# A Thread of HOL Development

MICHAEL NORRISH AND KONRAD SLIND

University of Cambridge Computer Laboratory Email: {mn200,kxs}@cl.cam.ac.uk

The HOL system is a mechanized proof assistant for higher order logic that has been under continuous development since the mid-1980s, by an ever-changing group of developers and external contributors. We give a brief overview of various implementations of the HOL logic before focusing on the evolution of certain important features available in a recent implementation. We also illustrate how the module system of Standard ML provided security and modularity in the construction of the HOL kernel, as well as serving in a separate capacity as a useful representation medium for persistent, hierarchical logical theories.

Received July 20, 2000; revised July 20, 2000; accepted July 20, 2000

#### 1. INTRODUCTION

In the early 1980s, Mike Gordon took up an implementation of Church's formulation of Simple Type Theory (1) as a platform for research in hardware verification. Both the logic and the system were called HOL (Higher Order Logic). Gordon recounts the story of the development of HOL in (2). The implementation and the logic were both influential, spawning much research, and also other implementations.

The original HOL implementation was derived from that of LCF (3), and subsequent implementations of HOL continue to be heavily influenced by ideas from LCF. The *LCF approach* implements a logic in a strongly typed programming metalanguage. The central design tenet of an LCF-style system is that the primitive inference rules of the logic are the only constructors of an abstract type of theorems. The type of theorems is the centerpiece of the *kernel* of the logic. On top of the kernel, people not experts in the logic can safely develop further facilities and theories by programming in the metalanguage.

There have been a number of implementations of the HOL logic in the LCF tradition:

- HOL88 (4) was the first major public release of an implementation of HOL. The system was responsible for the popularity of HOL, and had a large amount of support theories, and documentation. The system was programmed in Classic ML, the original ML language developed in the LCF project. Classic ML was itself implemented in Lisp.
- HOL90 (5) was a re-implementation, by the second author (under the supervision of Graham Birtwistle), of HOL88 in Standard ML (SML). The goal of HOL90 was to provide nearly-identical functionality to HOL88, but with more speed.

**ProofPower** (6) is a complete re-implementation of

HOL, also in SML, by a team from ICL. One major application of the system has been to provide a mechanization of the Z specification language. ProofPower was recently used to verify a 'one-way regulator'. This is a real industrial system and was the first formal verification to attain the highest level of certification from the UK authorities (7).

- Isabelle/HOL (8) is an application of the Isabelle logical framework to the HOL logic. Isabelle/HOL is one of the most highly-engineered HOL systems. There are many novelties in Isabelle/HOL; perhaps the most important is that rules are not programs, as in LCF, but are formulae in the Isabelle metalogic. In Isabelle, inference rules are applied via higher order unification.
- LAMBDA (9) was a commercial implementation of HOL by the company Abstract Hardware (now defunct). The implementation was influenced by the Isabelle approach: rule application was by higher order unification. LAMBDA provided a highly developed user interface for hardware design.
- HOL Light (10) had its genesis in an SML implementation by John Harrison and the second author aimed at discovering how small a HOL kernel could be. Harrison then ported the system to Caml-Light and has since redeveloped it completely. HOL Light is now being used for floating point algorithm verification at Intel (11).
- HOL98 began as an adaptation, by Ken Larsen and the second author, of HOL90 to Moscow ML. The emphasis of Moscow ML on fast separate compilation enabled a significant re-design of the system, parts of which are described in the sequel.

As can be seen, there are currently several distinct strands of HOL implementation in progress. Our purpose in this paper is to recount some of the significant milestones of one thread of HOL development, namely that proceeding from HOL88 to HOL90 to HOL98. We shall first give a survey of how certain important facilities of HOL98 evolved, including recent work aimed at wrapping proof tools up as distributed components in a 'plug-and-play' architecture. Finally, we describe some significant aspects of the implementation of HOL98.

# 2. THEOREM PROVING TOOLS

In this section we describe various important facilities supporting proof in HOL98. We do not claim that these facilities are unique (similar support exists in most contemporary mechanized proof assistants), and cannot point to a pre-existing 'grand design'. In fact, support tools and theories have typically been created in response to shortcomings in the system exposed by attempted verifications. Although this approach is *ad hoc*, the current set of tools in HOL98 allows much better progress to be made in formalizations than in earlier incarnations of HOL.

The facilities break down into definition principles, proof procedures, previously built theories, syntax support and inter-tool linkage.

# 2.1. Definition Principles

Making definitions is a crucial aspect of verification. The HOL logic only provides very simple definition facilities, which are too low-level for many formalizations. In response, an enduring aspect of HOL development has been the construction of ever more powerful *derived* definition principles—which reduce via inference to applications of the primitive definition facilities. In the following, we highlight some of the current functionality.

Inductively defined sets Inductive definitions are common in mathematics and computer science; for example, they are used to build datatypes like the natural numbers and lists. They are heavily used to model proof systems, in formalizing static and dynamic operational semantics; for example, SML (12), CCS (13) and C (14). They may also be used in the construction of recursive functions.

An inductive definition package takes as input a specification of the 'rules' used to add elements to the desired set. It then defines the desired set (or sets, mutually recursive definitions are supported), and returns a package of useful theorems: the rules for building elements in the set, an induction principle, and a 'cases' (or inversion) theorem, useful for breaking down elements of the set based on how they were constructed. The first inductive definition package for HOL was due to Melham (15); a subsequent implementation by Harrison additionally allowed infinitary hypotheses in rules (16). Datatype definitions Many HOL formalizations require the definition of new types. For example, ML-style datatypes are commonly used to model the abstract syntax of programming languages and the state-space of elaborate transition systems. In HOL98, such datatypes (at least, those that are inductive, or, alternatively, have a model in an initial algebra) may be specified using the syntax in Figure 1.

[binding;]\* binding type-spec ::= binding ::= *id* = *constr-spec* id = record - spec[clause |]\* clause constr-spec ::= idclause : : =  $id \text{ of } [type =>]^* type$ 1  $< | [id : type ;]^* id : type |>$ record-spec ::=



Datatype specifications allow the use of record types, which may be recursive. For example, the following datatype could be used to formalize a simple file system.

```
file = Text of string | Dir of directory ;
directory =
    <| owner : string ;
        files : (string * file) list |>
```

When a datatype is defined, a number of standard theorems are automatically proved about the new type: the constructors of the type are proved to be injective and disjoint, induction and case analysis theorems are proved, and each type also has a 'size' function defined for it. Size functions for types map values of the types into  $\mathbb{N}$  and are used by the function definition package in termination proofs. The standard theorems about datatypes are all collected in an internal database and used by several other packages, *e.g.*, the simplifier.

The datatype package of HOL98 took some time to evolve to its present state. The package in HOL88 was ported to HOL90 and was central in many verifications. However, it did not provide advanced features such as mutually recursive types and recursion under type operators. Since these features are often required, especially when modelling computer languages, extension packages, e.g., (17), were written in HOL90. These packages were crucial for many challenging verifications; however, they were somewhat difficult to use, and as a result, in HOL98 we have adopted and extended a datatype package from HOL Light.

One observation about datatype definition packages implemented via inference is that they are difficult to write, and also seem to be somewhat inefficient when dealing with large datatypes featuring much nesting under type constructors. Whether significantly more efficient inference-based packages are possible should be investigated.

Recursive function definition Recursive functions are pervasive in formalization. In HOL88 and HOL90, the only recursion style supported was primitive recursion over the constructors of datatypes. In order to make it easy to specify more complex recursive functions, a package based on well-founded recursion was developed (18, 19). The package handles many recursion styles (nested, mutual, higher-order, and schematic (19) and accepts equations in the pattern matching style popular in functional programming languages. In contrast to primitive recursion, the format of which guarantees termination (and thus totality), a function defined by well-founded recursion must be proved to terminate; HOL98 attempts to prove termination automatically, but the current termination prover, although useful, is quite basic.

A function defined by well-founded recursion also specifies a useful induction theorem in which the inductive hypothesis holds for the arguments to recursive calls. For each recursive definition, HOL98 automatically derives this 'customized' induction theorem from a general well-founded induction theorem (20).

#### 2.2. Proof procedures

The primitive inference rules of HOL are too low-level on their own to support interesting verifications. Thus, an enduring aspect of HOL development is proof automation, achieved by using ML to compose proof procedures. A constraining factor in this process has been the requirement that the procedure use deductive steps in HOL, but many (most?) of the decision procedures found in the literature are not described deductively. In spite of this obstacle, a number of such experiments, discussed below, have shown the feasibility of this enterprise.

*Decision procedures* When a proof goal arises that lies in a decidable set of formulas, it is desirable that the goal be proved without further human effort. A number of decision procedures have been implemented in HOL over the years.

For the fragment of HOL corresponding to propositional logic, a naïve algorithm has been provided in the system for some time; however, it is only suitable for small problems. An ROBDD-based algorithm is discussed in Section 2.5.

For the fragment of HOL corresponding to first order logic, a model elimination procedure (21) has been extremely useful in raising the level of automation in many verifications. An interesting aspect of this procedure is that it conducts proof search in ML using a non-HOL-specific representation; in case the search is successful, the representation provides enough information to generate a HOL proof.

HOL98 provides three decision procedures for various flavours of arithmetic. The earliest, due to Boulton (22, §5.2), solves most universally and some existentially quantified formulas in linear arithmetic over  $\mathbb{N}$ . The treatment of universal formulas is a adaptation of Hodes' method (23) to  $\mathbb{N}$  and is incomplete because it does not take divisibility into account; for example, it fails to prove  $\forall x. 2x \neq 5$ . The procedure for existential formulas is Shostak's SUP-INF procedure (24), which is also incomplete.

HOL98 includes a complete procedure for all universally quantified linear formulas over  $\mathbb{R}$ . This was implemented by Harrison as part of his doctoral research (25, §5). In that work, Harrison also implemented a version of Tarski's decision procedure for real fields but, due to its computational complexity, this procedure is not often useful and is not publicly available.

Recently, an implementation of Cooper's algorithm (26) for deciding full Presburger arithmetic over  $\mathbb{Z}$  and  $\mathbb{N}$  has been implemented by the first author. Results suggesting that Cooper's method can outperform Hodes' method on universal formulas have been reported (27). While our implementation of Cooper's method is not yet mature enough to compete with Boulton's code on those goals they can both solve, it proves many goals quickly enough to be promising for interactive use. For example, it proves the group axiom for  $\mathbb{Z}$ 

$$\exists !e. \ (\forall x. \exists !y. x + y = e) \land (\forall x. x + e = x)$$

in 0.6s on a 600 MHz Pentium III.

Nelson and Oppen's method for combining decision procedures, which includes an implementation of congruence closure, is also available in the system (28).

Simplification Simplification is a pervasive activity in proof. The HOL98 simplifier was implemented by Don Syme, inspired by the powerful simplifier in Isabelle. The core algorithm of the simplifier is based on a Paulson-style rewriter (29), which traverses the term M to be simplified, and repeatedly attempts to apply rewrite rules at all subterms of M. This basic engine is augmented with the following features:

• Conditional rewrite rules of the form

$$c_1 \wedge \dots \wedge c_n \supset (x = y)$$

may be applied. The simplifier recursively invokes itself on the side conditions generated. This invocation may in turn try to apply further conditional rewrites. The stack of side conditions is not allowed to exceed a user-specified depth.

- So-called *congruence rules* may be applied; these enable the rewriter to accumulate and discard context as a term is traversed. For example, while operating on a term if b then  $e_1 \\ else \\ e_2$ , simplification of  $e_1$  can assume b, and simplification of  $e_2$ can proceed while assuming  $\neg b$ . Similarly, P can be assumed while simplifying Q in  $P \supset Q$ .
- Rewrite rules are applied using a simple form of higher order matching, closely related to the algorithm discovered by Miller (30).

- Standard rewrite rules arising from datatype definitions are automatically incorporated into the simplification process.
- The simplifier can be extended with arbitrary proof procedures. These are invoked by the simplifier on appropriate sub-terms and are passed the current accumulated context. The use of Boulton's linear arithmetic procedure during simplification has dramatically shortened many verifications. This is by no means the last word on the integration of decision procedures into simplification: a higher level of interaction between arithmetic decision procedures and a simplifier has been reported in (31).

Support for simplification in HOL88 and HOL90 was much more basic: only unconditional rewrite rules could be applied, and on-the-fly invocation of decision procedures was thought to be prohibitively slow. The amazing increase in computational power in standard workstations and PCs over the last decade has meant that operations previously thought to be slow—and thus only suitable for interactive application during proof are now fast enough to be used deep inside automated tools.

Computation Many tasks arising during verification amount to no more than evaluation of logical functions, e.g., reducing ground terms built up from boolean and arithmetic operators. Historically, HOL systems implemented ad hoc simplifiers for such tasks. In recent work (32), Barras implemented an abstract machine for callby-value reduction in HOL. A novel aspect of this work is that the actual reduction steps are implemented by inference rules. The result is a derived inference rule that evaluates ground HOL terms, just as if by an ML interpreter. Although Barras' inference rule runs much slower than a conventional ML implementation, the slowdown is a constant factor; moreover, the procedure is sufficiently quick to provide a general-purpose solution for many reduction tasks arising during verification. In contrast to Barras' LCF-style inference rule, the ACL2 and PVS systems use computation in their metalanguage (Lisp) to perform such tasks as calculation in the logic. The ACL2 system also implements symbolic evaluation for logically defined functions, and this has been found to be very useful in verifications (33). It remains to provide a general purpose and relatively efficient symbolic simulation rule in HOL98.

# 2.3. Verification theories

A range of theories embodying basic mathematical structures have been built in the HOL system: pairs, disjoint unions, the 'option' type, numbers ( $\mathbb{N}$ ,  $\mathbb{Z}$  and  $\mathbb{R}$ ), lists, 'lazy' lists, relations, finite and infinite sets and multisets, finite maps, strings, and words. Other mathematical developments in the system include linear time temporal logic,  $\omega$ -automata, analysis over metric spaces, polynomials, and probability theory. In general,

these theories have not been provided by the HOL developers; instead, they are often *user* contributions. As a result, theory development has not been systematic: a theory is often built when needed to support a particular verification exercise. If the theory seems to have wide utility, it is then incorporated into the standard distribution.

Numerous computer science oriented formalizations have also been constructed in HOL, including theories for Hoare Logic, UNITY, CCS,  $\pi$ -calculus, SML, C, computability theory, *etc.* In contrast to the 'standard' mathematical theories above, these theories, while vital to their developers' needs, seem not to have been widely adopted. This seems to suggest that either a truly useful verification formalism has yet to be invented, or that standard mathematics suffices.

# 2.4. Syntax support

Verifications of any size require interaction with the proof assistant. A common activity is thus the parsing and printing of logical objects *e.g.*, formulas. The parsing and pretty-printing support offered in HOL98 is significantly more flexible than that of HOL90, which provided only a fixed grammar, and the ability to declare new infixes. Another change is that, in contrast to HOL88 and HOL90, the HOL98 parser and pretty-printer are now completely separate from the logical kernel. This has two benefits: core functionality is not compromised by the intrusion of unrelated code, and the parser can implement abstractions of its own (such as overloading) that have no reflection in the underlying logic. Below we discuss three syntax support features recently added to HOL98.

*Explicit grammars* The parser is explicitly parameterized by a grammar, which records the productions used by the parsing algorithm, as well as precedence levels for operators. For example, '+' is recorded as being a left associative infix at level 500. By making parsing explicitly dependent on grammar values, the system becomes more robust and flexible because library code can parse with respect to a specific grammar. This provides an important kind of static scoping for HOL expressions that are parsed as a library loads.

Mix-fix forms The parser uses operator precedence parsing (34, §4.6). This algorithm is simple to implement, deterministic, and runs efficiently. To add to the single-token prefixes, suffixes and infixes provided by this technique, we have made a simple extension to the notion of operator: an operator is not necessarily a single token (such as +), but can also be a sequence of tokens and non-terminals. There are four fixities: infix, prefix, suffix, and 'close-fix'; these are the four possibilities arising from two independent choices: whether or not an argument to the left of the operator is possible, and whether or not an argument to the right is possible. For example, if-then-else is viewed as prefix operator where the first two arguments are enclosed within the tokens (between the if-then and between the thenelse). Only the last argument is subject to binding competition with other operators. This simple extension makes sense of the close-fix fixity, which might be used to implement syntax representing "semantic brackets" as  $[| _ - |]$ .

Overloading Specifying that a constant is to be overloaded in HOL98 is straightforward: one simply specifies which constants should be instances of which overloaded names. For example, one can require that addition over  $\mathbb{N}$  and  $\mathbb{Z}$  (separate functions in HOL) should both be instances of the + identifier. Type inference is used to help resolve overloading. Following the example of Haskell, HOL98 also overloads numerals (sequences of digits) inhabiting different types. This is achieved by treating every occurrence of a numeral n as if it were the term &n, where & is an overloaded symbol denoting a contextdetermined injection function into the appropriate numeric domain. With this scheme, type inference is used as a basis for selecting the injection function coercing a numeral to an integer, natural number or a real.

Our parser, while quite flexible, is not as general, nor as sophisticated as that of Isabelle, which uses Earley's non-deterministic context-free parsing algorithm (35). Also, the approach to overloading in Isabelle is based on type classes (36), which in turn reflects the specification of types in the underlying object logic. HOL98 does not support type classes, so it is reasonable not to force the user to conform to any particular organisation of their overloaded constants.

# 2.5. Inter-tool linkages

An important facet of HOL98 is its acceptance of the rôle that external tools have to play in the use of interactive theorem-proving systems. Historically, the attitude in HOL implementations has been rather cautious: external tools might prove results quickly, and dependably, but they might also be incorrect.

Such a purist approach is not always pragmatically justified. Systems such as Lifted-FL (37), used by Intel to verify aspects of chip design, rely on a seamless integration between the logical core and external tools, which are typically highly engineered for their problem domain. The PVS system (38) has also integrated a number of external tools, including a modelchecker (39).

The PROSPER project (40) is one response to the demand for integration of external tools with HOL. It has developed a common framework for the linking together of verification tools. Communication of terms, types and theorems is done using the basic HOL representations, providing an abstract syntax with which to communicate logical objects over computer networks. Implementations of the interface are available for the C, Java, SML and Python programming languages.

An early inter-tool linkage using PROSPER was performed by Hurd (41), who linked HOL98 and the resolution prover Gandalf (42). Gandalf provides a reasonably detailed log of successful proofs, and Hurd's connection automatically translates these logs to HOL proofs that, when executed, prove the goal originally sent to Gandalf. Thus, worries about soundness are circumvented: Gandalf searches for and provides a proof, not just an assertion that the goal is true.

This kind of *proof-reconstruction* approach is not possible with a tool that produces nothing that could be construed as a proof. If one wants to use such a tool, the results it produces must simply be accepted as HOL theorems. However, the soundness of HOL is thereby imperiled. The solution adopted by HOL98 is to tag all such foreign theorems. A tag signifies that the theorem in question was generated externally, and identifies the source. The primitive inference rules of HOL have been adapted to accumulate and propagate any tags that appear in theorems, much in the same way that hypotheses are accumulated and propagated in inference. There is thus a logical 'audit trail' providing the user with information about which tools have contributed to a result. PROSPER technology has been used by Schneider and Hoffmann to link HOL98 to SMV (43), and it is also used to provide a component-style interface to Prover Technology's commercial implementation of Stålmarck's algorithm (44).

In a separate development, recent work by Gordon (45), allows ROBDDs (an efficient representation of propositional formulas) to be used inside HOL98 in a principled, yet efficient way. Purported theorems coming from application of ROBDD algorithms are tagged before being admitted as HOL theorems.

# 3. ASPECTS OF IMPLEMENTING HOL

We now describe some aspects of the HOL98 implementation, focusing mainly on design issues. Much of the art and science of implementing abstract specifications in a programming language involves mapping specification-level concepts onto constructs provided in the programming language. Below we discuss how the constructs of Standard ML supported the implementation of important properties required by HOL.

## 3.1. The HOL logic

The HOL logic is built on the syntax of a lambda calculus having a polymorphic type system somewhat similar to that of ML.<sup>2</sup> The logic is classical and has a set theoretic semantics, in which types denote non-empty sets and the function space denotes total functions. The logic comprises four components: types, terms, theorems, and theories.

 $<sup>^2\,\</sup>rm Type$  variables may occur in HOL types, just as in ML; however, full let-style polymorphism in the HOL logic would lead to inconsistency and is not allowed.

# 3.1.1. Types

Types and terms are built with respect to signatures. A type signature ( $\Omega$ ) assigns arities to type operators. A HOL type is either a type variable, or a compound type built by applying a type operator in  $\Omega$  of arity k to a list (of length k) of types. Initially,  $\Omega$  contains type operators denoting truth values (bool), function space (written  $\tau_1 \rightarrow \tau_2$ ), and an infinite set of individuals (ind). A type constant, such as bool, is a compound type built from a member of  $\Omega$  with arity 0. Although the abstract syntax of types is easily captured with an ML datatype declaration, the arity check in the construction of compound types means that types may not be freely constructed.

## 3.1.2. Terms

HOL terms are typed  $\lambda$ -calculus expressions built with respect to a signature  $(\Sigma_{\Omega})$ , which assigns types built from  $\Omega$  to term constants. A term can be constructed in only four ways: it is either a variable, an instance of a constant in  $\Sigma_{\Omega}$ , an application  $(M \ N)$  of a term M of type  $\tau_1 \rightarrow \tau_2$  to a term N of type  $\tau_1$ , or a lambda abstraction. (In our opinion, one of the strengths of HOL is the small number of ways a term may be constructed.) Initially,  $\Sigma_{\Omega}$  contains constants denoting equality (=), implication  $(\supset)$ , and Hilbert's indefinite description operator  $(\varepsilon)$ .

The abstract syntax of HOL terms is also easily captured with an ML datatype declaration; however, as for types, terms may not be freely constructed, since a term must be well-typed with respect to  $\Sigma_{\Omega}$ .

The interface routines for manipulating HOL terms present a so-called *name-carrying* syntax. In our thread of HOL development, there have been four implementations of this interface: the HOL88 implementation used a name-carrying representation from LCF; early implementations of HOL90 also used a name-carrying internal representation of terms; later versions represented terms with deBruijn indices; and the current implementation, due to Bruno Barras, uses explicit substitutions. Since terms are an abstract type, these internal changes have not been externally visible.

Types and terms form the basis of the *prelogic*, in which basic algorithmic manipulations on types and terms are defined: *e.g.*, the free variables of a type or term, substitution, matching, and  $\alpha$ - and  $\beta$ -conversion. These algorithms are used to implement higher-level syntax manipulations, and are also used to implement the deductive system.

#### 3.1.3. Theorems, Axioms, and Definitions

Different but equivalent presentations of the deductive system of HOL can be found in (4) or Appendix A of (25). The important thing is that a fixed set of primitive rules and axioms is used as a basis upon which more complex *derived* rules may be built by programming in the metalanguage. The primitive rules of HOL can be used to build a conventional set of natural deduction style rules, with introduction and elimination rules for all the standard logical connectives. The usual rules for equality reasoning are also derivable from the primitives. A final rule is provided for instantiating type variables in a theorem.

Theorems are easy to represent by an ML datatype with fields for the hypotheses and conclusion of the theorem. The type of theorems is abstract in order to meet the requirement that a new theorem may only be produced by application of a primitive rule of inference.

The HOL logic distinguishes between axioms and definitions. An *axiom* is an arbitrary well-typed formula that is simply asserted to be a theorem. It is very easy to assert axioms, but experience has shown that it is all too easy to introduce inconsistency this way. An alternative, popular among users of HOL, is to make definitions and derive the desired consequences by proof. Principles for defining types and terms form part of the HOL logic, and the previous section has described advanced definition facilities, which couple the convenience of asserting axioms with the soundness of using primitive inference.

#### 3.1.4. Theories

The HOL logic provides a very simple notion of *theory*: loosely speaking, a HOL theory is a collection of theorems that have been derived from a set of axioms in a signature. Since signatures are extensible, as is the set of axioms, and also the set of derived theorems, some mechanism is needed to handle different extensions of the initial theory occurring when different theories are formalized. It is straightforward to support such extensions in a single session; however, it is more difficult to support *persistent* theories, in which a theory may be developed and then stored on disk, to be reloaded in a subsequent session. This is discussed in more detail in Section 3.2.2.

#### 3.2. Implementing HOL

We discuss two central aspects of implementing HOL in this section: first, how to securely implement the HOL kernel; second, how to represent theories so that they are persistent. We do not imagine that our approaches are distinguished in any way over others; however, our discussion reveals—by example—useful ML programming idioms that we think are not well enough known. The first idiom uses SML modules to build a *multistructure* abstract datatype. The second idiom shows how ML structures, normally thought of as containers for programs, also serve very well as containers for hierarchical *data*.

#### 3.2.1. Multi-structure ADTs

The principal design challenge in implementing HOL comes from a tension between encapsulation and modularity. We have seen that HOL types, terms, and theorems must be implemented by abstract datatypes; theories keep track of the state of a logical development and thus they must also be protected from arbitrary user meddling. Thus, the implementations of types, terms, theorems, and theories need to be encapsulated in a *kernel*. Inside the kernel, direct access to representations of abstract types is permitted, and the state of the system can be viewed and altered by any piece of code in the kernel. From outside however, access to the representations and states of the kernel is strictly controlled by kernel code that maintains important system invariants, such as well-typedness.

On the other hand, it seemed quite natural in the implementation to divide the kernel into four separate modules Type, Term, Thm, and Theory implementing types, terms, theorems, and theories. The conflict arises when a function in a kernel module either needs access to the representation of an abstract type declared in another kernel module, or when it directly manipulates state held in another kernel module. In our design, this gives rise to mutual dependencies among the kernel modules. Since user level programming also accesses the very same Type, Term, Thm, and Theory modules, it would be catastrophic if critical kernel representations and datastructures were user-accessible.

ML provides several means of solving this problem. To some, the simplest solution would use the abstract datatype facility of Core ML. Although this approach is possible, it did not appeal to us: it would require four mutually recursive abstract types (they are mutually dependent, but not recursively so), held in one (large) file, which we thought was insufficiently modular. Instead, we were more interested in applying the abstraction facilities available at the module level.

The solution we settled on builds the kernel from its component structures. First a version of Type, called RawType, is built which exposes its representation and internal state. From RawType, a raw version of Term, called RawTerm, is built which also exposes its representation and internal state. RawType and RawTerm are then curtailed to the desired user-level structures Type and Term by signature restriction. Following this, the structures Thm and Theory can be built by applying the functors THM and THEORY. Note that Thm accesses the representation of terms. Also, Theory uses signatureupdate functions provided by RawType and RawTerm; signature restriction removes these dangerous functions to obtain the safe structures.

```
local structure RawType = TYPE()
    structure RawTerm = TERM(RawType)
in
    structure Type:Type = RawType
    structure Term:Term = RawTerm
    structure Thm = THM(RawTerm)
    structure Theory =
    THEORY(structure Thm = Thm
        structure Term = RawTerm)
end
```

The whole development is wrapped in a module-

level local ... in ... end block so that RawType and RawTerm are ephemeral, disappearing after the multistructure kernel is created. Users can only access kernel functions through Type, Term, Thm, and Theory, and cannot therefore directly access the internal representations or modify the state of the kernel.

This design is not perfect; for example, the functors TYPE, TERM, THM, and THEORY continue to exist after the kernel is built, cluttering up the functor namespace. They could be redefined to be vacuous, but that is hardly elegant.

### 3.2.2. Persistent theories

The concept of *theory segment* is used to formalize persistent theories. Conceptually, a segment is a container in which related logical entities are stored. Theory segments are hierarchically arranged by a *dependency* relation that tells when one segment depends on concepts or results formalized in another *parent* segment. The theory  $\mathcal{T}$  corresponding to a segment is built by taking the component-wise union of all the segments found in the transitive closure of the parent relation. There is a root segment corresponding to the initial signatures and axioms of HOL.

A typical piece of work with HOL98 consists in a number of sessions. In the first of these, a new theory,  $\mathcal{T}$ say, is created by importing some existing theory segments, making a number of definitions, and perhaps proving and storing some theorems in the current segment. Eventually the current segment is exported to disk. The concrete result will be a file containing the theory segment created during the session and whose ancestry represents the desired logical theory  $\mathcal{T}$ . Subsequent sessions can access the definitions and theorems of  $\mathcal{T}$  by importing the file; this avoids having to load the tools and replay the proofs that created the theory in the first place.

In HOL90, theory segments were stored on disk in an *ad hoc* format which required much code implementing input and output of segments. A more serious problem was that fetching elements from a segment was dynamic: for example, accessing the arithmetic theorem ADD\_CLAUSES was done via function call:

## theorem "arithmetic" "ADD\_CLAUSES"

Thus, mapping segment-level bindings to ML-level bindings could only happen dynamically. As a consequence, it was generally impossible to determine the dependencies in a collection of ML code with theories.

In HOL98, theory segments are directly represented by ML structures: the bindings of a theory segment can be represented by ML variable bindings, and the parenthood relation can be mapped into the dependency of structures. With this representation, the above theorem is stored in a structure named arithmeticTheory, under the binding ADD\_CLAUSES, which can be accessed using the standard 'dot' notation, *i.e.*, arithmeticTheory.ADD\_CLAUSES. Since theorylevel binding is achieved by the binding of variables in a structure, dependency analysis of HOL formalizations can be achieved by extending ML dependency analysis. This has been implemented by the first author, and the resulting tool, called Holmake, is now extensively used for dependency maintenance in large HOL98 formalizations, including the whole distribution.

# 4. CONCLUSIONS

The HOL system has been under development in one form or another for almost twenty years. In that time, though the number of implementations has burgeoned, the specification of the logic has been stable. This has given many people the chance to implement and develop various tools for the same logic. For example, the current implementation of HOL98 benefits greatly from the tools available in Harrison's HOL Light implementation. Significant effort has been expended to port tools from HOL Light to HOL98, but this has always been reasonably straightforward, since the two systems implement the same specification.

The amazing improvements in speed and memory capacity of computers over the past two decades has meant that different design choices became possible with the passage of time. We have given some indication of how this story has played out in the development of a sequence of HOL implementations. In general, the transition has been towards ever-higher levels of automation, and away from considerations based purely on speed and memory consumption. System interfaces have also been gradually generalized in many ways. We think that this trend will continue, for the wider uptake of formal verification will require simpler interfaces to more powerful tools.

## ACKNOWLEDGEMENTS

We thank the anonymous referees for their detailed and helpful comments.

The second author sincerely thanks Graham Birtwistle for inspiration and encouragement during the past 15 years. Graham adopted ML and HOL early and with characteristic vigour; as usual, he was streets ahead!

#### REFERENCES

- Alonzo Church. A formulation of the Simple Theory of Types. Journal of Symbolic Logic, 5:56-68, 1940.
- [2] M. J. C. Gordon. From LCF to HOL: a short history. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, language and interaction: essays in honour of Robin Milner*, Foundations of Computing, pages 169–185. MIT Press, 2000.
- [3] M. J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*. Number 78 in Lecture Notes in Computer Science. Springer, 1979.

- [4] M. J. C. Gordon and T. Melham (editors). Introduction to HOL: a theorem proving environment. Cambridge University Press, 1993.
- [5] Konrad Slind. An implementation of higher order logic. Master's thesis, University of Calgary Computer Science Department, 1991. Available as technical report 91-419-03.
- [6] R. D. Arthan. A report on ICL HOL. In Archer et al. (48), pages 280–283.
- [7] International Computers Limited (ICL).
   One way regulator, 1990. See http://www.itsec.gov.uk/cgi-bin/cplview.pl?docno=44.
- [8] Lawrence C. Paulson. Isabelle: A Generic Theorem Prover. Springer-Verlag LNCS 828, 1994.
- [9] Mick Francis, Simon Finn, Ellie Mayger, and Roger Hughes. Reference manual for the LAMBDA system. Technical Report 4.2.1, Abstract Hardware Limited, 1992.
- [10] 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.
- [11] John Harrison. A machine-checked theory of floating point arithmetic. In Bertot et al. (46), pages 113–130.
- [12] Myra VanInwegen. The machine-assisted proof of programming language properties. PhD thesis, University of Pennsylvania, December 1996.
- [13] Monica Nesi. A formalization of the process algebra CCS in higher order logic. Technical Report 278, Computer Laboratory, University of Cambridge, December 1992.
- [14] Michael Norrish. C formalised in HOL. PhD thesis, Computer Laboratory, University of Cambridge, 1998.
- [15] T. Melham. A package for inductive relation definitions in HOL. In Archer et al. (48), pages 350–357.
- [16] John Harrison. Inductive definitions: automation and application. In Schubert et al. (47), pages 200–213.
- [17] E. L. Gunter. A broader class of trees for recursive type definitions for HOL. In J. J. Joyce and C.-J. H. Seger, editors, *Higher Order Logic Theorem Proving and its Applications: 6th International Workshop (HUG'93)*, number 780 in Lecture Notes in Computer Science, pages 141–154. Springer-Verlag, Vancouver, B.C., August 11-13 1994.
- [18] Konrad Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics*, number 1125 in Lecture Notes in Computer Science, Turku, Finland, August 1996. Springer-Verlag.
- [19] Konrad Slind. Reasoning about Terminating Functional Programs. PhD thesis, Institut für Informatik, Technische Universität München, 1999. Accessible at http://www.cl.cam.ac.uk/users/kxs/papers.
- [20] Konrad Slind. Derivation and use of induction schemes in higher order logic. In *Theorem Proving in Higher Order Logics*, number 1275 in Lecture Notes in Computer Science, Murrary Hill, New Jersey, USA, August 1997. Springer-Verlag.
- [21] John Harrison. Optimizing proof search in model elimination. In M. A. McRobbie and J. K. Slaney, editors, 13th International Conference on Automated Deduction, volume 1104 of Lecture Notes in Computer

Science, pages 313–327, New Brunswick, NJ, 1996. Springer-Verlag.

- [22] R. J. Boulton. Efficiency in a fully-expansive theorem prover. PhD thesis, Computer Laboratory, University of Cambridge, May 1994.
- [23] Louis Hodes. Solving problems by formula manipulation in logic and linear inequalities. In D. C. Cooper, editor, Proceedings of the 2nd International Joint Conference on Artificial Intelligence, pages 553-559, London, UK, September 1971. William Kaufmann.
- [24] R. E. Shostak. On the SUP-INF method for proving Presburger formulas. Journal of the A.C.M., 24(4):529-543, October 1977.
- [25] John Harrison. Theorem Proving with the Real Numbers. CPHC/BCS Distinguished Dissertations. Springer, 1998.
- [26] D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, volume 7, pages 91–99, New York, 1972. American Elsevier.
- [27] P. Janicic, I. Green, and A. Bundy. A comparison of decision procedures in Presburger arithmetic, October 1997. Research paper #872, Division of Informatics, University of Edinburgh.
- [28] R. J. Boulton. Combining decision procedures in the HOL system. In Schubert et al. (47), pages 75–89.
- [29] Lawrence Paulson. A higher order implementation of rewriting. Science of Computer Programming, 3:119-149, 1983.
- [30] D. Miller. Unification of simply typed lambda-terms as logic programming. In K. Furukawa, editor, Logic Programming, Proceedings of the Eighth International Conference, pages 255-269. MIT Press, 1991.
- [31] R. S. Boyer and J S. Moore. Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic. *Machine Intelligence*, 11:83-124, 1988.
- [32] Bruno Barras. Programming and computing in HOL. In J. Harrison and M. Aagaard, editors, *Theorem Prov*ing in Higher Order Logics: 13th International Conference, TPHOLs 2000, volume 1869 of Lecture Notes in Computer Science, pages 17–37. Springer-Verlag, 2000.
- [33] J Moore. Symbolic simulation: An ACL2 approach. In G. Gopalakrishnan and P. Windley, editors, Proceedings of the Second International Conference on Formal Methods in Computer-Aided Design (FMCAD'98), volume LNCS 1522, pages 334-350. Springer-Verlag, November 1998.
- [34] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: principles, techniques and tools. Addison-Wesley, 1986.
- [35] J. Earley. An efficient context-free parsing algorithm. Communications of the ACM, 13(2):94-102, February 1970.
- [36] Tobias Nipkow. Order-sorted polymorphism in Isabelle. In Gérard Huet and Gordon Plotkin, editors, *Logical Environments*, pages 164–188. Cambridge University Press, 1993.
- [37] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-FL: a pragmatic implementation of combined model-checking and theorem proving. In Bertot et al. (46), pages 323-340.

- [38] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In Deepak Kapur, editor, 11th International Conference on Automated Deduction, volume 607 of Lecture Notes in Artificial Intelligence, pages 748-752. Springer-Verlag, 1992.
- [39] S. Rajan, N. Shankar, and M.K. Srivas. An integration of model-checking with automated proof checking. In *Conference on Computer-Aided Verification, CAV '95* (LNCS 939), pages 84–97, Liege, Belgium, July 1995.
- [40] Louise A. Dennis, Graham Collins, Michael Norrish, Richard Boulton, Konrad Slind, Graham Robinson, M. J. C. Gordon, and T. Melham. The PROSPER toolkit. In S. Graf and M. Schwartzbach, editors, Proceedings of the 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2000), volume 1785 of Lecture Notes in Computer Science, pages 78–92, Berlin, Germany, March/April 2000. Springer.
- [41] Joe Hurd. Integrating Gandalf and HOL. In Bertot et al. (46), pages 311–321.
- [42] Tanel Tammet. Gandalf. Journal of Automated Reasoning, 18(2):199-204, April 1997.
- [43] Klaus Schneider and Dirk W. Hoffmann. A HOL conversion for translating linear time temporal logic to ωautomata. In Bertot et al. (46), pages 255–272.
- [44] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. Formal Methods in System Design: An International Journal, 16(1):23-58, January 2000.
- [45] M. J. C. Gordon. Linking higher order logic to binary decision diagrams. In Jim Davies, Jim Woodcock, and Bill Roscoe, editors, *Proceedings of the Symposium in Celebration of the work of C. A. R. Hoare*, Cornerstones in Computing. MacMillan, to appear.
- [46] Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors. Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs '99, volume 1690 of Lecture Notes in Computer Science. Springer, September 1999.
- [47] E. T. Schubert, P. J. Windley, and J. Alves-Foss, editors. Proceedings of the 8th International Workshop on Higher Order Logic Theorem Proving and Its Applications, volume 971 of Lecture Notes in Computer Science. Springer-Verlag, September 1995.
- [48] Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors. Proceedings of the 1991 international workshop on the HOL theorem proving system and its applications. IEEE Computer Society Press, August 1991.