamma \vdash t_1\ x' = t_2\ x'$ [SPEC 1]
3. $\Gamma \vdash (\lambda x'.\, t_1\ x') = (\lambda x'.\, t_2\ x')$ [ABS 2]
4. $\vdash (\lambda x'.\, t_1\ x') = t_1$ [ETA_CONV]
5. $\vdash t_1 = (\lambda x'.\, t_1\ x')$ [SYM 4]
6. $\Gamma \vdash t_1 = (\lambda x'.\, t_2\ x')$ [TRANS 5,3]
7. $\vdash (\lambda x'.\, t_2\ x') = t_2$ [ETA_CONV]
8. $\Gamma \vdash t_1 = t_2$ [TRANS 6,7]

- **Generalization($\forall$-introduction)**

```
GEN term NUMBER
```

$$\frac{\Gamma \vdash t}{\Gamma \vdash \forall x.\, t}$$

```
GEN : term -> thm -> thm
```

The input term t is a variable which does not occur free in the assumption Γ.

[5] with appropriate type instantiation.

1.	$\Gamma \vdash t$	[Hypothesis]
2.	$\Gamma \vdash t = \mathsf{T}$	[`EQT_INTRO` 1]
3.	$\Gamma \vdash (\lambda x.\, t) = (\lambda x.\, \mathsf{T})$	[`ABS` 2]
4.	$\vdash \forall(\lambda x.\, t) = \forall(\lambda x.\, t)$	[`REFL`]
5.	$\vdash \forall = (\lambda P.\, P = (\lambda x.\, \mathsf{T}))$	[Definition of $\forall$[6]]
6.	$\vdash \forall(\lambda x.\, t) = (\lambda P.\, P = (\lambda x.\, \mathsf{T}))(\lambda x.\, t)$	[`SUBST` 5,4]
7.	$\vdash (\lambda P.\, P = (\lambda x.\, \mathsf{T}))(\lambda x.\, t) = ((\lambda x.\, t) = (\lambda x.\, \mathsf{T}))$	[`BETA_CONV`]
8.	$\vdash \forall(\lambda x.\, t) = ((\lambda x.\, t) = (\lambda x.\, \mathsf{T}))$	[`TRANS` 6,7]
9.	$\vdash ((\lambda x.\, t) = (\lambda x.\, \mathsf{T})) = \forall(\lambda x.\, \mathsf{T})$	[`SYM` 8]
10.	$\Gamma \vdash \forall(\lambda x.\, t)$	[`EQ_MP` 9,3]

- **Deducing equality from implications**

```
IMPANTISYMRULE NUMBER NUMBER
```

$$\frac{\Gamma_1 \vdash t_1 \supset t_2 \quad \Gamma_2 \vdash t_2 \supset t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_2}$$

`IMP_ANTISYM_RULE : thm -> thm -> thm`

The two theorems referred to by the numbers should be a pair of implications. The antecedent of one is the same as the conclusion of the other, and vice versa.

1.	$\vdash \forall b_1\, b_2.\, (b_1 \supset b_2) \supset (b_2 \supset b_1) \supset (b_1 = b_2)$	[`IMP_ANTISYM_AX`]
2.	$\Gamma_1 \vdash t_1 \supset t_2$	[Hypothesis]
3.	$\Gamma_2 \vdash t_2 \supset t_1$	[Hypothesis]
4.	$\vdash \forall b_2.\, (t_1 \supset b_2) \supset (b_2 \supset t_1) \supset (t_1 = b_2)$	[`SPEC`]
5.	$\vdash (t_1 \supset t_2) \supset (t_2 \supset t_1) \supset (t_1 = t_2)$	[`SPEC`]
6.	$\Gamma_1 \vdash (t_2 \supset t_1) \supset (t_1 = t_2)$	[`MP` 5,2]
7.	$\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_2$	[`MP` 6,3]

- **Transitivity of implications**

```
IMPTRANS NUMBER NUMBER
```

$$\frac{\Gamma_1 \vdash t_1 \supset t_2 \quad \Gamma_2 \vdash t_2 \supset t_3}{\Gamma_1 \cup \Gamma_2 \vdash t_1 \supset t_3}$$

`IMP_TRANS : thm -> thm -> thm`

Both theorems referred to by the numbers must be implications. The conclusion of the first theorem must be the same as the antecedent of the second theorem.

[6] with appropriate type instantiation.

1.	$\Gamma_1 \vdash t_1 \supset t_2$	[Hypothesis]
2.	$\Gamma_2 \vdash t_2 \supset t_3$	[Hypothesis]
3.	$t_1 \vdash t_1$	[ASSUME]
4.	$\Gamma_1 \cup \{t_1\} \vdash t_2$	[MP 1,3]
5.	$\Gamma_1 \cup \Gamma_2 \cup \{t_1\} \vdash t_3$	[MP 2,4]
6.	$\Gamma_1 \cup \Gamma_2 \vdash t_1 \supset t_3$	[DISCH 5]

- **Instantiation of free variables**

```
INST term_term_list NUMBER
```

$$\frac{\Gamma \vdash t}{\Gamma \vdash t[t_1, \ldots, t_n/x_1, \ldots, x_n]}$$

`INST :(term # term) list -> thm -> thm`

The `term_term_list` is a list of pairs. Their second fields are variables which do not occur free in the assumption Γ. Their first fields are terms having the same type as the corresponding variables. They are substituted for the variables in the theorem referred to by the `NUMBER`.

1.	$\Gamma \vdash t$	[Hypothesis]
2.	$\Gamma \vdash \forall x_1 \ldots x_n.\, t[x_1, \ldots, x_n]$	[GENL[7] 1]
3.	$\Gamma \vdash t[t_1, \ldots, t_n/x_1, \ldots, x_n]$	[SPECL[8] 2]

- **Abstraction introduction on equality**

```
MKABS NUMBER
```

$$\frac{\Gamma \vdash \forall x.\, t_1 = t_2}{\Gamma \vdash (\lambda x.\, t_1) = (\lambda x.\, t_2)}$$

`MK_ABS : thm -> thm`

The theorem referred to by the `NUMBER` must be an equality inside a single universal quantification.

1.	$\Gamma \vdash \forall x.\, t_1 = t_2$	[Hypothesis]
2.	$\vdash (\lambda x.\, t_1)x' = t_1[x'/x]$	[BETA_CONV]
3.	$\Gamma \vdash t_1[x'/x] = t_2[x'/x]$	[SPEC 1]
4.	$\vdash (\lambda x.\, t_2)x' = t_2[x'/x]$	[BETA_CONV]
5.	$\vdash t_2[x'/x] = (\lambda x.\, t_2)x'$	[SYM 4]
6.	$\Gamma \vdash (\lambda x.\, t_1)x' = t_2[x'/x]$	[TRANS 2,3]
7.	$\Gamma \vdash (\lambda x.\, t_1)x' = (\lambda x.\, t_2)x'$	[TRANS 5,6]
8.	$\Gamma \vdash \forall x'.\, (\lambda x.\, t_1)x' = (\lambda x.\, t_2)x'$	[GEN 7]
9.	$\Gamma \vdash (\lambda x.\, t_1) = (\lambda x.\, t_2)$	[EXT 8]

[7] `GENL` is an iterative version of `GEN` that applies `GEN` repeatedly to a list of variables.
[8] `SPECL` is an iterative version of `SPEC` that applies `SPEC` repeatedly to a list of terms.

- **Equality of combinations**

`MKCOMB NUMBER NUMBER`

$$\frac{\Gamma_1 \vdash f = g \quad \Gamma_2 \vdash x = y}{\Gamma_1 \cup \Gamma_2 \vdash f\,x = g\,y}$$

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

Both theorems referred to by the `NUMBER`s must be equations. Both sides of the first theorem are functions whose domains are the type of x in the second theorem.

1.	$\Gamma_1 \vdash f = g$	[Hypothesis]
2.	$\Gamma_2 \vdash x = y$	[Hypothesis]
3.	$\vdash f\,x = f\,x$	[`REFL`]
4.	$\Gamma_1 \vdash f\,x = g\,x$	[`SUBST` 1,3]
5.	$\Gamma_1 \vdash f\,x = g\,y$	[`SUBST` 2,4]

- **Converting $\forall$ to $\exists$**

`MKEXISTS NUMBER`

$$\frac{\Gamma \vdash \forall x.\, t_1 = t_2}{\Gamma \vdash (\exists x.\, t_1) = (\exists x.\, t_2)}$$

`MK_EXISTS : thm -> thm`

The theorem referred to by the `NUMBER` must be an equality inside a single universal quantification.

1.	$\Gamma \vdash \forall x.\, t_1[x] = t_2[x]$	[Hypothesis]
2.	$\Gamma \vdash t_1[x'/x] = t_2[x'/x]$	[`SPEC` 1]
3a.	$\Gamma \vdash t_1[x'/x] \supset t_2[x'/x]$	[`EQ_IMP_RULE`(right) 2]
3b.	$\Gamma \vdash t_2[x'/x] \supset t_1[x'/x]$	[`EQ_IMP_RULE`(left) 2]
4.	$t_1[x'/x] \vdash t_1[x'/x]$	[`ASSUME`]
5.	$\Gamma \cup \{t_1[x'/x]\} \vdash t_2[x'/x]$	[`MP` 3a,4]
6.	$\Gamma \cup \{t_1[x'/x]\} \vdash \exists x.\, t_2[x]$	[`EXISTS` 5]
7.	$\exists x.\, t_1[x] \vdash \exists x.\, t_1[x]$	[`ASSUME`]
8.	$\Gamma \cup \{\exists x.\, t_1[x]\} \vdash \exists x.\, t_2[x]$	[`CHOOSE` 7,6]
9.	$\Gamma \vdash \exists x.\, t_1[x] \supset \exists x.\, t_2[x]$	[`DISCH` 8]
9.	$t_2[x'/x] \vdash t_2[x'/x]$	[`ASSUME`]
10.	$\Gamma \cup \{t_2[x'/x]\} \vdash t_1[x'/x]$	[`MP` 3b,9]
11.	$\Gamma \cup \{t_2[x'/x]\} \vdash \exists x.\, t_1[x]$	[`EXISTS` 10]
12.	$\exists x.\, t_2[x] \vdash \exists x.\, t_2[x]$	[`ASSUME`]
13.	$\Gamma \cup \{\exists x.\, t_2[x]\} \vdash \exists x.\, t_1[x]$	[`CHOOSE` 12,11]
14.	$\Gamma \vdash \exists x.\, t_2[x] \supset \exists x.\, t_1[x]$	[`DISCH` 13]
15.	$\Gamma \vdash \exists x.\, t_1[x] = \exists x.\, t_2[x]$	[`IMP_ANTISYM_RULE` 9,14]

- **¬-elimination**

```
NOTELIM NUMBER
```

$$\frac{\Gamma \vdash \neg t}{\Gamma \vdash t \supset \mathsf{F}}$$

`NOT_ELIM : thm -> thm` The theorem refered to by `NUMBER` must be of the form of the hypothesis.

1.	$\vdash \neg = \lambda b.\, b \supset \mathsf{F}$	[Definition of $\neg$]
2.	$\Gamma \vdash \neg t$	[Hypothesis]
3.	$\Gamma \vdash (\lambda b.\, b \supset \mathsf{F})t$	[`SUBST` 1,2]
4.	$\vdash (\lambda b.\, b \supset \mathsf{F})t = t \supset \mathsf{F}$	[`BETA_CONV`]
5.	$\Gamma \vdash t \supset \mathsf{F}$	[`SUBST` 4,3]

- **¬-introduction**

```
NOTINTRO NUMBER
```

$$\frac{\Gamma \vdash t \supset \mathsf{F}}{\Gamma \vdash \neg t}$$

`NOT_INTRO : thm -> thm` The theorem refered to by `NUMBER` must be an implication whose conlusion is the constant F.

1.	$\vdash \neg = \lambda b.\, b \supset \mathsf{F}$	[Definition of $\neg$]
2.	$\Gamma \vdash t \supset \mathsf{F}$	[Hypothesis]
3.	$\vdash \neg t = \neg t$	[`REFL`]
4.	$\vdash \neg t = (\lambda b.\, b \supset \mathsf{F})t$	[`SUBST` 1,3]
5.	$\vdash (\lambda b.\, b \supset \mathsf{F})t = t \supset \mathsf{F}$	[`BETA_CONV`]
6.	$\vdash \neg t = t \supset \mathsf{F}$	[`SUBST` 5,4]
7.	$\vdash t \supset \mathsf{F} = \neg t$	[`SUBST` 6,3]
8.	$\Gamma \vdash \neg t$	[`SUBST` 7,2]

- **Specialization (∀-elimination)**

```
SPEC term NUMBER
```

$$\frac{\Gamma \vdash \forall x.\, t}{\Gamma \vdash t[t'/x]}$$

`SPEC : term -> thm -> thm`

The theorem referred to by the `NUMBER` must be universally quantified. The `term` has the same type as the bound variable x of the theorem.

1.	$\vdash \forall = (\lambda P.\, P = (\lambda x.\, \mathsf{T}))$	[Definition of $\forall$[9]]
2.	$\Gamma \vdash \forall(\lambda x.\, t)$	[Hypothesis]
3.	$\Gamma \vdash (\lambda P.\, P = (\lambda x.\, \mathsf{T}))(\lambda x.\, t)$	[SUBST 1,2]
4.	$\vdash (\lambda P.\, P = (\lambda x.\, \mathsf{T}))(\lambda x.\, t) = ((\lambda x.\, t) = (\lambda x.\, \mathsf{T}))$	[BETA_CONV]
5.	$\Gamma \vdash (\lambda x.\, t) = (\lambda x.\, \mathsf{T})$	[SUBST 4,3]
6.	$\vdash (\lambda x.\, t)\ t' = (\lambda x.\, t)\ t'$	[REFL]
7.	$\Gamma \vdash (\lambda x.\, t)\ t' = (\lambda x.\, \mathsf{T})\ t'$	[SUBST 5,6]
8.	$\vdash (\lambda x.\, t)\ t' = t[t'/x]$	[BETA_CONV]
9.	$\vdash (\lambda x.\, t)\ t' = (\lambda x.\, t)\ t'$	[REFL]
10.	$\vdash t[t'/x] = (\lambda x.\, t)\ t'$	[SUBST 8,9]
11.	$\Gamma \vdash t[t'/x] = (\lambda x.\, \mathsf{T})\ t'$	[SUBST 10,7]
12.	$\vdash (\lambda x.\, \mathsf{T})\ t' = \mathsf{T}$	[BETA_CONV]
13.	$\Gamma \vdash t[t'/x] = \mathsf{T}$	[SUBST 12,11]
14.	$\vdash t[t'/x] = t[t'/x]$	[REFL]
15.	$\Gamma \vdash \mathsf{T} = t[t'/x]$	[SUBST 13,14]
16.	$\vdash \mathsf{T}$	[Theorem TRUTH]
17.	$\Gamma \vdash t[t'/x]$	[SUBST 15,16]

- **Substitution (for all instances)**

  `SUBS NUMBER_list NUMBER`

  $$\frac{\Gamma_1 \vdash x_1 = t_1 \ldots \Gamma_n \vdash x_n = t_n \quad \Gamma \vdash t}{\Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma \vdash t[t_1, \ldots, t_n/x_1, \ldots x_n]}$$

  `SUBS : thm list -> thm -> thm`

  This is a generalized version of the primitive rule `SUBST`. It replaces all occurrences of the variables x_i in t by the corresponding term t_i.

- **Substitution (for some instances)**

  `SUBSOCCS NUMBER_list_NUMBER_list NUMBER`

  $$\frac{\Gamma_1 \vdash x_1 = t_1 \ldots \Gamma_n \vdash x_n = t_n \quad \Gamma \vdash t}{\Gamma_1 \cup \ldots \cup \Gamma_n \cup \Gamma \vdash t[t_1, \ldots, t_n/x_1, \ldots x_n]}$$

  `SUBS_OCCS : ((num)list # thm)list -> thm -> thm`

  This is another version of the primitive rule `SUBST`. It is more seletive than `SUBS`. The first argument to this rule is a list of pairs whose general form is

  $$([h_{i1}; \ldots; h_{im}], \Gamma_i \vdash x_i = t_i)$$

  It replaces only the occurrences of x_i in t specified by the numbers x_{ij}. The occurrences are numbered from left to right starting from 1. In addition to the similar checks as `SUBS`, checking of whether only the specific occurrences are replaced needs to be carried out.

[9] with appropriate type instantiation.

- **Substitution (conversion)**

```
SUBSTCONV NUMBER_term_list term term
```

$$\frac{\Gamma_1 \vdash t_1 = t'_1 \quad \cdots \quad \Gamma_n \vdash t_n = t'_n}{\Gamma_1 \cup \cdots \cup \Gamma_n \vdash t[t_1, \ldots, t_n/x_1, \ldots, x_n] = t[t'_1, \ldots, t'_n/x_1, \ldots, x_n]}$$

`SUBST_CONV : (thm # term)list -> term -> term -> thm`

This is a conversion performing substitution in a term similar to the primitive rule `SUBST`. The elements of the first argument to this conversion have the following form:

$$(\Gamma_i \vdash t_i = t'_i, "x_i")$$

The first `term` argument, $t[x_1, \ldots, x_n]$, contains the variables x_i marking the places where substitution is required. The second `term` argument should match the first `term` with occurrences of x_i replaced by corresponding t_i which is the left-hand side of the corresponding theorem in the list. This term is the left-hand side of the resulting theorem. The right-hand side is obtained by replacing the occurrences of x_i in t by corresponding t'_i. The checking of this justification can be done in two stages: the first ensures that the left-hand side of the resulting theorem is a substitution as specified by the input terms; the second checks for the correct substituion on the right-hand side of the resulting theorem.

- **Symmetry of equality**

```
SYM NUMBER
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}$$

`SYM : thm -> thm` The theorem referred to by `NUMBER` must be an equation.

1.	$\Gamma \vdash t_1 = t_2$	[Hypothesis]
2.	$\vdash t_1 = t_1$	[`REFL`]
3.	$\Gamma \vdash t_2 = t_1$	[`SUBST` 1,2]

- **Transitivity of equality**

```
TRANS NUMBER NUMBER
```

$$\frac{\Gamma_1 \vdash t_1 = t_2 \quad \Gamma_2 \vdash t_2 = t_3}{\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3}$$

`TRANS : thm -> thm -> thm`

Both theorems referred to by the `NUMBER`s must be equations. The right-hand side of the first theorem is the same as the left-hand side of the second.

1.	$\Gamma_1 \vdash t_1 = t_2$	[Hypothesis]
2.	$\Gamma_2 \vdash t_2 = t_3$	[Hypothesis]
3.	$\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3$	[`SUBST` 2,1]

5 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 except that the extra step of recording the proofs in a file is carried out only if the feature is enabled. This user interface is implemented as a system library. It provides a uniform way of working with the system whether or not the proof is being recorded. The typical use of the proof recording 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-run once more and is saved in a proof file.

To help explain the user interface, a simple interactive session with HOL in which proof recording is enabled is shown in Figure 3.

After loading the library, a new theory named `THY` is first created 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 `new_proof_file`.

To record forward proofs, one needs to use the library functions `begin_proof` and `end_proof` to enclose the proof procedures. Figure 4 shows how to record a simple forward proof. 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. The return of a null string indicates that no proof is in progress.

When mixing forward and tactical proof techniquess, care should be taken to make the direct calls to inference rules local to the call to the tactical functions. For exmple, in Figure 5, the simple forward proof to derive `lemma` is local to the

```
#load_library`record_proof`;;
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
```

Figure 3: Recoding a simple tactical proof

call to `prove_thm`. If the proof of `lemma` is placed outside the call to `prove_thm`, it will not be recorded.

While one is developing the proof, it is not necessary for 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 using 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.

```
#new_proof_file`ap_term.prf`;;
() : 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%422" : 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

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

Figure 4: Recoding a simple forward proof

```
prove_thm(`SIMPLE_THM`, "<goal here>",
 let lemma = <applying some inference rules here>
 in
   <tactic here>
 );;
```

Figure 5: Mixinf forward and tactical proof techniques

6 The Developer's Interface

Below the user interface described in the previous section, there are a small set of ML functions forming a lower level interface to the proof recorder for developers. 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 a basic inference rule is recorded in an internal buffer. Each inference is represented by an 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 for 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 is supposed to derive. It may be null. If so, 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, one 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.

7 The Implementation

HOL88 has been modified to incorporate the proof recording feature. This feature will be available from version 2.02. The implementation can be divided into two parts: the primitive functions and the `record_proof` library.

The primitive functions are the kernel of the proof recorder. They carry out the first stage of the process of recording proofs. They should be very efficient since the raw records are generated as every basic inference rule is applied. Therefore, they are incorporated into the system itself.

The `record_proof` library consists of functions for the tasks in Stage 2 and 3 described in Section 6 and the user interface functions described in Section 5.

The user functions can be divided into two groups: management functions and proof functions. The management functions are

`new_proof_file`, `close_proof_file`, `current_proof_file`, `begin_proof`, `end_proof` and `current_proof`.

The library has two definitions for each of these functions. One of the definitions is a dummy which is used when the `disable` part of the library is loaded. These dummy definitions only declare the names of the function, so the system will recognise them. Therefore the same source file without any change can be used whether or not the proofs are being recorded. The proper definition is used when the whole library is loaded. In this case, the proof recorder is enabled. These functions manage the proof file.

The proof functions in the library are

`TAC_PROOF`, `PROVE`, `prove` and `prove_thm`.

They perform the same tactical proof as their counterpart in the system and with the extra function of automatically recording the proof. The new definitions take effect when the whole library is loaded. When the `disable` part of the library is loaded, the original version of these functions in the system is in scope.

The remainder of this section describes the internal data structures and the process of each stage of recording proof and generating a proof file in more detail.

7.1 Data Structures

The internal buffer `steplist` for storing the raw records of inference steps is a list whose elements are of type `step`. Each step represents an application of a basic inference rule. After the proof recorder is enabled, information is cumulated in this buffer. It can only be accessed by the user indirectly by calling the function `get_steps`.

Information contained in each step consists of the actual arguments passed to the inference rule. For example, the primitive inference rule `ASSUME` requires a term `t`, so the step corresponding to the application of this rule is `(AssumeStep t)`. The complete definition of the type `step` can be found in Appendix B.

In the second stage, the step list is processed and a list of lines, i.e., a list whose elements are of type `line`, is created. Each step in the step list will generate a line in the line list. Each line represents a single inference. The definition of the type `line` is:

```
type line = Line of (int # thm # justification);;
```

The `int` field is the line number. The second field is a theorem resulting from applying the inference rule. The last field is the justification of the inference. It contains the name of the inference rule and its actual arguments. If an argument is a theorem, it is replaced by a line number. This refers to the line in which the theorem appears as the result of the inference. Theorems appearing as actual arguments to the inference rule may not be found in any previous lines. They are treated as the hypotheses of the proof. A line with a negative line number is generated for each of these theorems. The definition of the type `justification` can be found in Appendix B.

7.2 Internal States

Two internal variables, `record_proof_flag` and `suspended`, govern the recording of proof. They can only be accessed indirectly by the user using the following functions:

- `record_proof` This function sets/clears the `record_proof_flag` to enable/disable the recording.
- `is_recording_proof` This function returns the current setting of the internal variable `record_proof_flag`.
- `suspend_recording` This function suspends the proof recording temporarily. It clears `record_proof_flag` and sets `suspended`.
- `resume_recording` This function resumes the proof recording. It sets `record_proof_flag` and clears `suspended`.

The temporary suspension of recording allows an auxiliary proof to be performed in the midst of a main proof without being recorded. The internal states of the proof recording mechanism are illustrated in Figure 6. The states are labelled by the triple `(record_proof_flag, suspended, steplist)`. Three dots (...) in the square brackets indicate that `steplist` is not null. The transaction between states is caused by calling one of the functions above. The transaction labelled 'Record' is made by performing an inference step after the proof recording is enabled.

7.3 Recording Inference Steps

The actual recording of basic inference steps is done by the function `RecordStep`. This function takes a single argument of type `step`. It checks the value of the internal variable `record_proof_flag`. If it is `true`, the step is added to the internal buffer `steplist`.

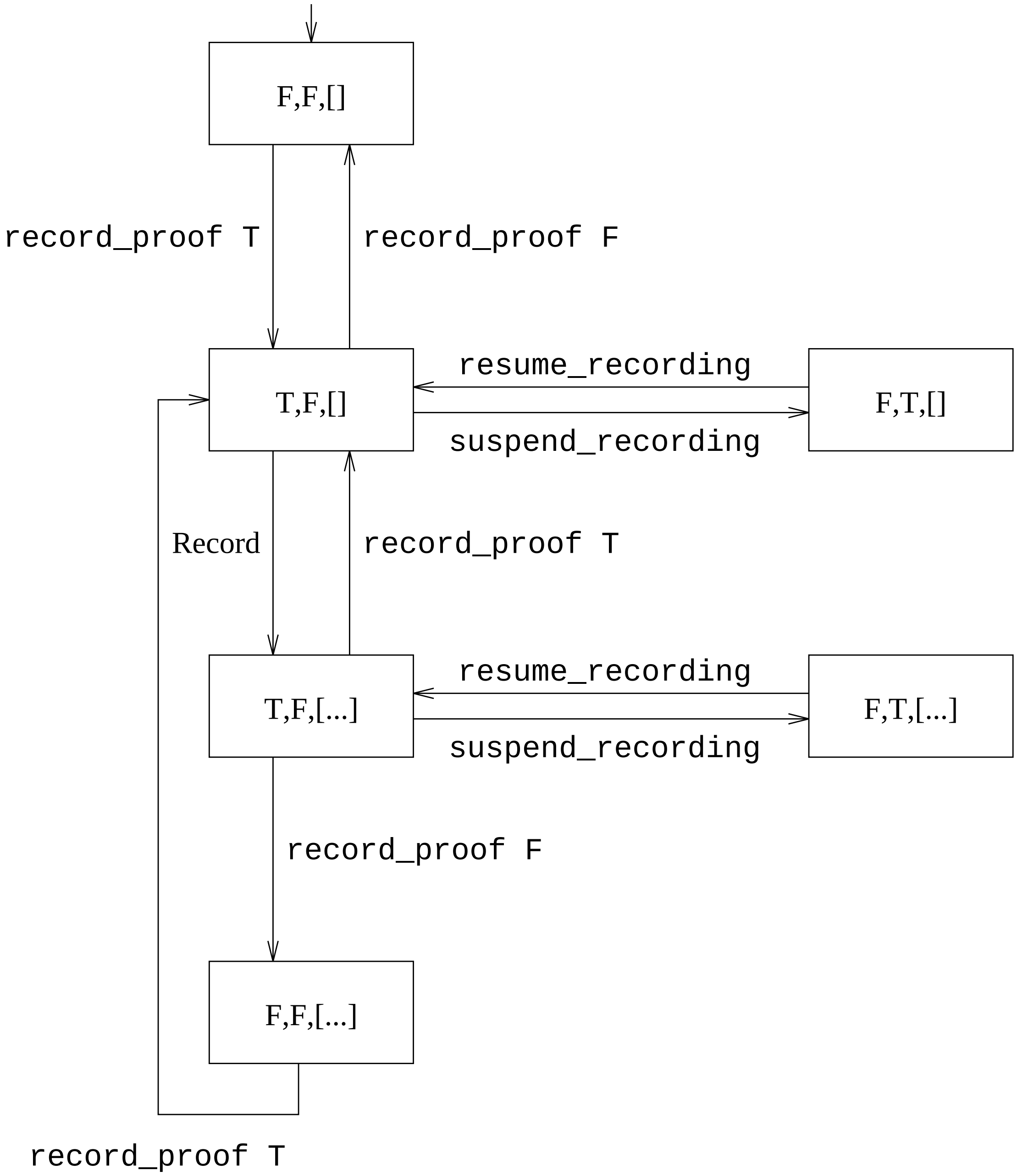

Figure 6: State diagram of proof recorder

As mentioned above, every application of a basic inference rule is recorded. A call to the function `RecordStep` with the appropriate argument is added in every function implementing the basic inference rules. There are in total fifty-one such functions in the HOL system. In this way, all the information about the application of inference rules is recorded dynamically. A list of all functions implementing basic inference rules can be found in Appendix C. Below is the step list of the proof shown in Figure 4. It is in reverse order, i.e., the head of the list is the last step of the proof. There are two steps in this proof: the first is a `ReflStep` which is an application of the primitive rule `REFL`, and the second is the primitive rule `SUBST`.

```
[SubstStep([(|- m + n = n + m, "GEN%VAR%422")],
           "SUC(m + n) = SUC GEN%VAR%422",
           |- SUC(m + n) = SUC(m + n));
 ReflStep"SUC(m + n)"]
```

The primitive rule `REFL` takes a term as its argument and returns an equational theorem. The primitive rule `SUBST` performs a substitution on the theorem `|- SUC(m + n) = SUC(m + n)`.

7.4 Generating a List of Lines

When a proof is complete, the list of steps is processed and a list of lines is generated. This is done by the function `MakeProof`. It converts each step into a line. The justification field of the lines contains the same information as in the steps. However, the theorems appearing in the steps are replaced by their line numbers. All line numbers are generated automatically by the function `MakeProof`.

Some of the theorems are not the result of any previous lines. These are the initial theorems of the current proof, i.e., the hypotheses. They should have been proved in other proofs. These are added to the line list as hypotheses. The hypothesis lines are given negative line numbers.

Below is the list of lines for the proof shown in Figure 4. Notice that the initial theorem is added as the hypothesis, and in Line 2, the last field of the justification is the line number 1 which refers to the theorem in Line 1. The corresponding step shown in the previous subsection contains the theorem explicitly.

```
[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%422")],
            "SUC(m + n) = SUC GEN%VAR%422", 1))]
```

Table 1: Benchmark of recording the multiplier proof

FILE	**No. of THMS**	**DISABLED**		**ENABLED**		SIZE	
		RUN	GC	RUN	GC	Raw	Compr'd
`mk_NEXT`	2972	–	–	116.0	12.8	2693853	62202
`MULT_FUN_CURRY`	670	–	–	83.3	7.1	1553642	29103
`MULT_FUN`	6943	–	–	488.1	120.0	8101358	188201
`HOL_MULT`	3946	–	–	1675.5	250.1	31200001	447036
TOTAL	14531	65.3	11.8	2412.9	390.0	43627841	726542

7.5 Outputting to Proof File

The top level functions for outputting proofs are `write_proof_to` which creates a new proof file and `write_proof_add_to` which appends to an existing proof file. They write a proof given as a list of proof lines into an output file. They are implemented using the output functions `write_env`, `write_thm_list` and `write_line`.

The function `write_env` takes a single argument, an output port. It works output the current proof environment, and outputs it to the port. The functions `write_thm_list` and `write_line` both take an output port as their first argument. Their second arguments are a list of theorems and a proof line, respectively. All these three functions call lower level functions to convert abstract representation of proof objects, such as proof lines, theorems, terms, and so on, into text string. These lower level output functions are organised in recursive descend fashion.

8 Benchmarking

A proof of correctness of a simple multiplier described in [2] is often used as a HOL benchmark. This is a small to medium size proof which generates 14500 intermediate theorems. We recorded and saved this proof in proof files. The HOL source of this proof is divided into four files, each contains a number of sub-proofs. Six proof files are generated. Each contains the sub-proofs of a separate theory file. The complete proof was performed twice: the first time without recording, the second time with recording enabled. The tests were run on a SUN Sparc 10 Server. The results including the run time, garbage connection and proof file sizes are listed in Table 1. The time is measured in seconds, and the file sizes are in bytes.

As the figures in the table show, the time (2412.9 seconds $\simeq$ 40 minutes) required to record and generate the proof files is considerable longer than just to perform the proof, but it is not excessive. Most of the extra time is spent in converting the internal presentation to the textual format and actually writing the disk file. This extra time is acceptable since the proof files will only be generated after the proof is completed satisfactorily (probably once).

The sizes of the proof files are also listed in Table 1. They are very large (43 Mbytes in total) because every intermediate theorem has to be saved. The size per theorem is comparable to the theory files. However, the proof files are intended for automatic tools not for human readers, and they can be stored in compressed form. The size of the compressed files is much smaller. It amounts to less than 2% of the raw size. The compress program used is `gzip`, a public domain program which implements the well-known Lempel-Ziv coding (LZ77) algorithm [6]. As the compression is done automatically, this does not pose too much burden to the user.

It should also be emphasized that the user is *not* restricted to use only basic inference rules described above. In fact, all higherer level derived inference rules, tactics and tacticals can be used in a proof. The proof will be recorded properly as long as the user adheres to the simple rules described in Section 5.

Acknowledgement

The idea of recording inference steps and generating proof lines has been suggested by many people including Malcolm Newey and Keith Hanna. Dr. M. J. C. Gordon implemented a prototype of the recording functions on which the implementation described in this paper is based. He and Dr. P.Curzon read an early draft of this report and gave many valuable comments. This research is supported by a grant from SERC (No. GR/G223654).

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] M. J. C. Gordon. LCF_LSM, a system for specifying and verifying hardware. Technical Report 41, University of Cambridge Computer Laborartory, 1983.

[3] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL—a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[4] Ministry of Defence. *Requirements for the procurment of safety-critical software in defence equipment.* Interim Standard 00-55 (Part 2), April 1991.

[5] J. von Wright. Representing higher-order logic proofs in HOL. Draft report, May 1993.

[6] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23(3):337–343, May 1977.

A Listing of a proof file

This appendix lists the proof file generated for the example proof in Figure 4.

```
(VERSION PRF FORMAT 1.0 EXTENDED)
(ENV HOL[][])

(PROOF proof1
[]
[
(LINE -1(Hypothesis)
(THM [](A (A (C = (o fun[(c num)(o fun[(c num)(c bool)])]))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))
(V m(c num)))(V n(c num))))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))
(V n(c num)))(V m(c num)))))
)

(LINE 1(Refl (A (C SUC(o fun[(c num)(c num)]))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))
(V m(c num)))(V n(c num)))))
(THM [](A (A (C = (o fun[(c num)(o fun[(c num)(c bool)])]))
(A (C SUC(o fun[(c num)(c num)]))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))(V m(c num)))
(V n(c num)))))(A (C SUC(o fun[(c num)(c num)]))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))(V m(c num)))
(V n(c num))))))
)

(LINE 2(Subst [{-1(V GEN%VAR%422(c num))}]
(A (A (C = (o fun[(c num)(o fun[(c num)(c bool)])]))
(A (C SUC(o fun[(c num)(c num)]))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))
(V m(c num)))(V n(c num)))))(A (C SUC(o fun[(c num)(c num)]))
(V GEN%VAR%422(c num))))1)
(THM [](A (A (C = (o fun[(c num)(o fun[(c num)(c bool)])]))
(A (C SUC(o fun[(c num)(c num)]))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))
```

```
(V m(c num)))(V n(c num)))))(A (C SUC(o fun[(c num)(c num)]))
(A (A (C + (o fun[(c num)(o fun[(c num)(c num)])]))
(V n(c num)))(V m(c num))))))
)

])
```

B Types defined for recording proofs

```
type step = AssumeStep of term
 | ReflStep of term
 | SubstStep of (thm#term)list # term # thm
 | BetaConvStep of term
 | AbsStep of term # thm
 | InstTypeStep of (type#type)list # thm
 | DischStep of term # thm
 | MpStep of thm # thm
 | MkCombStep of thm # thm
 | MkAbsStep of thm
 | AlphaStep of term # term
 | AddAssumStep of term # thm
 | SymStep of thm
 | TransStep of thm # thm
 | ImpTransStep of thm # thm
 | ApTermStep of term # thm
 | ApThmStep of thm # term
 | EqMpStep of thm # thm
 | EqImpRuleStep of thm
 | SpecStep of term # thm
 | EqtIntroStep of thm
 | GenStep of term # thm
 | EtaConvStep of term
 | ExtStep of thm
 | ExistsStep of (term # term) # thm
 | ChooseStep of (term # thm) # thm
 | ImpAntisymRuleStep of thm # thm
 | MkExistsStep of thm
 | SubsStep of thm list # thm
 | SubsOccsStep of (int list # thm) list # thm
 | SubstConvStep of (thm # term) list # term # term
 | ConjStep of thm # thm
 | Conjunct1Step of thm
```

```
 | Conjunct2Step of thm
 | Disj1Step of thm # term
 | Disj2Step of term # thm
 | DisjCasesStep of thm # thm # thm
 | NotIntroStep of thm
 | NotElimStep of thm
 | ContrStep of term # thm
 | CcontrStep of term # thm
 | InstStep of (term # term) list # thm
 | StoreDefinitionStep of string # term
 | DefinitionStep of string # string
 | DefExistsRuleStep of term
 | NewAxiomStep of string # term
 | AxiomStep of string # string
 | TheoremStep of string # string
 | NewConstantStep of string # type
 | NewTypeStep of int # string
 | NumConvStep of term;;

type justification =
   Hypothesis
 | Assume of term
 | Refl of term
 | Subst of (int#term)list # term # int
 | BetaConv of term
 | Abs of term # int
 | InstType of (type#type)list # int
 | Disch of term # int
 | Mp of int # int
 | MkComb of int # int
 | MkAbs of int
 | Alpha of term # term
 | AddAssum of term # int
 | Sym of int
 | Trans of int # int
 | ImpTrans of int # int
 | ApTerm of term # int
 | ApThm of int # term
 | EqMp of int # int
 | EqImpRuleR of int
 | EqImpRuleL of int
 | Spec of term # int
```

```
 | EqtIntro of int
 | Gen of term # int
 | EtaConv of term
 | Ext of int
 | Exists of (term # term) # int
 | Choose of (term # int) # int
 | ImpAntisymRule of int # int
 | MkExists of int
 | Subs of int list # int
 | SubsOccs of (int list # int) list # int
 | SubstConv of (int # term) list # term # term
 | Conj of int # int
 | Conjunct1 of int
 | Conjunct2 of int
 | Disj1 of int # term
 | Disj2 of term # int
 | DisjCases of int # int # int
 | NotIntro of int
 | NotElim of int
 | Contr of term # int
 | Ccontr of term # int
 | Inst of (term # term) list # int
 | StoreDefinition of string # term
 | Definition of string # string
 | DefExistsRule of term
 | NewAxiom of string # term
 | Axiom of string # string
 | Theorem of string # string
 | NewConstant of string # type
 | NewType of int # string
 | NumConv of term;;

type line = Line of (int # thm # justification);;
```

C Basic inference rules

This section lists all basic inference rules in the HOL system. The functions implementing these rules are modified to incorporate the feature of recording proof. They are grouped by the file where they are defined.

hol-rule.ml

```
ASSUME      REFL      SUBST    BETA_CONV
ABS         INST_TYPE DISCH    MP
```

drul.ml

```
MK_COMB     MK_ABS    ALPHA
```

hol-drule.ml

```
ADD_ASSUM  SYM        TRANS      IMP_TRANS
AP_TERM    AP_THM     EQ_MP      EQ_IMP_RULE
SPEC       EQT_INTRO  GEN        ETA_CONV
EXT        EXISTS     CHOOSE     IMP_ANTISYM_RULE
MK_EXISTS  SUBS       SUBS_OCCS  SUBST_CONV
CONJ       CONJUNCT1  CONJUNCT2  DISJ1
DISJ2      DISJ_CASES NOT_INTRO  NOT_ELIM
CONTR      CCONTR     INST
```

hol-syn.ml

```
new_axiom  axiom    definition    store_definition
new_type   theorem  new_constant  DEF_EXISTS_RULE
```

numconv.ml

```
num_CONV
```

D Standard environments

This appendix lists the three standard envirnments. The `MIN` environment is the smallest. Every environment is a superset of all those listed before it.

D.1 MIN

```
(ENV MIN [{bool 0}  {ind 0}]
[{==> (":bool -> (bool -> bool)")}
 {= (":* -> (* -> bool)")}
 {@ (":(* -> bool) -> *")}]
```

D.2 LOG

```
(ENV LOG []
[{! ":(* -> bool) -> bool"}
 {? ":(* -> bool) -> bool"}
 {T ":bool"}
 {F ":bool"}
 {~ ":bool -> bool"}
 {/\ ":bool -> (bool -> bool)"}
 {\/ ":bool -> (bool -> bool)"}
 {ONE_ONE ":(* -> **) -> bool"}
 {ONTO ":(* -> **) -> bool"}
 {TYPE_DEFINITION ":(* -> bool) -> ((** -> *) -> bool)"}
]
```

D.3 HOL

```
(ENV HOL [{fun 2} {prod 2} {num 0} {list 1}
          {tree 0} {ltree 1} {sum 2} {one 0}]
[{* ":num -> (num -> num)"}
 {+ ":num -> (num -> num)"}
 {, ":* -> (** -> * # **)"}
 {- ":num -> (num -> num)"}
 {0 ":num"}
 {<= ":num -> (num -> bool)"}
 {< ":num -> (num -> bool)"}
 {>= ":num -> (num -> bool)"}
 {> ":num -> (num -> bool)"}
 {?! ":(* -> bool) -> bool"}
 {ABS_list ":(num -> *) # num -> (*)list"}
 {ABS_ltree ":tree # (*)list -> (*)ltree"}
 {ABS_num ":ind -> num"}
 {ABS_sum ":(bool -> (* -> (** -> bool))) -> * + **"}
 {ABS_tree ":num -> tree"}
 {APPEND ":(*)list -> ((*)list -> (*)list)"}
 {AP ":(* -> **)list -> ((*)list -> (**)list)"}
 {ARB ":*"}
 {BINDERS ":* -> bool"}
 {COND ":bool -> (* -> (* -> *))"}
 {CONS ":* -> ((*)list -> (*)list)"}
 {CURRY ":(* # ** -> ***) -> (* -> (** -> ***))"}
 {DIV ":num -> (num -> num)"}
 {EL ":num -> ((*)list -> *)"}
```

```
{EVEN ":num -> bool"}
{EVERY ":(* -> bool) -> ((*)list -> bool)"}
{EXP ":num -> (num -> num)"}
{FACT ":num -> num"}
{FLAT ":((*)list)list -> (*)list"}
{FST ":* # ** -> *"}
{HD ":(*)list -> *"}
{HOL_DEFINITION ":bool -> bool"}
{HT ":tree -> num"}
{INL ":* -> * + **"}
{INR ":** -> * + **"}
{ISL ":* + ** -> bool"}
{ISR ":* + ** -> bool"}
{IS_ASSUMPTION_OF ":bool -> (bool -> bool)"}
{IS_NUM_REP ":ind -> bool"}
{IS_PAIR ":(* -> (** -> bool)) -> bool"}
{IS_SUM_REP ":(bool -> (* -> (** -> bool))) -> bool"}
{IS_list_REP ":(num -> *) # num -> bool"}
{I ":* -> *"}
{Is_ltree ":tree # (*)list -> bool"}
{Is_tree_REP ":num -> bool"}
{K ":* -> (** -> *)"}
{LENGTH ":(*)list -> num"}
{LET ":(* -> **) -> (* -> **)"}
{MAP2 ":(* -> (** -> ***)) -> ((*)list ->((**)list->(***)list))"}
{MAP ":(* -> **) -> ((*)list -> (**)list)"}
{MK_PAIR ":* -> (** -> (* -> (** -> bool)))"}
{MOD ":num -> (num -> num)"}
{NIL ":(*)list"}
{NULL ":(*)list -> bool"}
{Node ":* -> (((*)ltree)list -> (*)ltree)"}
{ODD ":num -> bool"}
{OUTL ":* + ** -> *"}
{OUTR ":* + ** -> **"}
{PART ":(num)list -> ((*)list -> ((*)list)list)"}
{PRE ":num -> num"}
{PRIM_REC_FUN ":* -> ((* -> (num -> *)) -> (num -> (num -> *)))"}
{PRIM_REC ":* -> ((* -> (num -> *)) -> (num -> *))"}
{REP_list ":(*)list -> (num -> *) # num"}
{REP_ltree ":(*)ltree -> tree # (*)list"}
{REP_num ":num -> ind"}
{REP_prod ":* # ** -> (* -> (** -> bool))"}
```

```
 {REP_sum ":* + ** -> (bool -> (* -> (** -> bool)))"}
 {REP_tree ":tree -> num"}
 {RES_ABSTRACT ":(* -> bool) -> ((* -> **) -> (* -> **))"}
 {RES_EXISTS ":(* -> bool) -> ((* -> bool) -> bool)"}
 {RES_FORALL ":(* -> bool) -> ((* -> bool) -> bool)"}
 {RES_SELECT ":(* -> bool) -> ((* -> bool) -> *)"}
 {SIMP_REC_FUN ":* -> ((* -> *) -> (num -> (num -> *)))"}
 {SIMP_REC_REL ":(num -> *) -> (* -> ((* -> *)->(num -> bool)))"}
 {SIMP_REC ":* -> ((* -> *) -> (num -> *))"}
 {SND ":* # ** -> **"}
 {SPLIT ":num -> ((*)list -> (*)list # (*)list)"}
 {SUC_REP ":ind -> ind"}
 {SUC ":num -> num"}
 {SUM ":(num)list -> num"}
 {S ":(* -> (** -> ***)) -> ((* -> **) -> (* -> ***))"}
 {Size ":tree -> num"}
 {TL ":(*)list -> (*)list"}
 {TRP ":(* -> (((*)ltree)list -> bool)) -> ((*)ltree -> bool)"}
 {UNCURRY ":(* -> (** -> ***)) -> (* # ** -> ***)"}
 {ZERO_REP ":ind"}
 {bht ":num -> (tree -> bool)"}
 {dest_node ":tree -> (tree)list"}
 {node_REP ":(num)list -> num"}
 {node ":(tree)list -> tree"}
 {o ":(** -> ***) -> ((* -> **) -> (* -> ***))"}
 {one ":one"}
 {trf ":num -> (((**)list -> **) -> (tree -> **))"}
]
```