The wHOLe System

Mark E Woodcock

Department of Defense, Fort Meade MD USA
Department of Computer Science and Electrical Engineering, UMBC, USA
woodcock@cs.umbc.edu

Abstract. A theorem-proving system derived from Higher Order Logic (HOL) is described. The system is designed to support the verification of code written in its own implementation language (SML/NJ), and to allow code which is proven to preserve equality of hol_terms to be used to extend the system. The extension is shown to be theoretically conservative, and the implementation to be reliable. It is hoped that significant efficiency gains can be realized employing this technique (Computational Reflection).

1 SYSTEM SOUNDNESS

1.1 Impossible Ideal

The ideal of formal logic correctness is known as Hilbert's programme. Hilbert hoped to devise a logical engine which could: 1) solve any logical problem, and 2) be used to demonstrate the soundness of the engine itself. Goedel's incompleteness theorems proved that the programme was unobtainable in both respects. The theorems established that:

1) for any interesting logic (i.e. one that includes at least arithmetic over the integers) there are statements in the logic that are true, but can not be proved

2) no formal logic is sufficiently powerful to establish its own soundness.

This induces a temptation to justify the soundness of a desired logic by using some more powerful logic. Unfortunately, this merely pushes the problem off, since there would remain an obligation to demonstrate that the "more powerful" logic was itself sound. Eventually, one of these logics would have to be justified without resort to a formal proof. Clearly, some level of informal analysis is unavoidable in the pursuit of highly trustworthy systems.

In practice, the more powerful logic is one such as set theory that has been accepted to be sound through (informal) community review. This logic is then used to justify the soundness of the theorem proving system in a non-automated proof[1].

[1] Automation is *not* required for formality, but it does provide for more reliable checking of details.

1.2 Practical Goal

Full (formal, automated or neither) analysis of a theorem proving system, however, would be a decidedly painful experience. Such systems are typically large, and include many features which have little relation to the soundness of the system. Consequently, the practical ideal which has developed in the community is of a system which is constructed in two main layers: 1) a small soundness relevant core, and 2) the rest of the system features wrapped around the core. This reduces the scale of the target of informal analysis to a manageable size.

A second possible benefit to a manage-ably-sized (for non-automated analysis) core is that the hand proof could be published and be vetted by the community at large. Unless the core itself is reasonably small, it is unlikely that the soundness demonstration would be broadly accessible (even to experts in the field).

A third benefit to a system with a small core of primitives is that the system could be designed to output the primitive proof steps, allowing a complete separation between the proof discovery and proof checking aspects of the system. This would provide compelling justification that only the (checking) core of the system would require informal analysis.

Even better, once such a checker had been built and vetted by the community at large, any proposition which had its proof checked by the primitive checker would have been effectively vetted by the community as well. This scheme would maximize the leverage which could be obtained from the investment needed to achieve the second benefit.

1.3 Plan for Realization

Fortunately, HOL is a system designed with this ideal in mind. The logic itself is based on a few axioms, rules of inference and principles of definition. The soundness of this logic is justified by a proof sketch, in chapters 15 and 16 of the HOL documentation[1]. Unfortunately, the system as originally implemented was not as simple or direct as the logic design. Many of the inference rules (for which justifications in terms of the primitive rules are well-documented) were implemented as primitives for efficiency reasons.

This was one of the key motivators for the use of the HOL Light (HOLL) [2] system as the basis of this effort. Only those things which are documented as primitives are actually implemented as primitives. The implementation of the core of the system (except for two rules which are presented later, merely to make the presentation flow smoothly) requires only about 1200 lines of ML code—most of the first 500 or so are just simple library functions [2]. The extension of the original proof sketch (for the revised core) of correctness is documented for this system [3].

[2] wHOLe includes another approximately 400 lines of library code designed to facilitate the translation from CAML to SML

2 EXTENSION

The system presented here is layered on top of the HOLL system, so we will only present justifications for the soundness of the extensions to HOLL. We have extended the core with only two rules:

1. a primitive rule of inference which constructs new theorems by applying one of a set of "certified" pieces of ML code to an existing theorem

```
fun USE_PR n (Sequent(asl,c)) =
    if (can (assoc n) (!valid_rules))
    then (Sequent (asl, (assoc n (!valid_rules)) c))
    else raise Failure "No rule for "^n^" defined.";
```

2. an extension rule which checks that proposed pieces of ML code have the desired property, and if so, adds the code to the "certified" set

```
  fun mk_proof_rule name text prf =
  let val abs_fun = hol_parse text
       val oper = X ("APPexp_e ("^abs_fun^")")
       val rando = X "(PARatexp_e (x:exp_e))"
       val rhs = Comb(oper,rando)
       val x = Var("x",Tyapp("exp_e",[]))
       val (Comb (Const ("SOME_My:A->A option_My"),"result:A")) =
             (mk_comb(X "compute_exp s_i E'_0",rhs))
       val () = mk_comb(X "FST:(val_pack#state)->val_pack",

       val goal = mk_forall(x,(mk_eq(x,result)))
      val func = EVAL.eval text
  in
      (PROVE(goal,prf) handle Failure _ =>
raise Failure "Function not truth preserving!";
      if (can (assoc name) (!valid_rules))
      then raise Failure "Rule Name already in Use"
      else (valid_rules := (name,func)::(!valid_rules);
   (rule_just := (name,prf)::(!rule_just));
   TextIO.output(TextIO.stdOut,
"New proof rule "^name^" has been accepted\n")))
   end;

(``X'' is the function which parses strings into hol_terms)
```

Clearly, these two rules are closely related in their impact on soundness, since the first rule merely applies the code "certified" by the second rule. They are built on top of two simple list structures which store the rules which have been certified and their proofs:

```
val valid_rules = ref []: (string*(term->term)) list ref

val rule_just = ref []: (string*tactic) list ref
```

2.1 Application Rule

Assuming we are able to satisfactorily establish that the extension rule only adds code/proofs to these lists if the code is a sound extension, then the application of the "USE_PR" rule is trivially (theoretically) sound. It does, however, raise two interesting related issues: misuse of the rule structure by the user, and the ability of the resulting system to support an isolated primitive proof checker.

Misuse Two sorts of misuse should be considered: accidental and malicious misuse. We consider the risks of accidental misuse extremely low. The user interface (essentially the traditional HOL interface) to the system is entirely via function applications; users would have no motivation to perform assignments on variables which they have not defined (since, even if they determined the need to use a "ref" variable to manage their work, they would reasonably expect assignment to an undefined variable to fail). Even if a user were to freshly define the variable, static scoping would prevent a conflict with the critical structure.

Malicious misuse, on the other hand, would be easily accomplished. As the system is currently implemented, a brief reading of the code (or this document) would provide the name of the key structure, improper code could be added by simple assignment. Even so, we consider the risk of (successful) malicious use very low:

1. A completed proof is not (currently) an economic commodity; any gain that is achievable in this way is unlikely to be sufficient motivation for the risk of personal embarrassment.
2. Checking that such a deception had been attempted would be quite trivial: any competent reviewer (who examined the proof closely) would note it, or a simple automated scan (e.g. grep) would be more than adequate to detect it.
3. The current implementation is essentially designed to be in "debugging" mode, since its primary purpose is to support this research project—two different techniques could be employed to dramatically decrease the feasibility of malicious misuse, if a production implementation were developed. The first is the straightforward application of the module feature in SML. By placing the basic system inside a module and only allowing the primitive rules to be publicly accessible (thereby hiding the key structures), it would force the malicious user to rewrite the system to achieve such a deception.

Of course, there are a wide variety of ways that a user could get a doctored (rewritten) system to appear to accept bogus proofs—this feature would be just one more target for this attack. Fortunately, the proposal for a split prover (primitive checker/proof assistant) would provide a more than adequate defense for this type of attack (and it's the next issue to be considered).

The Split Prover Finding techniques for creating such a system is still an active research area, recent work includes a proposal for a implementation of such system based on HOL. No effort, consequently, was made to solve a second open

problem or apply such techniques to this system. We claim, though, that these extensions would make the implementation of such techniques no more difficult.

The essential difficulty in the creation of a such a system is reducing an arbitrary proof to one that only makes reference to the primitive rules. Any invocation of one of the code rules by "USE_PR" could be be replaced by instantiating the theorem ($\forall$ t. t = f t) with the conclusion of the current goal, and looking up the proof of this theorem in the second data structure (using the name of the code as an index). Having the precise theorem and its proof should not only be sufficient, but more than is usually available to such a tool. So, while some simple coding may be needed to handle these rules, it is pretty trivial.

2.2 The Extension Rule

This places the obligation to meet the key condition for soundness (i.e. is the justification for these pieces of code sufficient to allow their use) squarely on the second rule. The rule, allowing for three assumptions, has a very straightforward justification. It requires that there is a proof that for any possible HOL term, the result of applying the ML code to that term is equivalent to that term. In any situation where the code rule would be used, a completely ordinary HOL proof is also available. Instead of applying the code rule, the system could instantiate the already proved theorem, and simply rewrite the conclusion of the goal. Since an equivalent primitive proof exists[3], the extension rule is a conservative extension.

The three assumptions that must be justified are: 1) that the parser correctly converts ML code into the appropriate internal form (abstract syntax) 2) the evaluation rules correctly define the operation of SML and 3) the user submits a function object which was created by the system evaluating the string ("text") submitted at the same time.

Parser Fortunately, the construction of a parser was performed with the assistance of automated tools (ML-Lex and ML-Yacc, provided in the SML/NJ distribution). The front end of the parser merely required the straight-forward encoding of the syntax from [5], while the back end ("code generation" of the abstract syntax) is an encoding drawn directly from the holML representation of the formal definition [7]. The tools helped identify a number of minor errors.

Formal Definition—Evaluation Rules The evaluation rules are drawn from HOL-SML, Myra VanInwegen's [7] encoding of the formal definition of SML [5]. HOL-SML includes only the Core of SML, and completely eliminates any of the rules on equality types, real numbers and input/output. A number of the differences between HOL-SML and Core SML '90 (e.g. value restriction versus weak typing,

[3] it may be necessary to reduce subordinate code rule applications to primitives to construct an actual primitive proof

restrictions on the use of the "ref" constructor) needed to make proofs about the language tenable have been incorporated into the SML '97 definition[4].

We have made two small changes to the definition: extending the default SML base environment with the types and constructors for modeling HOL terms (since we want to write SML code which modifies HOL terms); and coercing the variables in two of the pattern phrases into longvars to facilitate parsing. We have only converted the evaluation rule sections of the HOL-SML to the wHOLe system, the elaboration rules, as well as the proof type preservation have not been addressed (work on the determinism proof is underway).

The effort that went into those proofs, however, gives significant credibility to the accuracy of the evaluation rules. Not only was this formal, automated analysis of the SML definition able to achieve interesting results, but had influence on the development of the language. Conversion of the HOL-SML rules for the wHOLe system merely required some syntactic modifications (double quoted strings to represent HOL terms instead of single-quoted, a few more parenthesis to clarify scope of operators)—the difficult part was converting the supporting packages (which exposed numerous distinctions between the HOL '90 and HOLL/wHOLe systems).

The String and the Object While it is trivially easy (using a cut and paste editor) for the user to correctly submit matching text and executable objects, we would prefer to relieve the user of this obligation (and close a pretty large soundness hole). We expect that more careful study of the SML/NJ system will identify a mechanism for handling this detail.

3 USE OF THE SYSTEM

3.1 Basics

Briefly stated, wHOLe is a derivative of the original HOL system. Except for the implementation language, and the modifications described above, it is HOLL. The canonical features of such a system are: that interaction is via a dialect of ML (literally the Meta Language), that backward proof by tactics is the primary metaphor (although other schemes are supported), and the HOL object language (which is Higher Order Logic). Much more detail is available in [1, 4].

It is important to note, however, that HOLL was designed to be a reference version of HOL—the foundation for future research which could feed into system as it developed. HOLL does not include features like libraries or theory management & storage which would be preferred in a production system.

[4] Ironically, there was some trepidation when SML/NJ 93 implementation inadequacies forced this project to upgrade to SML/NJ 110 (and therefore, SML '97)—fearing that it would cause *more* language inconsistencies.

3.2 Extension

A helper function (called prove_SML) has been defined which takes a string (the text of some SML code), and invokes the HOL subgoal package on the theorem ($\forall$ t. t = f t) (where f is the SML code). At this point, the user can interactively apply tactics in the usual way, attempting to reduce the goal to statements that are known to be true. The user is obliged to keep careful track of the tactics employed to prove the goal, so that a single tactic which proves the goal can be constructed (e.g. "tactic1 THEN tactic2" is a single tactic which first applies tactic1, then applies tactic2 to the result).

The user must also compute the executable object equivalent to the text of their function. This can be done by using

> this "val" declaration: val my_sml_function = The Text of the SML Code;
> when the code looks like this: "The Text of the SML Code"

It will create an executable version of the object, and attach to it the name "my_sml_function". We recommend that this be done before the proof of correctness, since we find little value in proving properties of invalid SML code.

We anticipate that proofs of the kind we require will be greatly facilitated by the construction of a "symbolic evaluator" [6] which will generate the tactics to do the obvious case-splitting and rewriting for the analysis of SML code. We have deliberately put off the design of such a tool, however, until we have adequate experience performing such proofs.

4 DISCUSSION

4.1 How it Applies to Practice

Efficiency One of the long-time concerns with theorem proving tools is their speed. Because this tool provides a way to replace proof strategies based on primitive invocations with more efficient direct term manipulation, it should allow for a significant, although probably linear, speed-up.

Paradigm Unification An issue which has been pushed to the forefront by the development of algorithmic methods (e.g. model-checkers), is how to merge analysis tools of varying degrees of formal basis into a single methodology. This tool lays out the path to the ultimate solution to this dilemma: code verification that such an algorithm is truth preserving. While significant improvements to the support for and understanding of the method will be needed before such a proof could be attempted, at least the foundation has been laid.

Life-Cycle and Support The ability of SML/NJ 110 to import and run "C" code (via the CINTERFACE) offers a near-term solution for combining tools and a migration path for achieving the eventual goal. Initially, only the theorem prover would be implemented in SML, while any other tools would just be imported C. To increase the reliability of the tools, they could be a) translated to SML, b) shown to have certain key properties and c) shown to be truth preserving. This plan would allow one to selectively focus resources on the highest risk areas.

4.2 How it Applies to Research

Back-to-Basics Formal verification began as a research field with code techniques developed by Floyd, Hoare and Dijkstra. It is refreshing to be able to provide a modern level of automated support for the techniques on which the field was founded. Even better, the results of these verifications will not be just academic exercises, but can contribute to the development of the tools as well.

Self-Perpetuation The most appealing aspect of a system based on computational reflection via code proofs, though, is that successful application of the system will facilitate even more success. The user can add code rules to the system which will make it faster and more useful. This additional power can be used to incorporate the more sophisticated language features which have not yet been analyzed. The ability to employ more of the language will allow even more code rules to be accepted by the system.

5 ACKNOWLEDGMENTS

Great thanks to Matt Kaufmann, Myra VanInwegen, Richard Boulton, Elsa Gunter, John Reppy, Ian Soboroff, Konrad Slind, Donald Syme, John Harrison.

References

1. Michael J. C. Gordon and Tom F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, Cambridge, UK, 1993.
2. John Harrison. HOL light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.
3. John Harrison. A Tour of HOL light. Unpublished Draft, October 1996.
4. Laboratory for Applied Logic, Brigham Young University, http://lal.cs.byu.edu/lal/hol-documentation.html. *The HOL Theorem Proving System.* Web site featuring manuals, links to tutorials, on-line searchable libraries.
5. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge MA USA, 1990.
6. Donald Syme. Reasoning with the formal definition of Standard ML in HOL. In Jeffrey J. Joyce and Carl-Johan H. Seger, editors, *Higher Order Logic Theorem Proving and Its Applications (HUG'93); 6th International Workshop*, volume 780 of *Lecture Notes in Computer Science*, pages 43–58. Springer-Verlag, 1993. Vancouver, BC, CANADA.
7. Myra VanInwegen. *The Machine-Assisted Proof of Programming Language Properties.* PhD thesis, University of Pennsylvania, August 1996.