Recording and Checking HOL Proofs

Wai Wong*

Department of Computing Studies Hong Kong Baptist University Kowloon Tong, Hong Kong

Abstract

Formal proofs generated by mechanised theorem proving systems may consist of a large number of inferences. As these theorem proving systems are usually very complex, it is extremely difficult if not impossible to formally verify them. This calls for an independent means of ensuring the consistency of mechanically generated proofs. This paper describes a method of recording HOL proofs in terms of a sequence of applications of inference rules. The recorded proofs can then be checked by an independent proof checker. Also described in this paper is an efficient proof checker which is able to check a practical proof consisting of thousands of inference steps.

1 Introduction

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 using *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 proofs. The result of this process is usually a set of theorems which can be stored in a disk file and used in subsequent proofs. HOL is one of the most popular theorem proving environments. The users interact with the system by writing and evaluating ML programs. They instruct the system how to perform 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 safety-critical applications, computer systems are used to implement some of the highest risk category functions. The design of such a system is often formally verified. The verification usually produces a large proof consisting of tens of thousands, even up to several millions, of inferences. [Won93a] describes a proof of correctness of an ALU consisting of a quarter of a million inference steps. In such situations, it is desirable to check the consistency of the sequence

^{*} The work described in this paper was carried out by the author while he was in the University of Cambridge Computer Laboratory supported by a grant from SERC (No. GR/G223654)

of inferences with an independent checker. The reasons for requiring independent checking are:

- the mechanically generated formal proofs are usually very long;
- the theorem proving systems are usually very complex so that it is extremely difficult (if it is not impossible) to verify their correctness;
- the programs that a user develops while doing the proof are very often too complicated and do not have a simple mapping to the sequence of inferences performed by the system.

An independent proof checker can be much simpler than the theorem prover so that it is possible to be verified formally. The U. K. Defence standard 00-55 calls for such an independent proof checker when the 'highest degree of assurance in the design' is required [oD91].

The necessary condition for a HOL proof to be checked by an independent checker is to have the proof expressed as a sequence of inferences. To achieve this, a method of recording HOL proofs has been developed and implemented. This comprises a proof file format, a small modification to the HOL core system and a library of user functions for managing the proof recorder. The approach of adding the proof recording feature to the HOL system is discussed in Section 3. This is followed by a section describing briefly the proof file format and a section on the proof recording library. Section 6 discusses some issues of implementing a proof checker. An efficient proof checker has been implemented and will be described in Section 7 and 8.

2 Proofs in HOL

A detailed description of the HOL logic and its proof theory, together with several tutorial examples of using the HOL system can be found in [GM93]. For the benefit of the readers who are not familiar with HOL, an overview of the HOL deductive system and the theorem-proving infrastructure is given in this section.

A proof is a finite sequence of inferences Δ in a deductive system. Each inference in Δ is a pair $(L, (\Gamma, t))$ where L is a (possibly empty) list of sequents $(\Gamma_1, t_1) \dots (\Gamma_n, t_n)$ and (Γ, t) is known as a sequent. The first part Γ of a sequent is a (possibly empty) set of terms known as the **assumptions**. The second part t is a single term known as the **conclusion**. A particular deductive system is usually specified by a set of schematic rules of inference (also known as primitive inference rules) written in the following form

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

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 satisfy one of the inference rules of the deductive system. There are eight primitive inference rules in HOL. They are described in detail in Section 16.3.1 of [GM93]. In HOL, rules of inference are implemented by ML functions.

More complex inference can be created by combining the primitive inference rules. For example, the rule of symmetry of equality ([SYM]) can be specified as

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}.$$
(2)

This can be derived using the primitive rules as follows:

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

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

Derived rules are also represented by ML functions. They are implemented in terms of the primitive rules. A theorem prover in which all proofs are fully expanded into primitive inferences is known as *fully-expansive*[Bou92]. 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 and frequently used derived rules, such as **SYM**, are not fully expanded, but are implemented directly in ML.

These rules, including the primitive rules and derived rules that are implemented directly in ML, will be referred to as *basic inference rules* or simply *basic rules* 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 far 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 subgoals are resolved. In such a proof, the user does not call the inference rules directly. However, a correct sequence of inferences is calculated and performed by the system behind the scenes automatically to derive the theorem.

A proof in HOL as described above is carried out within an environment which 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 (\mathbf{C}, σ) where **C** is the name and σ is its type and all the type constants that occur in the σ s must be in Ω . This provides a context against which the well-typedness of terms can be checked.

A formal theory of the HOL proof system has been developed by J. von Wright [vW94]. The notion of types, terms, inferences and proofs are captured in his theory. This provides a formal base for developing a proof checker.

3 Recording HOL Proofs

In the HOL systems, there exists no object as a HOL proof once a theorem is derived whatever the proof style used in the derivation. In order to check the consistency of a HOL proof by an independent system, one needs to preserve the proof.

Since the HOL system is a fully-expansive theorem prover, it is possible to record the sequence of inference rules together with the hypotheses and the conclusion in the derivation of a theorem. The recording can be done at the time the system performs each inference. Thus, the proof can be preserved as a sequence of inferences and saved into a disk file. A proof file format has been defined for this purposes. While the complete definition of the proof file format can be found in [Won93b], it is described briefly in Section 4 below.

The approach suggested above requires that the HOL system be modified so as to enable each inference rule to be recorded at the time it is performed. The principle of implementing the recording feature is to make as little change to the core system as possible. Furthermore, the modification to the HOL system should have as little penalty on the system performance as possible especially when the recording feature is not enabled.

The actual modifications to the core system were

- to define a new ML data type to represent the recorded inference rules;
- to modify all basic inference rules to save their names and arguments to an internal list;
- to add a small number of functions to enable/disable the recording and to access the list of saved inferences.

These modifications have been implemented in HOL88 version 2.02 as a small set of low level functions in the core system. The details are described in [Won93b].

In order to use the proof recording feature for practical proofs, a flexible and convenient user interface should also be provided. Such an interface was implemented as a HOL library in HOL88. It is described briefly in Section 5 while the details can be found in [Won94].

A benchmark of the proof recorder was carried out on the correctness proof of a simple multiplier described in [Gor83] which is often used as a benchmark for the HOL system. The results can be found in Section 9.

4 Proof File Format

A recorded proof is saved in a disk file in a format known as **prf** format. Proof files in this format are intended primarily for automatic checkers. They follow the Hilbert style of proofs as described in Section 2. It is a linear model which simplifies both the generation and the checking of proofs.

The proof file format **prf** has two levels: the *core* level allows only primitive inference rules in a proof, and the *extended* level allows all basic inference rules.

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

Although proof files in the **prf** format are text files, they are primarily for use by programs such as proof checkers. They are not for showing proofs to a human reader.

5 The record_proof Library

The **record_proof** Library serves as an interface to the proof recording feature in the core system. It is organised into two levels: the upper level is the user interface intended for users to record proofs and manage proof files; the lower level is for developers who may develop other utilities using the proof recording feature.

5.1 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. The typical use of this feature is

- 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**. Then, the proof is re-done once more in batch mode, and a proof file is generated.

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

load_library'record_proof:disable';;

Usually, the proof script 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 parts of the library as required, the same script file can be used to perform normal proofs and to generate proof files without any modification.

5.2 The Developer's Interface

In addition to the user interface, the **record_proof** library also provides a lower level interface to the proof recorder. This interface consists of a small number of ML functions to allow finer control of the proof recorder. They are useful for developing alternative user interfaces or applications other than proof checking.

The process of recording proofs 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 was 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 documented in [Won94].

6 Checking HOL Proofs

Having modified the HOL system to incorporate the proof recorder and developed the **record_proof** library, HOL proofs can now be saved in proof files. The next phase is to develop an efficient proof checker to check the proofs.

The dominant requirement of this checker is to be able to check large proofs generated from real applications which consist of thousands or tens of thousands of inference steps. This means that the implementation should be fast and efficient, and should be able to perform reasonably well with limited resources, i.e., limited amount of physical memory and disk space. With the eventual formal verification in mind, the checker follows fairly closely von Wright's formal theory, especially the critical part, i.e., the checking of the inferences.

A checker accepting the core level proof file will be relatively simple, so it may possibly be verified formally. A checker for the extended level proofs could be implemented in two different ways. The first approach is to write a program to expand the inference steps involving derived rules into a sequence of primitive steps before being sent to the core checker. This approach has the advantage of utilising the core checker which may be formally verified, therefore, achieving higher confidence in the consistency of the proof. However, this approach can increase the number of inference steps considerably so the amount of time required to check the proof will be much longer.¹ The second approach checks all basic inference rules directly. This approach can result in a more efficient checker since the basic derived rules are relatively simple to check.

No matter which approach is used to implement a checker, its memory requirement is very large for large proofs because all theorems derived in the sequence have to be kept in memory. This is because a theorem derived in an earlier step may be referred to by the very last step. Logically, many modern systems are able to address many gigabytes, even up to terabytes of virtual memory, but physical memory is still limited. When large numbers of theorems are kept in memory, thrashing occurs, thus slowing down the process. This problem has been solved in the efficient checker by processing the proof file in two passes (see Section 8).

In fact, two versions of a checker have been implemented in Standard ML of New Jersey. One of them, the more formal version, accepts the core level proofs only. Its critical part was implemented by translating von Wright's HOL proof theory directly into SML functions. The other version, the more efficient version, can check proofs in the extended level using the direct approach mentioned in a previous paragraph. It is also based on the formal theory but with optimised use of memory. The major differences between these two versions are in the internal representation of types and terms, and the handling of theorem reference. The critical part — the checking of each inference — in both versions follows very closely to the formal theory. The non-critical parts of the two versions, for instance the proof file parser, the I/O handling and so on are identical, and use the same SML source files.

7 Using the Proof Checker

To a user, the checker is a program which reads a proof file, checks the proofs in it and reports back with either a success which means the proofs are correct or a failure which means the opposite. It creates a log file containing information of what hypotheses and stored theorems have been used and the resulting theorems of the proofs. The log file is in a format similar to the proof file.

7.1 Loading the Checker

Currently, the checker program modules have to be loaded into SML by evaluating the expression

¹By examining the derivations of the derived rules, one can see that each derived rule may be expanded into five to twenty primitive rules.

use "join1.sml";

This will compile and link the modules to form the checker. After loading the modules, a top-level function **check_proof** is defined as the entry point to the checker.

When the program becomes stable, it will be possible to save an executable image. Then, the checker will be invoked as a shell command.

7.2 Invoking the Checker

The checker is invoked in SML by evaluating the function check_proof which takes a string as its sole argument. The string is the proof file name which, by convention, has the suffix .prf but the checker accepts any name. If the filename has a suffix .gz, the checker will assume it is a compressed file. It will run a decompresser automatically, and the log file will also be stored in a compressed form. The default compression/decompression utilities are the GNU gzip/gunzip programs. Below is a sample session of using the checker to check a compressed proof file named MUL_FUN_CURRY in the directory proofs parallel to the current directory.

```
- check_proof "../proofs/MULT_FUN_CURRY.prf.gz";
                                                                     1
Current environment: MULT_FUN_CURRY
Proof: MULT_FUN_CURRY
Proof MULT_FUN_CURRY has been checked
Proof: MULT_FUN_CURRY_THM
Proof MULT_FUN_CURRY_THM has been checked
Using the following hypotheses:
<-8>
      - T :bool
   {... theorems deleted}
Proof: MULT FUN
Proof MULT_FUN has been checked
Proof: MULT_FUN_DEF
Proof MULT_FUN_DEF has been checked
Using the following hypotheses:
  {... theorems deleted}
val it = () : unit
```

8 Implementation of the Checker

This section describes briefly the implementation of the checker. The full details including the complete source code can be found in [Won95].

The checker is structured into a number of modules as shown in Fig. 1. The modules can be divided into two groups: the core group and the auxiliary group. Modules in the core group are shown in the figure with thick border, whereas other modules are shown with thin border.

8.1 The Core Modules

The core modules implement the internal representation of HOL types, terms, theorems and proofs, and the checking of the inferences. In the formal version, these modules are translated directly from von Wright's formal theory of HOL proofs. In the efficient version, the internal representation is slightly different. De Bruijin's name-free representation is used for terms.

The Check module contains all functions for checking the consistency of inference rules. In the formal version, this module contains only eight checking functions for the eight primitive rules. These functions are translated directly from the formal theory. As an example, the function for checking the primitive rule ASSUME and its formal definition in the HOL proof theory is shown in Fig. 2. The SML function and the HOL definition are very close.

In the efficient version, the Check module contains functions for checking all basic inference rules. The functions for the primitive rules are the same as the formal version except very minor changes to take care of the slightly different representation of HOL types and terms. The functions for checking other basic rules are derived from specification of these rules found in [GM93]. Fig. 3 shows the basic inference rule SYM and its checking function.

One major difference between the two versions of the checker is that the formal version processes the proof file in one pass while the efficient version in two passes. Since a theorem derived in an inference step may be referred to by any subsequent steps, the checker has to cumulate all theorems in main memory while processing a proof. A practical proof may consist of thousands of inference steps. The storage for theorems will be huge. To overcome this problem, the efficient version processes the proof twice.

In the first pass, it builds a theorem reference table. This table consists of two dynamic arrays whose elements are integers as shown in Fig. 4a. Each element represent a proof line. The indices to the elements are the proof line numbers. Since the proof lines are numbered with both positive and negative numbers, but only non-negative numbers are allowed in indexing the array, two arrays are used. The **TabHyp** array is for the hypothesis lines whose line numbers are negative, and the **TabLine** array is for proof lines whose numbers are positive. These arrays are created using the **DynamicArray** module in the SML/NJ library. Using dynamic arrays instead of static ones releases the upper limit of the number of lines in the proof.



Fig. 1. Checker module structure

Fig. 2. Checking function and formal definition of the primitive rule ASSUME

In the first pass, the checker looks at the justification part of the proof lines. When it encounters a reference to a theorem in a previous proof line, it enters the current line number into the element corresponding to the referred line in the table. For example, when the checker is at Line 3, it finds that this line refers to the theorem in hypothesis Line 1. It enters 3 into the first element of **TabHyp**. To speed up Pass 1 process, the checker can skip over other parts of the proof line quickly. This is done by scanning the input and looking for matching parentheses only. At the end of this pass, each element of the theorem reference table will contain the highest line number which is the latest line referring to the theorem. In the table shown in Fig. 4a, Line 5 is the last line referring to the theorem derived in Line 2 and Line 4.

In the second pass, the checker stores theorems referred to by other proof lines in a theorem table. This table is implemented by a dictionary in the **Dict** module of the SML/NJ library. The key of each entry is the line number. Since the dictionary is represented by balanced splay tree, searching for a theorem is fast. After checking a proof line, the checker examines the theorem reference table, if the value of the current element is greater than the current line number, i.e., it will be referred to later, the theorem is saved in the theorem table. Fig. 4b illustrates the situation in which the checker has just stored the theorem derived in Line 2. when the checker retrieves theorem, it also examines the theorem reference table. If the current line is the last one to refer to the theorem, i.e.,

```
\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}
fun chk_Sym(line, n, thm) =

let val thm1 = get_thm(line, n)

val (left,right) = dest_eq (concl thm1)

in

((right,left) = dest_eq (concl thm)) andalso

(HtermSet.equal((hyp thm), (hyp thm1)))

end
```

Fig. 3. Basic rule SYM and its checking function



a) theorem reference table b) theorem Table

Fig. 4. Data structures for theorem references

the current line number is equal to the value in the table, the theorem is removed from the dictionary. Continuing the scenario in Fig. 4b, the next line is Line 3, it refers to hypotheses Line 1. Since this is the last line referring to the theorem, the checker removes it from the table. This arrangement minimises the number of theorems stored in the table, thus reduces the memory requirement.

8.2 Auxiliary Modules

The HOLProofKey module defines the concrete syntax, i.e., the tags, of the proof files. The **Parsing** module consists of several higher order parsing functions. The parser proper is in the modules **Pass1** and **Pass2**. It is a recursive descend parser.

The Exception and Debug modules are responsible for handling errors. The Debug module maintains a debug flag for each module. The values of these flags are non-negative integers. Higher the value, more the information will be display while checking a proof. The Report module is for formatting the output to the log file.

The Io module handles all file input and output. When the checker is invoked, it creates a decompression process running as a filter in the background. The communication between this process and the checker is via a UNIX domain socket. In the case the file is uncompressed, no decompression is needed, but a dummy cat process is created, and the communication is still via a socket. This arrangement simplifies the checker as its input routine always reads from the input socket. Similarly, an output socket is created with a compression process to compress the output to the log file on the fly. This arrangement is illustrated in Fig. 5.



Fig. 5. Checker input/output arrangement

9 Benchmarking

A proof of correctness of a simple multiplier described in [Gor83] is often used as a HOL benchmark. This is a small to medium size proof which generates 14500 intermediate theorems. This proof has been used to test the proof recorder and the checker.

The multiplier proof consists of four ML files. A proof file is generated for each ML file. It contains all the sub-proofs in the corresponding ML file. Table 1 lists the time taken to record this proof and the proof file size. Two tests was carried out: the first with the proof recorder disabled; and the second with it enabled. The run time and the garbage collection time (GC) reported by the HOL system are listed under the columns headed **DISABLED** and **ENABLED**, respectively. The tests ran on a SUN Sparc 10 Server.

As the figures in the table show, the time (2412.9 seconds \simeq 40 minutes) required to record the proof and generate the proof files is considerable longer than to perform the proof only, 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 files. This extra time is acceptable since the proof files will only be generated after the proof is completed satisfactorily (probably once) and be ran in batch mode.

The sizes of the proof files are also listed in Table 1. They are very large (43

FILE	No. of	DISABLED		ENABLED		SIZE	
	THMS	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	1	-	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

Table 1. Benchmark of recording the multiplier proof (Time in seconds and size in bytes)

Proof	\mathbf{Time}					
${f File}$	Run	\mathbf{System}	GC	Real		
mk_NEXT	139.3	15.3	3.7	170.0		
MULT_FUN_CURRY	77.3	9.6	2.6	100.4		
MULT_FUN	406.3	44.6	15.4	488.8		
HOL_MULT	1472.1	152.0	98.5	1783.3		
Total	2095.0	138.8	120.2	2542.5		

Table 2. Benchmark for checking the multiplier proof (Time in seconds)

Mbytes in total) because every intermediate theorem has to be saved. The size per theorem is comparable to the theory files in HOL88. 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. As the compression is done automatically, this does not pose too much burden to the user.

The multiplier proof files were successfully checked by the checker. No error was found from the proof files. Table 2 lists the time taken to check the proof files. This test ran on a SUN SparcStation 20.² The time is in the same order of magnitude as the recording. One important observation is that the process size is relatively small when performing the checking. The process size of the checker when it is just loaded is 14 Mbytes. The maximum size when performing the checking is only 16 Mbtyes. This shows that the implementation does keep the memory usage very small.

10 Conclusions

The research described in this paper shows a method of independent checking to ensure the consistency of mechanically generated proofs in LCF-like systems. A benchmark has shown that the proof checker is able to check a practical proof consisting of several thousands of inference steps. The application of this method is mainly in the formal verification of safety-critical and high-integrity systems. After a formal proof has been generated by a contractor, there is a need for an independent means of assessing the consistency of the proof.

There has been little work in the area of verified proof checking. The notable exception is the work of Boyer and Dowek on proof checking for Nqthm proofs [BD93]. Other theorem prover, such as nurpl[ea96], which makes use of proof objects in runtime to implement transformation tactics, may utilise similar approach to do proof checking.

If the proof checker itself can be formally verified, it will greatly boost the confidence in the consistency of checked proofs. Since the checker has been implemented using von Wright's formal theory of the HOL proof system as its

 $^{^{2}}$ The recording tests was carried out much earlier than the checking test. In fact, it was done before the checker was developed. The author was not able to access the same type of machine to do the checking test.

specification, it is possible to formally verify the checker provided that a formal semantics of Standard ML is developed. Attempts have been made to establish a formal semantics of Standard ML in HOL [Sym93] [VG93]. One approach to formally verify the proof checker is to reason based on the formal semantics directly, but there still more work needs to be done before one can attempt a formal verification of a practical program such as the checker. Another approach of using refinement has been suggested by von Wright [vW95].

In addition to being a format to communicate proofs between the HOL system and the proof checker, the proof file format may be used to communicate between different theorem prover systems with similar logic, such as between HOL88 and HOL90. It can also make the proofs themselves become deliverables. Having a proof saved as a sequence of inference steps allows new tools to be developed to analyse the proof, for instance, a tool generating a dependency graph to show the use of theorems in deriving a new theorem.

Although the recorder and checker is able to handle medium size proofs, there are several improvements could be made. Firstly, the proof file format could be changed to eliminate some redundancy and to reduce the file size. Secondly, the proof recorder could be improved so that the proof could be written into a file while it is being generated. This improvement will likely be done in HOL90. The proof checker now checks individual proofs in a flat environment. It is important to develop a system to manage the proofs and theorems in a more structured and hierarchical way if the checker is to be used in real applications which may contain many subproofs and their dependency is very complex.

Acknowledgement

The idea of recording inference steps and generating proof lines has been suggested by many people including Malcolm Newey and Keith Hanna. Mike Gordon implemented a prototype of the recording functions in HOL88. The proof recorder described in this paper improved and enhanced this prototype. The translation of the formal HOL proof theory into ML functions used in the formal version was carried out by John Herbert. It was a great pleasure to work with Mike, Paul, Brian and others in the Hardware Verification Group in the Computer Laboratory in Cambridge. The work described here would not have been completed without their invaluable advice, help and company.

References

- [BD93] Robert S. Boyer and Gilles Dowek. Towards checking proof checkers. In Workshop on types for proofs and programs (Type '93). 1993.
- [Bou92] R. J. Boulton. On efficiency in theorem provers which fully expand proofs into primitive inferences. Technical Report 248, University of Cambridge Computer Laboratory, 1992.

- [ea96] Constable et al. Implementing Mathematics with the Nuprl proof development system. Prentice-Hall, 1996.
- [GM93] M. J. C. Gordon and T. F. Melham, editors. Introduction to HOL a theorem proving environment for higher order logic. Cambridge University Press, 1993.
- [Gor83] M. J. C. Gordon. LCF_LSM, A system for specifying and verifying hardware. Technical Report 41, University of Cambridge Computer Laborartory, 1983.
- [oD91] Ministry of Defence. Requirements for the procurement of safetycritical software in defence equipment. Interim Standard 00-55, April 1991.
- [Sym93] D. Syme. Reasoning with the formal definition of standard ML in HOL. In Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science No. 780, pages 43-58. Springer-Verlag, 1993.
- [VG93] M. VanInwegen and E. Gunter. HOL-ML. In Higher Order Logic Theorem Proving and Its Applications, Lecture Notes in Computer Science No. 780, pages 59-72. Springer-Verlag, 1993.
- [vW94] J. von Wright. Representing higher order logic proofs in HOL. In Thomas F. Melham and Juanito Camilleri, editors, Higher Order Logic Theorem Proving and Its Applications: 7th International Workshop, volume 859 of Lecture Notes in Computer Science, pages 456-470. Springer-Verlag, September 1994.
- [vW95] J. von Wright. Program refinement by theorem prover. In Proceedings of the 6th Refinement workshop, Lecture Notes in Computer Science. Springer-Verlag, 1995.
- [Won93a] W. Wong. Formal verification of VIPER's ALU. Technical Report 300, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, ENGLAND, May 1993.
- [Won93b] W. Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, ENGLAND, July 1993.
- [Won94] W. Wong. The HOL record_proof Library. Computer Laboratory, University of Cambridge, 1994.
- [Won95] W. Wong. A proof checker for HOL proofs. Technical report, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, ENGLAND, 1995. to be published as technical report.

This article was processed using $\LaTeX\mbox{T}_{\mbox{E}}X\,2\,\varepsilon$ according to the LLNCS style