Checking Proofs from Linked Tools

Paul Curzon[†] and Wai Wong[‡]

[†]University of Cambridge Computer Laboratory, UK.
[‡]Hong Kong Baptist University, Hong Kong

Abstract

We describe a Cambridge project (now completed) which demonstrated the feasibility of producing independent, verified proof checkers for the HOL theorem proving system[1]. We then briefly overview a joint Cambridge University/Hong Kong Baptist University proof checking project which is about to commence. It aims to extend the HOL work to other logics and proof tools. We discuss how this relates to the formal linking of tools and theories.

1 Independent Proof Checking

There is a growing interest in the use of formal methods in the validation of computer systems. Correctness proofs tend to be very long and shallow in the sense that they are not mathematically interesting. As such they can only realistically be carried out with any degree of confidence using machine assistance. A wide variety of different theorem proving systems incorporating various degrees of automation have been developed to this end, embodying various underlying logics. However, theorem provers are themselves just computer systems which can themselves contain errors. In many correctness-critical applications (eg safety and security critical systems) some form of independent certification of the correctness case is required. In the case of a formal proof, one approach is the independent checking of the proofs generated by the theorem prover. For the same reasons that machine help is required to generate proofs, it is also needed to certify them. This is especially so, due to the absence of social processes that are normally used to validate mathematical proofs. Machine assistance can be provided in the form of a proof checker. For example, the proposals for proof contracts in the UK Draft Interim Defence Standard 00-55 [3] state:

> Any formal proof of correctness shall be supplied in a computer readable form. This shall be suitable for input into a deterministic proof checking program.

Draft Interim Defence Standard 00-55 also mandates that the tools used in the development of a system should be validated to a similar degree of rigour as the system itself. This suggests that proof checkers should also be formally verified. Such a requirement occurs in the security critical community.

A proof checker can be much simpler than a full theorem prover because it only needs to check proofs, not search for or generate them. For this reason formal verification is much more tractable. It also means that different issues arise in their design. The feasibility of such verified proof checkers, was demonstrated by the HOL proof checker project. A proof

[1] For further information see URL http://www.cl.cam.ac.uk/Research/HVG/proofchecker/

checker was developed by Wai Wong [10] for proofs created using HOL [2], a widely used theorem proving system based on higher-order logic. Furthermore a version of the proof checker was also formally verified by Joakim von Wright using HOL [8]. We overview this project in Section 2.

The general idea of conducting proof checking independently of proof generation was, to our knowledge, first proposed by Malcolm Newey in the early 1970's. He was then working at Stanford University with Robin Milner. More recently the idea was revived by Keith Hanna of Kent, in the specific context of hardware verification. The proposal for the HOL proof checker project was developed originally in conjunction with Adelard Ltd. The need for independent proof checking has also been recognised by the QED international collaboration which aims to build a machine-readable repository of a significant body of mathematics [5]. To this end, McCune has recently written an (unverified) proof checker for the Otter theorem prover.

2 A Proof Checker for HOL generated proofs

The goal of the HOL proof checker project was to demonstrate the feasibility of the *independent* and *trusted* validation of the proofs generated by existing theorem provers. This work was done by Joakim von Wright, Wai Wong and Mike Gordon. The main achievements of the project were as follows.

- A computer representation suitable for communicating large, formal, machine generated proofs was developed.
- The HOL system was modified to allow primitive inference proofs to be recorded in the above format.
- Theories of higher-order logic, Hilbert-style proof and HOL proof were formalised within the HOL theorem proving system. This amounts to specifying the requirements and giving a higher-order logic functional specification of a proof checker.
- An imperative version of the proof checker was developed and formally verified against the functional specification. The programming language used was 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, an efficient proof checker for proofs in the proof format was also implemented.
- The efficient proof checker was used to check the proof of a standard HOL benchmark proof consisting of more than 14 thousand inferences.
- Errors and anomalies in HOL's rules were found and corrected.

2.1 Definition of Proof Texts

The Hilbert-style sequent notion of proofs was adopted for this work. A proof is a linear sequence of lines where each line is justified by a primitive logical inference applied to preceeding lines. A proof format was developed by Wai Wong for recording such proofs [9, 10], 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 and detailed to be understood by humans.

2.2 Generation of Proof Texts

HOL proofs are normally transient: concrete proofs are not stored. The HOL system was therefore modified by Mike Gordon and Wai Wong to generate a record of the primitive inferences used in a proof [9, 10]. 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.

2.3 A Formal Theory of HOL Proof

A formal theory of HOL proof was developed within the HOL system by Joakim von Wright [8]. Data types within the HOL logic were 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 data types. 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 was 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. Their semantics were also defined. 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 the notion of a proof was considered. 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.

A theory of concrete HOL proofs was thus developed. The more abstract notion of *provability* was also developed. 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. A correctness theorem to this effect was proved in HOL, thus validating the proof function proof checker. Other "reasonableness" properties of proofs were also proven such as the fact that proofs can only yield sequents with well-typed conclusions.

A notion of derived inference rule in terms of provability was defined. Some of the derived HOL inference rules are actually hard-wired into the implementation for efficiency reasons (ie treated as primitive inferences). These derived inference rules could be formally verified using the formal theory. Furthermore, the notion of proof using only the primitive rules and the notion of proof using both primitive rules and correct derived rules were proved as being equally strong.

2.4 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. A novel weakest precondition based proof methodology devised by Joakim von Wright [7] was used to formally verify that the imperative proof checker satisfies the functional specification. This involved proving that each procedure in the implementation implements the corresponding functional specification for the procedure.

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

2.5 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 [10]. 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 inputing 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 [11].

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.

3 Proof Checkers for Linked Proof Tools

A wide variety of different proof systems exist with various underlying logics. As formal proof gains wider acceptance even more proprietary systems may be developed, tailored for specific applications. Furthermore, the verification of any particular system is likely to make use of more than one verification tool. Such use of a wide range of tools increases the risks that individual tools contain bugs which could undermine the entire verification. Therefore the need for proof checkers may be even more acute.

One approach which overcomes this problem to a large extent is to embed new tools and their underlying theory in a general purpose theorem prover. For example a proof tool for the Duration Calculus has been embedded in PVS [6]. A wide range of different theories have also been embedded within the HOL system (see for example, [1]). This approach has the advantage that a very secure system can be built, especially if the definitional fully-expansive methodology is followed. Furthermore, only a single checker for the proof system is required by certification authorities. However, such systems can be harder and more time-consuming to create than a directly coded system. A particular problem is in ensuring that the proof is conducted at the level of the embedded theory so that the host theory is hidden. Such systems can also be slower than directly coded tools, and ensure that embedded systems are usable is an ongoing research area. Furthermore, commercial realities mean that a single universal tool in which all tools are embedded is not likely to happen. Thus a wide variety of different, directly coded proof tools will exist.

Proof checking may offer a way to increase the security of directly coded tools, irrespective of certification authority requirements. In this approach each new theory is formalised within a single universal logic. This amounts to embedding the theory in the logic. However, the difficult interface, speed and usability problems do not arise. A proof checker is developed from this specification (and possibly verified against it). The proof checker can be very simple: as with the HOL checker it can be essentially the translation of the formal description into an executable language. The proof tool can be separately and directly coded. It will be more complicated than the proof checker. The user interface issues must be addressed. It must also be able to search for proofs, taking large steps, rather than just checking long chains of small steps. It is thus more likely to contain errors. It must also be able to output a recorded proof, which omits the search steps in a form to be input by the checker.

The directly coded proof tools can then be used during the development process. The checker is used to validate the final proofs. In this way the correctness of the proof tool is no longer critical and in particular does not need to be verified. Only the correctness of the checker is critical. Certification authorities can use their own independently programmed checkers to certify the proof.

Ideally, different proof tools should not be used together in an ad hoc manner, but formally linked. Indeed, the requirements of certification authorities are liable to make this a necessity for safety critical systems. The proof checking scenario fits well with formally linked tools, since both involve formalising the underlying theory of the tools. The formalisation of the links will also provide the basis for the checking of those links.

For the above scenario to be feasible, checkers for new theories must be quickly and easily created, since checkers are needed for each theory. The aim of the Hong Kong/Cambridge project is to investigate the feasibility of quickly creating different checkers.

We aim to develop a system whereby verified proof checkers for the proofs generated by theorem provers implementing different logics can be quickly obtained. Formal proof is at its best when tailored to particular application areas. In this situation results can be reused and generic tools can be developed. This should be true for proof checkers. There is a great deal of similarity between different proof systems and many concepts are shared.

Thus it should be possible for much of the work in developing and verifying proof checkers to be reused or automated. Furthermore, whilst different checkers would be required for each theory, a single checker could be used for different tools for that theory.

We will design a notation for defining logics. It is envisaged that this notation be similar to those used by generic theorem provers such as Isabelle [4]. However, different issues will predominate. For example, it may be possible for the notation to be much simpler than for a generic theorem prover. Details such as precedence and infix status of operators will not be needed since the proofs to be checked are not intended to be human readable.

We will define tools for translating this notation into a series of definitions in higher-order logic suitable for input to the HOL system. These definitions will provide the specification of the proof checker for the new logic. They will draw from standard definitions and templates of definitions. Facilities will need to be provided so that non-standard definitions can also be provided by the user. We will provide tools to aid the creation of both an implementation of the proof checker specification, and of the formal proof that the specification and implementation correspond.

References

[1] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van-Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design*, pages 129–156. North-Holland, 1992.

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

[3] MoD: Interim defence standard 00-55 on the procurement of safety critical software in defence equipment, 1991.

[4] L.C. Paulson. Introduction to Isabelle. Technical Report 280, University of Cambridge, Computer Laboratory, January 1993.

[5] The QED Manifesto. Ftp from info.mcs.anl.gov at pub/qed/manifesto, 1993.

[6] J.U. Skakkebæk. A verification assistant for a real-time logic. Technical Report ID-TR:1994-150, Department of Computer Science, Technical University of Denmark, 1994.

[7] J. von Wright. Verifying modular programs in HOL. Technical Report 324, University of Cambridge Computer Laboratory, 1994.

[8] J. von Wright. Representing higher-order logic proofs in HOL. *The Computer Journal*, 38(2):171–179, 1995.

[9] W. Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, July 1993.

[10] W. Wong. Recording and checking HOL proofs. In P.J. Windley E.T. Shubert and J. Alves-Foss, editors, *Higher Order Logic Theorem Proving and Its Applications: 8th International Workshop*, volume 971 of *Lecture Notes in Computer Science*, pages 353–368. Springer-Verlag, 1995.

[11] W. Wong. A proof checker for HOL. Technical Report 389, University of Cambridge Computer Laboratory, March, 1996.