A Brief Overview of HOL4

Konrad Slind[1] and Michael Norrish[2]

[1] School of Computing, University of Utah
slind@cs.utah.edu
[2] National ICT Australia
Michael.Norrish@nicta.com.au

Abstract. The HOL4 proof assistant supports specification and proof in classical higher order logic. It is the latest in a long line of similar systems. In this short overview, we give an outline of the HOL4 system and how it may be applied in formal verification.

1 Introduction

HOL4 is an ML-based environment for constructing proofs in higher order logic. It provides a hierarchy of logical theories which serves as a rich specification library for verification. It also provides a collection of interactive and fully automatic proof tools which can be used in further theory building or in the provision of bespoke verification procedures.

Implementation history. The original HOL system was created in the mid-1980's when Mike Gordon at Cambridge University performed surgery on the Edinburgh LCF system, installing a version of Church's higher order logic as the object language of the system. The metalanguage in which the logic was encoded was Edinburgh ML, itself implemented on top of Lisp. An enhanced version of HOL, called HOL88 [6], was publically released (in 1988), after several years of further development. HOL90 (released in 1990) was a port of HOL88 to SML by Slind at the University of Calgary. The Lisp substrate was abandoned, and some of the more recondite underlying LCF technology was trimmed away or reimplemented. HOL90 ran on Poly/ML and SML/NJ. HOL98 (released in 1998) was a new design, emphasizing separate compilation of theories and proof procedures [12], thus allowing HOL to be ported to MoscowML.

HOL4 is the latest version of HOL, featuring a number of novelties compared to its predecessors. HOL4 continues to be implemented in SML; it currently runs atop Poly/ML and MoscowML. HOL4 is also the supported version of the system for the international HOL community [11].

Project management. The HOL project[1] is open source, managed using the facilities of SourceForge, and currently has about 25 developers, not all of whom are active. In general, control of user contributions is relaxed; anyone who wishes to make a contribution to the system may do so, provided they are willing to

[1] Located at http://hol.sourceforge.net

O. Ait Mohamed, C. Muñoz, and S. Tahar (Eds.): TPHOLs 2008, LNCS 5170, pp. 28–32, 2008.

provide support. However, modifications to the kernel are scrutinized closely by the project managers (the present authors) before being accepted.

2 Technical Features

We now summarize some notable aspects of HOL4.

2.1 Logic

The logic implemented by HOL4 is essentially Church's Simple Type Theory [3], with polymorphic type variables. The logic implemented by HOL systems, including ProofPower and HOL-Light, has been unchanged since the release of HOL88. An extremely important aspect of the HOL logic, not mentioned by Church, is primitive definition principles for consistently introducing new types and new constants.

An ongoing theme in HOL systems has been adherence to the derivation judgement of the logic: all theorems have to be obtained by performing proofs in higher order logic. However, in some cases, it is practical to allow external proof tools to be treated as oracles delivering HOL theorems *sans* proof. Such theorems are tagged in such a way that the provenance of subsequent theorems can be ascertained.

2.2 Kernels

As is common with HOL designs, the kernel implementation of the logic is kept intentionally small. Only a few simple axioms and rules of inference are encapsulated in the abstract type of theorems implemented in the logic kernel. Currently in HOL4 we maintain two kernels, one based on an explicit substitution calculus, and one based on a standard name-carrying lambda calculus. The desired kernel implementation may be chosen at build time. Informal testing indicates that each kernel outperforms the other on some important classes of input, but that neither outperforms the other in general.

2.3 Derived Rules and Definition Principles

Given such a simple basis, serious verification efforts would be impossible were it not for the fact that ML is a programmable metalanguage for the proof system. Derived inference rules and high-level definition principles are built by programming: such complex logical steps are reduced to a sequence of kernel inferences. For example, some of the current high-level definition principles for types are those supporting the introduction of quotient types and ML-style datatypes. Datatypes can be mutually and nested recursive and may also use record notation. At the term level, support is provided for defining inductively specified predicates and relations; mutual recursion and infinitary premises are allowed. Total recursive functions specified as recursion equations, possibly using ML-style pattern matching, are defined by a package that mechanizes the wellfounded recursion theorem. Mutual and nested recursions are supported. Simple termination proofs have been automated; however, more serious termination proofs have of course to be performed interactively.

2.4 Proof Tools

The view of proof in HOL4 is that the user guides the proof at a high level, leaving subsidiary proofs to automated reasoners. Towards this, we provide a database of type-indexed theorems (case analysis, induction, *etc*) which supports user control of decisive proof steps. In combination with a few 'declarative proof' constructs, this allows many proofs to be conducted at a high level.

HOL4 has a healthy suite of automated reasoners. All produce HOL proofs. Propositional logic formulas can be sent off to external SAT tools and the resulting resolution-style proofs are backtranslated into HOL proofs. For formulas involving $\mathbb{N}$, $\mathbb{Z}$, or $\mathbb{R}$, decision procedures for linear arithmetic may be used. A decision procedure for n-bit words has recently been released. For formulas falling (roughly) into first order logic, a robust implementation of ordered resolution has become very popular.

Probably the most commonly used proof tool is simplification. We provide a call-by-value evaluation mechanism which reduces ground, and some symbolic, terms to normal form [1]. A more general (and more heavily used) tool, the simplifier, provides conditional and contextual ordered rewriting, using matching for higher order patterns. The simplifier may be extended with arbitrary context-aware decision procedures.

For experienced users, most simple proofs can be accomplished via a small amount of interactive guidance (specifying induction or case-analysis, for example) followed by application of the simplifier and first order proof search.

2.5 Theories and Libraries

The system provides a wide collection of theories on which to base further verifications: booleans, pairs, sums, options, numbers ($\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, fixed point, floating point, n-bit words), lists, lazy lists, character strings, partial orders, monad instances, predicate sets, multisets, finite maps, polynomials, probability, abstract algebra, elliptic curves, lambda calculus, program logics (Hoare logic, separation logic), machine models (ARM, PPC, and IA32), temporal logics (ω-automata, CTL, μ-calculus, PSL) and so on. All theories have been built up definitionally.

HOL4 also has an informal notion of a *library*, which is a collection of theories, APIs, and proof procedures supporting a particular domain. For example, the library for $\mathbb{N}$ provides theories formalizing Peano Arithmetic and extensions (numerals, gcd, and simple number theory), a decision procedure, simplification sets for arithmetic expressions, and an extensive collection of syntactic procedures for manipulating arithmetic terms. Loading a library extends the logical context with the types, constants, definitions, and theorems of the comprised theories; it also automatically extends general proof tools, such as the simplifier and the evaluator, with library-specific contributions.

Both theories and libraries are persistent: this is achieved by representing them as separately compiled ML structures. A 'make'-like dependency maintenance tool is used to automatically rebuild formalizations involving disparate collections of HOL4 libraries and theories, as well as ML or external source code in other programming languages.

2.6 External Interfaces

There is a variety of ways for a logic implementation to interface with external tools. On the input side, as we have mentioned, purported theorems coming from external tools need to be accompanied with enough information to reconstruct a HOL proof of the theorem. An example of this is the interface with SAT solvers which can supply proof objects (we currently favour `minisat`).

Another approach is illustrated by the integration of a BDD library into HOL. This has been used to support the formalization and application of model-checking algorithms for temporal logic. Since HOL theorems are eventually derived from operations on BDDs representing HOL terms, the oracle mechanism mentioned earlier is used to tag such theorems as having been constructed extra-logically.

On the output side, HOL formalizations confining themselves to the 'functional programming' subset of HOL may be exported to ML. This gives a pathway from formalizations to executables. The generated code is exported as separately compilable ML source with no dependencies on the HOL4 implementation. Thus, the theory hierarchy of HOL4 is paralleled by a hierarchy of ML modules containing exported definitions of datatypes and computable functions formalized in HOL. We support the substitution of more efficient versions of such modules; for example, the GMP library used in the `mlton` compiler may be used instead of the relatively slow code generated from our theory of numerals.

Finally, higher order logic can be used as a metalogic in which to formalize another logic; such has been done for ACL2 [4,5]. HOL4 is used to show that ACL2 is sound. This allows a two-way connection between the two systems in which a HOL formalization may be translated to the HOL theory of ACL2, this formalization is then transported to the ACL2 system and processed in some way (*e.g.*, reduced using the powerful ACL2 evaluation engine) and then the result is transported back to HOL4 and backtranslated to the original HOL theory.

3 Current Projects

Network specification and validation. Peter Sewell and colleagues have used HOL4 to give the first detailed formal specifications of commonly used network infrastructure (UDP, TCP) [2]. This work has heavily used the tools available in HOL4 for operational semantics. They also implemented an inference rule which tested the conformance of real-world traces with their semantics.

Programming language semantics. As an application of the HOL4 backend of the `Ott` tool [14], Scott Owens has formalized the operational semantics of a large subset of OCaml and proved type soundness [13]. The formalization heavily relied upon the definition packages for datatypes, inductive relations, and recursive functions. Most of the proofs proceeded by rule induction, case analysis, simplification, and first order proof search with user-selected lemmas. In recent work, Norrish has formalized the semantics of `C++` [10].

Machine models. An extremely detailed formalization of the ARM due to Anthony Fox sits at the center of much current work in HOL4 focusing on the

verification of low-level software. The development is based on a proof that a micro-architecture implements the ARM instruction set architecture. In turn, the ISA has been extended with so-called 'Thumb' instructions (which support compact code) and co-processor instructions. On top of the ISA semantics, Myreen has built a separation logic for the ARM and provided proof automation [8].

Compiling from logic; decompiling to logic. It is possible to compile a 'functional programming subset' of the HOL logic to hardware [15] and also to ARM code [7]. This supports high-level correctness proofs of low-level implementations. As well, one can map in the other direction and *decompile* machine code to HOL functions with equivalent semantics [9].

References

1. Barras, B.: Proving and computing in HOL. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 17–37. Springer, Heidelberg (2000)
2. Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Rigorous specification and conformance testing techniques for network protocols, as applied to TCP, UDP, and Sockets. In: Proceedings of SIGCOMM. ACM Press, New York (2005)
3. Church, A.: A formulation of the Simple Theory of Types. Journal of Symbolic Logic 5, 56–68 (1940)
4. Gordon, M.J.C., Hunt, W.A., Kaufmann, M., Reynolds, J.: An embedding of the ACL2 logic in HOL. In: Proceedings of ACL2 2006, ACM International Conference Proceeding Series, vol. 205, pp. 40–46. ACM Press, New York (2006)
5. Gordon, M.J.C., Reynolds, J., Hunt, W.A., Kaufmann, M.: An integration of HOL and ACL2. In: Proceedings of FMCAD 2006, pp. 153–160. IEEE Computer Society, Los Alamitos (2006)
6. Gordon, M., Melham, T.: Introduction to HOL, a theorem proving environment for higher order logic. Cambridge University Press, Cambridge (1993)
7. Li, G., Slind, K.: Compilation as rewriting in higher order logic. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603. Springer, Heidelberg (2007)
8. Myreen, M., Gordon, M.: Hoare logic for realistically modelled machine code. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424. Springer, Heidelberg (2007)
9. Myreen, M., Slind, K., Gordon, M.: Machine-code verification for multiple architectures: An application of decompilation into logic. In: FMCAD 2008 (submitted 2008)
10. Norrish, M.: A formal semantics for C++. In: Informal proceedings of TTVSI 2008 (2008)
11. Norrish, M., Slind, K.: HOL-4 manuals (1998-2008), `http://hol.sourceforge.net/`
12. Norrish, M., Slind, K.: A thread of HOL development. Computer Journal 45(1), 37–45 (2002)
13. Owens, S.: A sound semantics for OCaml-Light. In: Proceedings of ESOP 2008. LNCS, vol. 4960. Springer, Heidelberg (2008)
14. Sewell, P., Nardelli, F., Owens, S., Peskine, G., Ridge, T., Sarkar, S., Strnisa, R.: Ott: Effective tool support for the working semanticist. In: Proceedings of ICFP 2007. ACM Press, New York (2007)
15. Slind, K., Owens, S., Iyoda, J., Gordon, M.: Proof producing synthesis of arithmetic and cryptographic hardware. Formal Aspects of Computing 19(3), 343–362 (2007)