HOL Light: A tutorial introduction

John Harrison

Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14a, 20520 Turku, Finland

Abstract. HOL Light is a new version of the HOL theorem prover. While retaining the reliability and programmability of earlier versions, it is more elegant, lightweight, powerful and automatic; it will be the basis for the Cambridge component of the HOL-2000 initiative to develop the next generation of HOL theorem provers. HOL Light is written in CAML Light, and so will run well even on small machines, e.g. PCs and Macintoshes with a few megabytes of RAM. This is in stark contrast to the resource-hungry systems which are the norm in this field, other versions of HOL included. Among the new features of this version are a powerful simplifier, effective first order automation, simple higher-order matching and very general support for inductive and recursive definitions.

Many theorem provers, model checkers and other hardware verification tools are tied to a particular set of facilities and a particular style of interaction. However HOL Light offers a wide range of proof tools and proof styles to suit the needs of various applications. For work in high-level mathematical theories, one can use a more 'declarative' textbook-like style of proof (inspired by Trybulec's Mizar system). For larger but more routine and mechanical applications, HOL Light includes more 'procedural' automated tools for simplifying complicated expressions, proving large tautologies etc. We believe this unusual range makes HOL Light a particularly promising vehicle for floating point verification, which involves a mix of abstract mathematics and concrete low-level reasoning. We will aim to give a brief tutorial introduction to the system, illustrating some of its features by way of such floating point verification examples. In this paper we explain the system and its possible applications in a little more detail.

1 The evolution of HOL Light

In verification tasks, low-level proof checkers are often too tedious to use, while fully automatic provers are seldom effective. The ideal seems to be a judicious combination of interaction and automation. Edinburgh LCF [4] was an influential methodology for achieving this sort of balance. Low-level primitive inference rules are provided which produce theorems, and by the use of an abstract type, it is ensured that theorems can *only* be created by these rules. However the user is free to write arbitrary proof procedures in the metalanguage ML which decompose to these inferences; such procedures can in principle implement practically any automated proof method (e.g. resolution, Presburger arithmetic), and they will be correct per construction, in the sense that they cannot produce a false 'theorem'. Experience suggests that this can usually be done reasonably efficiently. Hence LCF systems offer a unique combination of reliability and programmability.

Paulson [8] and Huet improved Edinburgh LCF, and based on their implementation, Gordon produced a new system based on classical higher order logic rather than Scott's Logic of Computable Functions. This system, HOL, was specifically designed for hardware verification, the use of higher order logic for this purpose having been advocated by Hanna [5] and by Gordon himself [2]. HOL became increasingly popular in the late 80s and early 90s, attracting many users worldwide in academia, industry and military and government agencies. However the `HOL88` version was a conceptually complicated and inefficient system, with many of the primitive term operations coded in LISP rather than ML. A new version, `hol90`, corrected these deficiencies, but did not greatly improve the higher level organization or the theorem-proving technology; HOL proofs are often long and difficult compared with those in other contemporary systems like Isabelle [9] and PVS [7]. Moreover `hol90` is at present tied to the New Jersey SML compiler, which is notoriously memory-hungry; this means that it is still slow or even unusable on typical machines.

HOL Light represents an attempt to improve the HOL system more radically, while retaining its traditional strengths. The system is small and light, using the excellent CAML Light interpreter [12] developed at INRIA Roquencourt. The logical development has been cleaned up, and new and powerful proof techniques added. It even supports a new proof *style*, based on Mizar, which allows many proofs to be written more elegantly. Nevertheless, we want to stress that although HOL Light is a recent development, it can be seen as the culmination of a successful line of research spanning about 20 years.

2 Highlights of HOL Light

HOL Light has the following features:

1. It is open. The entire system is coded in ML and all the source code is freely available. Because ML is a fairly readable and high-level language, the implementation often looks quite close to an abstract algorithm description. Hence users can see what is happening inside and gain real confidence and understanding; the system is not just a mysterious black box.
2. It is sound and coherent. The LCF methodology ensures that all extensions with custom proof procedures are correct by construction. Apart from the benefits of soundness, this also helps to make the structure of the system logically clean and comprehensible, rather than its being a complicated entwining of different proof procedures.
3. It is extensible. If users want to implement special purpose tools and decision procedures, they can implement them, and the LCF methodology ensures that they cannot produce false results. The system source code contains numerous examples of special proof procedures, ranging from the simple to the complex.

4. It is an easy matter to write special interface code and connect HOL Light to other systems. For example, we have used the Maple computer algebra system to perform factorization and integration steps, and an implementation of Stålmarck's algorithm [10] to prove tautologies; HOL Light can check the validity of these results internally, maintaining the usual guarantee of correctness even in a mixed system.
5. It is small and lightweight. The system does not require a state-of-the-art workstation to run. CAML Light is very economical and can work quite well on small machines such as PCs and Macintoshes with just a few megabytes of free memory. Nevertheless, we aim to show that the system is not merely a toy, but is capable of being used for real verification tasks.
6. Different proof styles are supported. Though the 'machine code' of the system is a simple set of forward inferences, one can interact with the system in various different high-level ways to prove theorems. For example, one can prove theorems in a backward fashion using tactics, or write a more orthodox mathematical proof in the style of Mizar [11]. A facility for window inference, useful for program refinement and other transformational design methodologies, is under development. All these styles can be intermixed freely in the same script, and even in the same proof.
7. Special-purpose proof procedures are available. In many situations, the user will find that the system already contains tools to handle tricky steps. For example, there is code to automate the definition of inductive relations and types as well as recursive functions with arbitrary well-founded measures. Support for pure logic includes several complementary tautology-proving modules and proof search procedures for first order logic, while there are several domain-specific tools such as decision procedures for linear arithmetic (over naturals, integers and reals).

3 HOL Light in hardware verification

The advantages of theorem-proving techniques over model-checking for hardware (and software) verification are:

1. Specifications can be written in an elegant way using the normal resources of mathematics.
2. Any supporting mathematical theories required can be called upon, after themselves being formally proved, in the verification effort.

We will demonstrate how HOL Light can be used for a particular example: we plan to show the proof of correctness of a CORDIC algorithm for evaluating floating point natural logarithms. Though only a single example, we hope it illustrates many interesting features of the HOL Light system, and the use of theorem provers for verification in general. Floating point correctness is a topical issue, with the Pentium fiasco still a recent memory; moreover, it seems that an incorrect treatment of overflow in a floating-point to integer conversion routine was responsible for the destruction of the European Space Agency's Ariane 5 rocket on its maiden flight.

The first step is to establish what it means for the floating point algorithm to be 'correct'. The natural idea is to specify that the result, when interpreted as a real number, corresponds closely to the true mathematical value. We do indeed choose such a specification, but point out how there are several subtly different ways of making 'corresponds closely' more precise. For the elementary algebraic operations, including square roots, the IEEE Standard [6] mandates that the result should be *the closest representable* value to the true mathematical value (with choices between equally close values resolved using a convention of 'round to even'). For transcendental functions, this is very difficult to achieve in practice, because of a phenomenon known as the 'table maker's dilemma'. We explain two alternative forms of correctness specification that we consider more suitable for the transcendental functions. It is against the second of these that our verification is performed.

The next task is to model the floating point algorithm itself inside HOL Light's logic. It is possible to use a gate-level circuit description, but a more transparent description written in a formalized 'pseudocode' is used here. This gives a good opportunity to discuss techniques which have been used for semantically embedding idealized programming languages [3] and real hardware description languages [1] in HOL. These techniques use a special kind of denotational interpretation of the language as ordinary logical and mathematical constructs, and allow a rather direct means of formal reasoning about programs. Our system description is, therefore, both readable and rigorous.

The verification rests on a number of properties of logarithms. To give a simple example, we use the fact that $ln(1 + x) \leq x$ whenever $x \geq 0$. The ultimate foundation for these results is a rigorously developed theory of real analysis, based on a definitional construction of the real numbers from the natural numbers. We cannot give many details here, but we do hope to give some illustrations of how HOL Light can be used in this more abstract field. In particular, its Mizar-style proofs allow one to write proofs in a rather elegant style (compared with the arcane tactic scripts of most systems, other HOL proof styles included). At the same time, the automated facilities of HOL Light can be called on when required, e.g. to avoid directing routine algebraic steps in detail.

The overall proof is quite intricate, and though it can be rerun in a few minutes, we will concentrate on the general feel of the theorem proving facilities provided by HOL Light. We show how the simplifier can be used extensively to avoid complex and tedious chains of reasoning. Indeed, in a related verification effort of a square root algorithm, the central mathematical invariant property could be proved completely automatically. However we stress that low-level facilities are always there to provide a finer degree of control.

Finally, we show that certain precomputed constants, required for a real implementation, can be calculated in HOL Light using the normal inference rules of the logic. Though slow, this is acceptable in such special circumstances, and gives a cast-iron guarantee on the error bounds.

4 Conclusions

We hope to illustrate that HOL Light can be applied to realistic verification tasks. Moreover, the method of embedding other formalisms, together with the inherent programmability, make HOL Light an excellent core for application-specific reasoning systems.

References

1. R. Boulton et al. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions A: Computer Science and Technology*, pages 129–156, Nijmegen, The Netherlands, 1993. North-Holland.
2. M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. Technical Report 77, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1985.
3. M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.
4. M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
5. F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: A case study. In G. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 179–213, 1986.
6. IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.
7. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.
8. L. C. Paulson. *Logic and computation: interactive proof with Cambridge LCF*. Number 2 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1987.
9. L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. With contributions by Tobias Nipkow.
10. G. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula. United States Patent number 5,276,897; see also Swedish Patent 467 076, 1994.
11. A. Trybulec. The Mizar-QC/6000 logic information language. *ALLC Bulletin (Association for Literary and Linguistic Computing)*, 6:136–140, 1978.
12. P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993. See also the CAML Web page: `http://pauillac.inria.fr/caml/`.

This article was processed using the LaTeX macro package with LLNCS style