

Representation and Validation of Mechanically Generated Proofs Final Report

Principal Investigator: M.J.C. Gordon
University of Cambridge Computer Laboratory

1 Introduction

The goal of this project was to demonstrate the feasibility of the *independent* and *trusted* validation of the proofs generated by existing theorem provers. Our intention was to design, implement and formally verify a proof checking program for HOL [5] generated proofs. A proof checker can be much simpler than a full theorem prover such as HOL as it is only concerned with checking existing proofs rather than searching for or generating them. Our work has clearly demonstrated the feasibility of this approach. In particular, the main achievements of the project are as follows.

- We have developed a computer representation suitable for communicating large, formal, machine generated proofs.
- We have modified the HOL system to allow primitive inference proofs to be recorded in the above format.
- We have formalised, within the HOL theorem proving system, theories of higher-order logic, Hilbert-style proof and HOL proof. This amounts to specifying the requirements and giving a higher-order logic functional specification of a proof checker.
- Proof tools to execute the specifications using formal proof have been developed. They have been used to validate via testing some of HOL's rules.
- We have developed an imperative, modular programming language. A novel verification system has been developed for it on top of HOL.
- An imperative version of the proof checker has been developed and formally verified against the functional specification. The programming language used is not in itself executable. The proof checker was therefore translated into a similar subset of ML to allow execution.
- The above version of the proof checker was a prototype developed to demonstrate the feasibility of verified proof checkers. However, it is not very efficient and does not use the proof format developed. Therefore to demonstrate that efficient proof checkers are feasible, we have also implemented an efficient proof checker for proofs in the proof format.
- The efficient proof checker has been used to check the proof of a standard HOL benchmark proof consisting of more than 14 thousand inferences.
- We have developed program refinement technology which would allow the inefficient checker to be refined into an efficient one whilst preserving correctness.
- Errors and anomalies in HOL's rules have been found and corrected.

2 Definition of Proof Texts

We have adopted the Hilbert-style sequent notion of proofs for this work. A proof is a linear sequence of lines where each line is justified by a primitive logical inference applied to preceding lines. A proof format has been developed for recording such proofs [15, 14], so that they can be communicated between the theorem prover and an independent proof checker. The format is similar to LISP S-expressions. It starts with an environment: the constants and types known to the theory. The environment is followed by a series of recorded proofs. Each recorded proof consists of a list of the goals of the proof and a series of proof lines. The proof lines consist of the theorem proved by the line, together with a justification giving the inference rule and the previously proved theorems used. The proof format is text based, but is primarily for use by proof checkers rather than human readers. Machine generated proofs are too large to be understood by humans.

3 Generation of Proof Texts

HOL proofs are normally transient: concrete proofs are not stored. The HOL system was therefore modified to generate a record of the primitive inferences used in a proof [15, 14]. This consisted of two tasks. Firstly, the primitive inference rules were modified to save their names and arguments to an internal list when used. Secondly, a library was developed to enable and disable proof recording and to write the proofs out to disk in the proof format. Typical proofs involve a vast number of primitive inference steps. For this reason the proof files were compressed as they were created. These changes were incorporated into HOL Version 2.02.

4 A Formal Theory of HOL Proof

A formal theory of HOL proof has been developed within the HOL system [11]. Datatypes within the HOL logic have been defined to represent HOL types and terms. Notions of well-typedness, free and bound variables, alpha renaming, substitution, beta reduction and type instantiation are defined. These are represented by higher-order logic predicates over the new datatypes. The predicates are true if their arguments represent an instance of the given notion. For example `Pfree x t` is true if `x` is a free variable in the term corresponding to `t`. These predicates form the basis of the definitions of valid inferences.

A new HOL type of sequent is defined together with a new type representing the primitive inference rules (with one constructor per inference rule). The latter gives an abstract syntax for inference steps. We also defined their semantics. Inference rules relate a series of hypothesis sequents to a conclusion sequent. The semantics of an inference rule is thus represented by a predicate on these sequents. Other information in the form of a term is sometimes needed to represent side conditions. Such predicates have been defined for each of the primitive inference rules (and axioms) of the HOL logic. An ML proof function is also defined for each inference rule. They execute the rules by proof: expanding definitions and simplifying the result using automatic formal proof. This gives theorems stating the truth or falsity of applications of the predicates to given concrete arguments. The proof functions can thus be used to check if specific inferences are valid.

Next we considered the notion of a proof. Syntactically a proof is a sequence of inference steps. Semantically, each step must be a correct inference, and all hypotheses must appear as conclusions earlier in the sequence. This is captured by a predicate `is_proof`. This is a functional specification for a proof checker of HOL proofs. A corresponding execution function in ML was also defined. It is an

executable proof checker, if a slow one. It is very secure in that it is executing the actual specifications. It has been used to validate inference rules by applying it to HOL's results for specific inferences.

We have thus developed a theory of concrete HOL proofs. We also define the more abstract notion of *provability*. A goal is provable if it is an axiom or can be inferred from some sequence of provable sequents by the correct application of an inference rule. This is defined inductively. Provability gives the requirements specification for the concrete notion of proofs. A concrete proof as defined above should only satisfy the predicate `is_proof` if it proves a provable theorem. We have proved a correctness theorem to this effect in HOL, thus validating the proof function proof checker. Other “reasonableness” properties of proofs have also been proven such as the fact that proofs can only yield sequents with well-typed conclusions.

We have defined a notion of derived inference rule in terms of provability. Some of the derived HOL inference rules are actually hard-wired into the implementation for efficiency reasons (ie treated as primitive inferences). Our formal theory allowed us to formally verify these derived inference rules. Furthermore, we have proved that the notion of proof using only the primitive rules and the notion of proof using both primitive rules and correct derived rules are equally strong.

5 Verification of a Proof Checker

The formal specification of the proof checker was used to develop an imperative, modular implementation. The implementation language is a simple, clean language devised for the purpose. For the proof checker to be verified the implementation language must have both a formal semantics and a corresponding proof system. We developed a weakest precondition calculus of total correctness for our implementation language together with parsing and pretty printing support. A novel verification methodology was devised. This was necessary because reasoning about large programs using conventional Hoare logic is not very practical due to the complexity of proof rules for procedures. Our methodology involves showing that calls to procedures of the implementation are equivalent to calls to functions in the specifications [12]. This methodology was used to formally verify that the imperative proof checker satisfies the functional specification [9]. This involved proving that each procedure in the implementation implements the corresponding functional specification for the procedure.

No compilation technology was implemented for our implementation language, so the proof checker was not executable as such. Instead, we translated the procedures into ML, using its imperative constructs. This translation was largely straightforward as the subset of ML used is very similar to the deterministic fragment of our language [9].

6 Implementation of an Efficient Proof Checker

The implementation that was verified was relatively inefficient because it closely followed the functional specification. It also did not use the concrete proof format generated by the proof recorder. Instead the proofs that were checked were in the abstract syntax developed for the theory of proofs. This was because it was intended as a prototype to demonstrate that a proof checker could be verified, rather than as a practical checker. However, it was desired, not only to demonstrate the feasibility of verifying proof checkers, but also to show that independent proof checking itself was a practical technology. Therefore, in parallel to the development of the verified proof checker, a second efficient Standard ML implementation was developed [16, 14]. It

was designed to check proofs in the proof recording format described above and be efficient in both time and space requirements. In particular, in addition to local optimisations, it deals with inputting proofs (decompressing them as needed). A two pass algorithm is used to minimise the amount of the proof that must be stored in memory at once.

Whilst not being formally verified, this checker was developed using current best practice in that it was implemented from the formal specifications. As such it can be regarded as a relatively secure system. The proof checker was documented using a literate programming system: `mweb`. Thus the level of documentation is very high and it is a very open system.

The efficient checker has been used to check the proof of a multiplier circuit. It is used as a benchmark to compare the performance of different HOL systems running on different platforms. It consists of over 14 thousand primitive inference steps. It was successfully checked.

7 Refinement Technology

Ultimately, a proof checker that is both verified and efficient (in time and space) is desirable. This could be achieved by refining the verified but inefficient checker into a form similar to the efficient version in a way that preserves total correctness properties. In conjunction with researchers in Åbo Akademi University, Finland, we have developed advanced refinement technology within HOL [13, 10]. It allows refinements to both control and data structures of the kind that would be required. We have developed a style of data refinement which, unlike conventional techniques, can be performed separately on each subcomponent. The system uses HOL's window inference system [6] which supports a transformational style of reasoning ideally suited for refinement. Embedding the verification and refinement technology in HOL in this way means that programs can be developed, verified, refined and all proof obligations discharged within a single unified framework.

8 Other Results

Other work performed under this grant include the formalisation of a model of the lambda calculus in HOL-ST [1]. HOL-ST [4] is a development system devised to investigate the use of set theory as the foundation of a HOL-style theorem prover. The formalisation used the inverse limit construction of domain theory. The integration of set theory into HOL is a step towards providing independent proof checking for reasoning within set based formalisms such as Z and VDM. A practical application of this would be the independent checking of HOL proofs that SPARC ADA programs meet Z specifications.

We also developed a prototype theory manager for HOL which allows multiple theories to be developed within a single HOL session [3]. This makes the management of large proofs much easier, and can help reduce the size of proofs. This eases the space problems of proof checking.

9 Related Work

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 [2]. Other related work includes that of Homier and Martin on the verification of a verification condition generator in HOL [7]. They verified a functional specification of a verification condition generator. Our verification methodology could be used

to verify an imperative version. There has been work on proof checking using a different proof paradigm to that which we have investigated: the propositions-as-types paradigm. It involves creating lambda-terms as proof objects. The soundness of these proof objects is verified by type checking. There has also been some work on using proof checkers to check meta-theory [8].

10 Conclusions

The importance of our work is that we have demonstrated that verified proof checking is feasible for the kind of theorem provers that are likely to be used by industry in the future. We have demonstrated both the practicality of independent proof checking and the feasibility of verifying a proof checker. This involved formalising the notion of proof within HOL. We have also developed a novel proof methodology for verifying and refining large programs. In addition to verifying a full proof checker for HOL, we have demonstrated that our formal theory of proof can be used to verify derived inference rules, allowing them to be treated as primitive, thus increasing our faith in the HOL system itself. Indeed, as a result of this work, an error and various anomalies were discovered in the implementation of the basic inference rules in the HOL system. This has led to changes to the HOL system.

The work supports the needs of the safety and security critical communities. For example, the Draft Interim Defence Standard 00-55 mandates the use of proof checkers. Applications of proof checking technologies include the checking of proofs of security properties of systems. Interest in the work has already been shown by SRI and is expected from other industry.

11 Further Work

Whilst this work has achieved all our original objectives, there are several areas where further work is desirable.

The efficient version of the proof checker was not formally verified. Whilst we developed refinement technology, it remains to actually do the refinement of the inefficient version.

The verified checker was implemented in a language for which no compiler technology exists. It would be desirable to develop and formally verify a compiler implementation for the language. This would both provide a trusted means of executing the proof checker and provide a large example for the program verification methodology. Compiler verification is traditionally split into two. The implementation is verified against a functional specification which is in turn verified against a more abstract specification. It would thus fit our verification methodology well.

The efficient checker was used to verify a small to medium sized proof. Further work is required to demonstrate that the technology scales to industrial sized proofs which can be orders of magnitude larger. The proof checker checks individual proofs. No checks are made as to whether the external theorems used by the proof have themselves been checked. Thus, for large verification efforts involving many proofs, a higher level of management is required. It is not clear how this might best be done without overly complicating the checker and duplicating the prover.

The HOL system was used to formalise its own logic and validate its own derived inference rules. Similar formulations could be given for other logics and systems. A more general framework for devising such formulations and validating corresponding proof checkers could be developed. This would also open the possibility of comparing the notions of proofs in different systems which could be used as the formal basis for linking together different proof systems.

References

- [1] S. Agerholm. Formalising a model of the λ -calculus in HOL-ST. Technical Report 354, University of Cambridge Computer Laboratory, November 1994.
- [2] Robert S. Boyer and Gilles Dowek. Towards checking proof checkers. In *Workshop on Types for Proofs and Programs (Types '93)*, 1993.
- [3] Paul Curzon. Virtual theories. Submitted for Publication, 1995.
- [4] Mike Gordon. Merging HOL with set theory. Technical Report 353, University of Cambridge, Computer Laboratory, November 1994.
- [5] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [6] J. Grundy. A window inference tool for refinement. In *Proc. 5th Refinement Workshop*, London, January 1992. Springer-Verlag.
- [7] Peter V. Homeier and David F. Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. 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 269–284. Springer-Verlag, September 1994. To appear in *The Computer Journal*.
- [8] J. McKinna and R. Pollack. Pure type systems formalized. In Herman Geuvers, editor, *Typed Lambda Calculus and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 289–305. Springer-Verlag, 1993.
- [9] J. von Wright. The formal verification of a proof checker. SRI internal report, November 1994.
- [10] J. von Wright. Program refinement by theorem prover. In *Proc. 6th Refinement Workshop*, London, January 1994. Springer-Verlag.
- [11] J. von Wright. Representing higher-order logic proofs in HOL. Technical Report 323, University of Cambridge Computer Laboratory, 1994. Also in *Higher Order Logic Theorem Proving and Its Applications*, *Lecture Notes in Computer Science*, 859, 1994. To appear in *The Computer Journal*, 1995.
- [12] J. von Wright. Verifying modular programs in HOL. Technical Report 324, University of Cambridge Computer Laboratory, 1994.
- [13] J. von Wright, J. Hekanaho, T. Långbacka, and P. Luostarinen. Mechanising some advanced refinement concepts. *Formal Methods in Systems Design*, 3:49–81, 1993.
- [14] W. Wong. Recording and checking HOL proofs. Submitted for publication.
- [15] W. Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, July 1993.
- [16] W. Wong. A proof checker for HOL proofs. To appear as a University of Cambridge Computer Laboratory technical report, 1995.