A Reference Version of HOL

John Harrison[1] and Konrad Slind[2]

[1] University of Cambridge Computer Laboratory,
New Museums Site, Pembroke Street, Cambridge, CB2 3QG, England.
[2] Institut für Informatik,
Technische Universität München, 80290 München, Germany.

Abstract. The second author has implemented a reference version of the HOL logic (henceforth called gtt). This version, written in Standard ML, is as simple as possible, making as few assumptions as necessary to present the essence of HOL. This simplicity makes the implementation easy to understand, to port, to develop, to change, and to informally reason about. The first author has ported gtt to another dialect of ML, and developed the parsing, prettyprinting, and typechecking support needed to take gtt beyond its initial rudimentary conception. The implementation of gtt has already been of use in developing a variant of the HOL logic.

As of this writing, there are at least four or five extant implementations of the HOL logic. These have been intensively developed, in some cases over decades, which leads us to an overwhelming question: why another? In particular, why gtt? There are several answers to this, stemming from different desires and needs in the HOL community.

Changing the logic a little bit. Some authors [Mel92, Bou92, Sli92, Gun91] have recently proposed extensions or alterations to the HOL logic. Such modifications are difficult to implement in HOL88 because such fundamental changes need to be performed in the "sea of Lisp" that underlies HOL88. Such alterations are much easier to do in hol90 because it is a far better structured system that is furthermore written all in one language; still, some aspects of the core of the hol90 implementation are difficult to modify. The gtt implementation provides a particularly simple setting to prototype changes that might eventually migrate into a real implementation of HOL. Later in this paper we give an overview of one such development.

Changing the logic a lot. Another conceivable use stems from noticing how small and simple the primitive inference rules are, given the prelogic. Someone who wanted to implement a logic other than HOL on top of a lambda calculus with ML-style polymorphic types could use the gtt prelogic as an 'implementation-springboard'. Although logical frameworks such as Isabelle would seem to have cornered the market in this area, the gtt style of implementation offers finer control, and this can be useful in situations where the object logic embeds awkwardly into the logical framework. For example, a dependent type theory is difficult to model effectively in Isabelle, since one would want to directly use the contexts of Isabelle to implement the contexts of the object logic; however, an Isabelle context is basically a set (of lambda terms), and that is too simple to capture some aspects of type theory contexts. Alternatively, one could build a faithful type theory implementation directly on top of the gtt prelogic. This example suggests that, for the purposes of quickly implementing a logic, perhaps the issue of whether inference rules are lambda terms or ML programs is less important than the provision of an expressive prelogic plus general tools such as extensible parsers and pretty printers, parameterized rewriting engines, and other specializable reasoning tools. For discussion of these issues from a somewhat different perspective, see [CH90].

Exploring important implementation issues. One interesting research field is the investigation of representations of the lambda calculus for theorem provers (some of the choices are explored in [Hue89]). As one example of the tradeoffs to be considered, de Bruijn terms simplify substitution and allow quick tests of alpha-equivalence; however, they also generate more garbage, since the breaking apart of a lambda abstraction involves a full term traversal to replace the bound variable with a free variable. Here the benefit of gtt's simplicity is that changes to the implementation can be more easily effected since the scope of such changes is smaller. Also there is the methodological point that different approaches can be more easily compared in such a simple setting. The downside of gtt currently is that is not a large body of examples for evaluating such changes and deciding whether they ought to be propagated to a real implementation. However, such changes, if they work in the reference version, are probably simple to propagate to a cleanly written real implementation.

Portability. Although there are several good implementations of Standard ML available, the compiler technology for this language has not yet developed to the point that reliable ports to new architectures can be achieved in a timely fashion. Spread of the many formal methods tools that have sprung up around ML is thus impeded. Furthermore, the size of a real HOL system is a genuine problem in making HOL-based verification tools widely available. Not everybody can afford a high-powered workstation (or equally importantly, the memory) to run HOL on. Therefore, a simple but perhaps limited version of HOL that is easily portable to 'ML-like' languages might help increase the HOL user base.

Teaching. gtt is a concrete, graspable, and executable answer to the question "what is an implementation of HOL"? Possibly gtt can be of use to those wishing to teach aspects of higher-order logic and theorem proving. Another use would be to provide an executable introduction to the essence and structure of LCF implementations in general (like [Gor82]), and the hol90 implementation in particular. An analogous development for the Calculus of Constructions can be found in [Hue89].

Correctness. Verification tools ought to be correct. gtt is certainly small enough to reason about informally, and that is of fundamental importance when developing an implementation. However, we would like to go further and assert that gtt is simple enough to reason about formally. The recent work of von Wright [vW94] is encouraging in this regard, although we caution that verification of gtt must deal with code that uses exceptions and reference variables: the former are frequently used in the implementation and the latter are required in order to implement the symbol tables (*signatures* in standard logical parlance) for types and terms.

The rest of this paper divides up as follows: we first describe LCF-style theorem provers and list various shortcomings common in HOL implementations; then the basic implementation of gtt is explored, plus additions to make it usable. After that, we discuss a port of gtt to a different dialect of ML. Then we mention some aspects of how gtt was used in the development of an extension of the HOL logic. The rest of the paper is a discussion of a grab-bag of ideas and issues that arose in the course of this work.

1 The ideal and the real

An idealized LCF-style implementation provides three basic modules upon which the entire rest of the system is developed by secure definitional extension and ML programming:

1. A logical engine, dealing solely with terms, types and theorems, including the primitive inference rules and principles of definition. This kernel should be totally independent of the other two modules.
2. Support for organizing logical developments, maintaining their interdependencies, and storing/retrieving them in a safe manner in the file system. Such *theories* are stored either using a custom file format or perhaps by exploiting facilities for persistent storage in the ML implementation.
3. A front end, including term and type parsers and prettyprinters as well as (perhaps) a typechecker for terms.

Current HOL implementations, including hol90, but perhaps excepting ProofPower, which we are not adequately familiar with, deviate somewhat from the ideal. Following are some examples of how these notionally orthogonal modules are conflated in current HOL implementations.

– The function `new_specification` performs definitional extension of the logic with new term constants. But apart from the existential theorem justifying the definition and the name of the constant, it expects a name under which to store the definition on disk, as well as indications of the parse status of the new constant(s).

– The ML type of HOL types typically includes more than just type constructors and variables. For example, in hol90 this type has separate fields for 'user' and 'system' type variables and utilizes reference cells for efficient implementation of unification by pointer redirection. These extras are only used for typechecking, so interface support is mingling with the logical operations inappropriately.
– A further weakness is that a term constructed by parsing is dependent on the correctness of the typechecker for its well-typedness. Thus a bug in the typechecker (which is notionally separate from the logic) could cause the construction of badly-typed 'terms'. This can be seen as compromising the integrity of a HOL implementation. However, the HOL implementations we are familiar with are not dependent on the parser and typechecker for terms, so the paranoid user can always use the term constructors directly or write their own parser and typechecker.

2 The kernel of gtt

Since the authors were interested in isolating the essence of HOL, the core implementation of gtt intentionally lacks almost all the features that HOL users are accustomed to. However, we have been careful to ensure that all such features can be built on top of the core. To make a long story short, in the kernel there is no notion of parsing or prettyprinting of the objects defined in the kernel; there is no quotation or antiquotation; there is no type inference; there is no notion of theory (internal or on disk); no term or type constants have been hardwired in; no definitions or axioms have been predeclared; and there are no derived rules, conversions, or tactics.

What then *does* it have? The core of gtt comprises a collection of support functions, the implementations of types and terms, a few derived syntactical operations, the encapsulation of the basic rules of inference in the abstract type of theorems, and the three basic principles of definition. The gtt kernel is coded in a style that favours clarity over efficiency. We have, however, not omitted any of the error checking of real versions of HOL, since that is an essential part of the behavioural specification of the system. Therefore gtt is *not* a toy implementation: it may be naively coded in some parts, but its kernel is completely adequate with respect to its expected behaviour, as one would demand of a system that claims to be a reference.

The implementation of the gtt kernel comprises approximately 750 lines of functorized SML. We now go through the modules that make up the system.

2.1 Utilities

This is a small library of utility functions, including the definition of a uniform error facility. Some of these functions, e.g. `itlist`, already exist in various ML dialects, though usually under different names (e.g. `fold` or `it_list`). Collecting them all together aids portability and helps to make dependencies explicit. Approximately 30 functions are defined here, and their source takes up about a quarter of the total size of the system (measured in lines of code).

2.2 Types

This module builds the abstract type of `hol_type`s, together with the constructors and destructors and a function to substitute types for type variables in another type. The module includes a reference variable which holds the current type constructors and their arities (the *type signature*). As we mentioned, in the core there is no notion of concrete syntax; therefore, type variables and constructors can be any ML string. This simplifies the logic of the implementation a great deal, but not too much: the construction of a compound type still involves a check that the arity of the type constructor is respected. About a tenth of the system size comes from the definition of types.

2.3 Terms

This module builds the abstract type of `term`s, together with constructors, destructors, substitution functions, *etc.* This includes a reference variable which holds the current term constants and their types (the *term signature*). Some notable details about the implementation of `term` follow.

1. As in hol90, (a slight modification of) de Bruijn terms [dB72] are used, since they give simple implementations of the operations of substitution and type instantiation. (On the other hand, the constructors and destructors for abstractions become a little more complex.) Unlike substitution in current implementations of hol90, *no* renaming is done: with de Bruijn terms this is not necessary from a logical point of view and it is possible to implement renaming versions of *subst* and *inst* outside the kernel. The benefit of this is that a user-interface issue (renaming for readability and re-parsability) is separated from the logical operation of substitution.
2. The `mk_const` function has been simplified. It now takes a type substitution to apply to a constant's generic type, rather than using matching to find the desired substitution. This small change frees the kernel from the relatively sizable code implementing matching. The function is much more efficient and often easier to use. For example, when constructing a universally quantified term, it is just necessary to substitute the type *ty* of the bound variable for the universal quantifier's type variable α, rather than build up the entire type of the quantifier, namely $(ty \rightarrow bool) \rightarrow ty$. Matching, and thus a conventional `mk_const`, can be provided external to the core.

3. As for types, there is no restriction on the lexical form of variable and constant names.
4. We do not need to include the 'renaming' functions `mk_primed_var`, `genvar`, and `variant` in the `gtt` kernel. (One of the reasons for this is that substitution does no renaming.) This is fortunate, because these functions make some assumptions about the concrete syntax, *e.g.*, that a prime mark (`'`) is an allowable part of a variable name. More important is the fact that the properties achieved by renaming are notoriously complex to specify and reason about. For example, the logical definition of `variant` is an interesting exercise in wellfounded recursion. In a formalized version of the system, all functions would be definable using abbreviation or primitive recursion (and without any of the contortion needed, for example, in defining Ackermann's function primitive recursively). This gives strong evidence for the simplicity of `gtt`. *Caveat*: we are ignoring how to model exceptions and references in this assertion.

The term module takes up about a quarter of the `gtt` implementation.

2.4 Derived syntax operations

This small module contains derived term operations for the primitive logical constants for implication (`==>`) and equality (`=`). This is the minimum required to implement the remainder of the kernel.

2.5 Theorems

This module provides the abstract type of theorems and the primitive inference rules: `REFL`, `ABS`, `SUBST`, `ASSUME`, `DISCH`, `MP`, `SUBST` and `INST_TYPE`. Many HOL implementations provide some additional primitive rules for the sake of efficiency, but we limit ourselves to the standard set of rules. Given the functions available in the preceding modules, the inference rules find very easy and compact expression. The majority of the code is in fact taken up with checking side conditions and issuing the right error messages. The module of theorems takes about an eighth of the implementation.

2.6 Principles of definition

This module implements the three basic principles of definition for the HOL logic: *constant definition* (defining a constant equal to an existing term), *constant specification* (defining a constant as a witness for an existential theorem) and *type definition* (defining a new type in bijection with a nonempty subset of an existing type). We also supply the `new_axiom` function, since at this point, we have not yet introduced the axioms of the HOL logic (they depend on definitions that have not yet been made). We have simplified the principle of type definition so that it no longer has a redundant argument (the pattern argument essentially duplicates information already present in the justifying theorem). Dropping the checking of parsing information from these functions makes them markedly simpler. The code for the principles of definition takes about a fifth of the implementation.

2.7 What about theories?

The elimination of any notion of theory hierarchy achieves, at a stroke, an enormous simplification in the implementation. Since gtt has no concept of the external storage of theorems (*i.e.*, theory files), users must rely on the facilities of ML (top level bindings and perhaps `structure`s). If the system were to be used for real work, more support might be required, although systems like Isabelle and ProofPower seem to do just fine without a custom notion of theory file. Such support can in any case be implemented outside the logical core, although in the absence of the ability to safely export and import ML binaries to disk, the core of gtt might have to be extended so that theorems stored externally in ASCII can be safely brought back into the image.

Conclusion

We have now finished reviewing the implementation of the gtt kernel. The current status of the original SML implementation of gtt is that on top of the core, we have performed the definitions of the basic logical operators and asserted the axioms of HOL. A derived implementation in another dialect of ML has been taken a bit further, and will be discussed in the next section. Our belief is that gtt is small enough to 'fit in the palm of one's hand': prospective changes to HOL can be implemented and experimented with in an environment providing none of the distractions of a large system. We will discuss such a development in a subsequent section.

3 A front end

Now we turn to the provision of support for higher levels of interaction with gtt; without this, the system is unusable. In gtt we have fully disengaged the front end from the underlying logical core. Most of this was achieved by paring down the existing hol90 system, although the parser was written from scratch. Some code in hol90 was also improved in the light of our reexamination. The front end support for gtt although somewhat meager, has been sufficient to exercise the implementation on some preliminary examples. The front end separates into type checking, parsing, and prettyprinting.

3.1 Typechecking

First we define 'preterms' and 'pretypes' which are used as an intermediate representation for the parser and typechecker. The notion of pretype achieves complete separation from the logical engine of typechecking operations. The typechecker implements the Hindley-Milner algorithm with side-effecting unification and is a slightly more elegant version of the one in hol90. Various functions are provided to translate into true types and terms.

3.2 Parsing

It is at this stage that we make choices about the concrete syntax of HOL. We developed a simple lexical analyzer based on the SML model (*i.e.*, separate classes of 'alphanumeric' and 'symbolic' identifiers). The lexer output is fed to a recursive descent parser constructed using higher-order combinators. A handwritten parser is far more portable than an automatically generated one, and is certainly much more readable (although it is less readable than the ML-Yacc grammar of the hol90 parser). The concrete syntax of types and terms has been slightly rearranged to be simpler than in hol90, and three useful (and compatible) changes have been made.

1. Any term (subject to some mild restrictions) is allowed as the *varstruct* in a binder, which both regularizes the syntax and allows more general notions of matching (the special status of pairing rightly disappears).
2. Any alphanumeric identifier is permitted as a type variable, which allows much more readable names. Resolution of type constants and type variables is subsequently made on the basis of the symbol table, exactly as for term constants and term variables.
3. A new parsing class of *prefix* is introduced, for unary operators which are right associative, as opposed to the usual left. In current HOL implementations negation is treated this way on an *ad hoc* basis. A more general facility might be useful *e.g.*, for modal operators[3].

As yet, quotation and antiquotation are not supported in the lexer and parser: we regard this as primarily an ML issue; it is simple to adjust the front end if the ML provides quote/antiquote, relatively involved otherwise.

3.3 Printing

We initially used a simple prettyprinter which hardly deserved that appellation. It was nonetheless enough to make the system tolerable to use. In

[3] This was suggested by Ching-Tsun Chou

general, we find that a little prettyprinting goes a long way! More recently, the front end has been augmented with a prettyprinter which behaves very similarly to the ones in mainstream versions of HOL. This augmentation was developed by Richard Boulton using a port of the prettyprinting engine from his HOL88 library, in conjunction with a prettyprinter generator.

4 Portability

The gtt system has already been developed in parallel SML and CAML Light versions, with some work being done first in SML, some in CAML. CAML Light [Mau93] is a compact, well-engineered implementation of ML which, in stark contrast to any implementations of SML or 'Classic' ML, runs well on small machines. Apart from the interest of porting HOL to other languages, a CAML port opens up a number of new possibilities.

- CAML Light currently runs on a wider range of machines than SML. This makes it much easier to test on different architectures. For example, we have already run the system on a DEC Alpha. This observation is reminiscent of one of the advantages of RISC over CISC: by building a simpler system we can quickly take advantage of new technological developments.
- CAML is an influential language in France, and the availability of a CAML port might help to generate more interest in HOL in that country. Furthermore CAML is almost upward-compatible with the Classic ML as used in HOL88, and ISWIM traditionalists may find it more congenial than the somewhat 'heavier' syntax of SML.
- As mentioned above, CAML Light can be used on quite small machines, for example an 8086 PC with at least 640K of memory. By contrast, SML/NJ requires at least 16M of memory to run in a way that can be called satisfactory, and if using a PC you would need at least a 386-based system. This means that many researchers will be able to run gtt not just on the state-of-the-art workstation on their desk, but on the old toy computers they have at home. And furthermore, in many countries (*e.g.*, Russia) state-of-the-art workstations are almost unheard of but PC systems are common.

One of the great merits of gtt is that it provides a simple platform for experimentation with new logical ideas or implementation methods. On the debit side, its very smallness militates against meaningful benchmarking since there just isn't a substantial suite of proof scripts to try. Nevertheless we believe it should be straightforward, if tedious, to port firstly the derived system and then proofs, from HOL88 or hol90. The following tasks remain before it is possible to port over derived rules and theories:

– implement matching,
– supply a standard `mk_const`,
– add all the derived syntax operations,
– supply standard versions of definition principles, *i.e*, taking parsing status of introduced constants into account, and
– implement simple in-memory theories, as an organizational aid.

Many of these have already been done in the CAML Light version at the time of writing. New alternatives have also been attacked, such as higher-order rewriting. We discuss this later in the paper.

5 New logical ideas

In this section, we discuss a list of possible changes to the underlying logic which have been suggested in the past but ran out of steam because of implementation complexity. With gtt, making most of the changes described below would be simple (though the theoretical ramifications might be complex). The first possibility has recently been implemented, in fact by using gtt. The others have not, as yet, made their way into public distribution.

5.1 Type quantification

In very recent work, Tom Melham and the second author have used gtt as a springboard to a full HOL system that implements the type quantification proposal put forth in [Mel92]. Briefly, this proposal involves supporting an ability to universally quantify, in the term language, over type variables. Implementation of this would require extensive modification to the kernel of, say, hol90, so instead it was decided to modify the implementation of gtt.

In all, there were three implementations completed, in less than one month of time:

1. A deBruijn version, where quantified type variables were represented with numbers. No renaming is performed in substitution. This allowed the fundamental aspects of the representations of types and terms to be quickly implemented. The development of this did not progress beyond the kernel.
2. A deBruijn-with-renaming version. This was as the previous implementation, except that renaming would occur if a substitution would result in the textual capture of an incoming variable. This was the most complicated part of the effort, since there are two levels of capture that can occur. This version did not progress past the implementation of the kernel and extensive testing of the prelogic, especially for renaming behaviour.

3. A full-blown HOL version, based on the previous version, but with support for type-quantified terms extended to parsing and prettyprinting, plus all the derived inference facilities of HOL such as rewriting and forward-chaining.

Perhaps this implementation could have been directly achieved in the same amount of time by modifying an existing HOL implementation, but being able to focus on the appropriate issues at the appropriate times in development, *without clutter*, was found to be useful.

5.2 Other possibilities

The following proposals are loosely ordered in terms of how thoroughly they have been thought through. In fact, the first few items have already been implemented in some form or other; we include them merely for completeness.

1. Lazy theorems could be added, as proposed by Richard Boulton [Bou92].
2. Notions of abstract theory [Win92] could be directly supported, as in [Gun91]. For example, a theorem could have two separate assumption lists, one of them considered as an abstract context. The intended interpretation would be exactly the same (as if the two assumption lists were unioned together). This could give considerable organizational benefits without elaborating the system's simple semantics.
3. Theorems could be tagged with 'unsafe' flags, indicating for example that a primitive version of a supposedly derived rule is used, or that some external tool has been used to derive the 'theorem'. This is an elaboration of a scheme proposed by Mike Gordon and described in [Gor93]. More sophisticated tagging schemes might allow precise tracking of theorem interdependencies.
4. Alternative representations for numerals could be experimented with. This idea has been a preoccupation of the first author for some time. At present, numerals are implemented as logical constants, and all arithmetic based on a primitive rule `num_CONV` which evaluates the successor function. This has the dual defects of being hopelessly slow for large numbers and relying on the accuracy of the native ML integer arithmetic (of which more below). An interesting alternative is to use some binary representation inside the logic, and present numerals to the user only by prettyprinting.
5. It is very useful in certain applications, *e.g.*, program semantics, to use more sophisticated varstructs in lambda-abstractions than just single variables. A HOL library written by Jim Grundy exists to duplicate all the usual proof facilities, but for paired abstractions. It would be interesting to incorporate this into the core, as has been done in ProofPower. Even more general forms of varstruct are allowed by the new parser (see above) and it

would be interesting to explore the extent to which these can be supported automatically. We have in mind a logical interpretation of:

$$\lambda(E[x_1, \ldots, x_n]).F[x_1, \ldots, x_n]$$

into the following primitive logical form:

$$\varepsilon f.\ \forall x_1, \ldots, x_n.\ f(E[x_1, \ldots, x_n]) = F[x_1, \ldots, x_n]$$

A simple modification to this, to allow several possible varstructs would allow `CASE` statements and readable pattern matching inside the logic. An appropriate semantics for sequential matching is quite simple.

6. The primitive rule `SUBST` could be supplanted by a number of simpler rules, *e.g.*, `MK_COMB`, `SYM`, `TRANS` and `EQ_MP`. This seems like a lot of new primitive rules, but each of these are very simple, and we believe that the resulting system would be more amenable to formal analysis.
7. The principles of definition could be changed. Currently the principles of constant definition and specification are very nearly interderivable. Specifically, any application of `new_definition` can be replaced by a derived use of `new_specification` (since the theorem `|- ?x. x = t` is always provable if `t` is closed) *except* the definition of the existential quantifier itself! And using ε-terms, constant specifications can be defined as HOL terms, but unlike the constants returned by `new_specification`, it is possible to prove equalities between terms based on the same predicate. It seems crude to have two principles of definition, and that any of them should rely on derived constants (the existential quantifier and the constant `TYPE_DEFINITION`). In principle there is a transitory phase before the existential quantifier has been defined when HOL is unsound, as one could define that quantifier differently and then abuse `new_specification`. One solution would be to allow the introduction of a new constant `c` and theorem `|- P c` given only `|- P t` for a closed term `t`. This has the merit of subsuming the two principles of term constant definition, and helping to separate the assumption of the Axiom of Choice from the principles of definition.

6 New implementation ideas

There are various investigations which could be undertaken regarding the implementation. We do not claim any originality in these ideas: we imagine that many others have had thoughts along these lines.

1. The assumptions could be implemented as a more efficient data structure than a list. This might have a big impact on the efficiency of certain derived decision procedures, as described elsewhere [Har94]. It is likely that such a modification would imply a change in assumption ordering, and cause some existing proofs (especially those relying on `FIRST_ASSUM`) to break.

2. The use of *annotations*, as proposed by Sara Kalvala [KAL92], could be supported. These allow the maintenance of logically irrelevant but practically useful information in the theorem or proof structure. Possible applications include documenting proofs and controlling prettyprinting of embedded languages.
3. Structure sharing techniques could be tried. At one extreme, global hash-consing is possible, though to keep structures garbage collectable it should be done inside the ML compiler. Reasonable variants might include sharing only types (garbage collection is less of a problem here because types are not often created then completely discarded), only variables, or only terms in theorems produced via `PROVE`.

   In the case of types, the HOL88 system stores types at all nodes in a term. By contrast hol90 stores them only at leaves (i.e. constants and variables). This means that hol90 is more space-efficient but that finding the type of a term is more expensive. For example, building up a term tree via the abstract constructors `mk_comb` and friends (which check types independently on each call) can quadratic time usage in the size of the tree, because at each constructor application it is necessary to traverse (some of) the subtrees to find their types. It would be interesting to see if sharing types would tip the balance back in favour of the HOL88 method.
4. There are interesting possibilities for partial evaluation, or so the people who do partial evaluation tell us [Lau90]. The simplicity of `gtt` is such that it should port easily to a language with partial evaluation; however, we caution that the benefits of partial evaluation will need to be measured in a full-scale HOL implementation.
5. A small system might be a better target for proof recording as has already been done in HOL88 by Wai Wong [Won93]. This could be used in conjunction with proofcheckers that have been written with respect to formalized notions of HOL proof, such as those developed by Rob Arthan [Art90] and Joakim von Wright [vW94].
6. The 'integers' in both SML and CAML Light are highly unsatisfactory. In CAML Light they are underlying machine integers minus one bit (presumably needed for garbage collection). They will wrap silently in both directions without generating an exception. The SML Standard has almost nothing useful to say about the behaviour of integers, rendering it extremely awkward for serious correctness proofs. Poly/ML sensibly provides bignums, whereas SML/NJ relies on machine arithmetic (though it does detect overflow).

   Owing to the use of de Bruijn terms, integers are used in an essential way in the `gtt` core; however although this may be an impediment to completely formal verification, it is hardly conceivable that overflow could occur. More worrying is the use of integers to provide unique variable names via `genvar`. It is not completely inconceivable that on in a large proof, these could wrap, causing derived rules to fail in puzzling ways. This raises the issue of how best to generate unique names, and whether it might be satisfactory (though presumably less efficient) to use variants of free variables in terms under consideration in a given derived rule.

Another issue is the use of ML integers to perform arithmetic in the logic. Without bignums this cannot be done for large numbers, though the change to numeral representation described in the previous section might make this concern irrelevant.

7. One slight irritant when writing functions which recursively traverse terms is that the term constructors like `mk_comb` are not the primitive constructors, but rather abstract functions. This means that it is not possible to pattern match against them, which creates difficulties. Rather than using the usual elegant style of SML function definitions, the programmer is forced to use explicit destructors. Elsa Gunter has invented a way of achieving such matching, though it might rely on the fact that destructors do no real computation and hence might only work with a non-de Bruijn implementation of terms. Again, the size of `gtt` makes this kind of investigation easy.
8. Overloading. In many situations (*e.g.*, algebra or number theories) the overloading of constant names can be tremendously useful. `gtt` is a good place to prototype various proposals for this feature.
9. Higher order matching. The limitations of the matching strategy used by current HOL implementations in rewriting, *modus ponens*, forward chaining, *etc.*, can be restrictive. Sometimes one has to resort to manually instantiating and then beta-reducing a theorem. Even a limited form of second order matching would be much more flexible. In particular, the quantifier movement conversions (and even beta-conversion) could be done by rewriting. This has already been explored in the CAML Light version of `gtt` and results are promising. The more general matching has even been integrated with the term nets used in rewriting, so there is little or no slowdown when looking for higher order matches.

Conclusion

We have developed a simple reference implementation of the HOL logic. The code is easy to port, simple to grasp, and we have developed a small but useful front end. It has already proved to be useful in developing an alternative HOL implementation. We hope that it is of use to others who wish to experiment with HOL. In closing, we would like to point out the obvious: any development of `gtt` into a large and complex system, *i.e.*, a full implementation of HOL, will not remove the need for a simple, executable reference for that large and complex system.

References

[Art90] R.D. Arthan. A formal specification of HOL. Technical Report DS/FMU/IED/SPC001, ICL Defence Systems, April 1990.

[Bou92] Richard Boulton. Lazy techniques for fully expansive theorem proving. In L. Claesen and M. Gordon [LM92].

[CH90] R. Constable and D. Howe. Nuprl as a general logic. In P. Oddifreddi, editor, *Logic and Computer Science*, pages 77–88. Academic Press, 1990.

[dB72] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indag. Math.*, (34):381–392, 1972.

[Gor82] Michael Gordon. Representing a logic in the LCF metalanguage. In D. N'eel, editor, *Tools and Notions for Program Construction*, pages 163–185. Cambridge University Press, 1982.

[Gor93] Michael Gordon. Note to info-hol mailing list, archive number 0914. Tagging theorems for proof acceleration, January 1993.

[Gun91] Elsa Gunter. Abstract Theories for HOL. In *Informal Proceedings of the 1991 Hol User's Group Workshop*. Unpublished, Aarhus, Denmark, 1991.

[Har94] John Harrison. Binary decision diagrams as a hol derived rule. In *HUG94 (LNCS 859)*, Malta, 1994.

[Hue89] Gerard Huet. The constructive engine. In *Calculus of Constructions: Documentation and User's Guide - Version 4.10 (INRIA Techreport 110)*, 1989.

[KAL92] Saraswati Kalvala, Myla Archer, and Karl Levitt. Implementation and use of annotations in HOL. In L. Claesen and M. Gordon [LM92].

[Lau90] John Launchbury. *Projection Factorisation in Partial Evaluation.* PhD thesis, Glasgow University, 1990.

[LM92] L. Claesen and M. Gordon, editors. *International Workshop on Higher Order Logic Theorem Proving and its Applications*, Leuven, Belgium, September 1992. IFIP TC10/WG10.2, Elsevier Science Publishers.

[Mau93] Michel Mauny. Functional programming in CAML-Light. Technical report, INRIA, France, 1993.

[Mel92] Tom Melham. The HOL logic extended with quantification over type variables. In L. Claesen and M. Gordon [LM92].

[Sli92] Konrad Slind. Adding new rules to an LCF-style logic implementation: Preliminary report. In L. Claesen and M. Gordon [LM92].

[vW94] Joakim von Wright. Representing higher order logic proofs in HOL. Technical report, University of Cambridge, 1994. forthcoming.

[Win92] Phil Windley. Abstract theories in HOL. In L. Claesen and M. Gordon [LM92].

[Won93] Wai Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, 1993.

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