From LCF to HOL: a short history

Mike Gordon[1]

1 Introduction

The original LCF system was a proof-checking program developed at Stanford University by Robin Milner in 1972. Descendents of LCF now form a thriving paradigm in computer assisted reasoning. Many of the developments of the last 25 years have been due to Robin Milner, whose influence on the field of automated reasoning has been diverse and profound. One of the descendents of LCF is HOL, a proof assistant for higher order logic originally developed for reasoning about hardware.[2] The multi-faceted contribution of Robin Milner to the development of HOL is remarkable. Not only did he invent the LCF-approach to theorem proving, but he also designed the ML programming language underlying it and the innovative polymorphic type system used both by ML and the LCF and HOL logics. Code Milner wrote is still in use today, and the design of the hardware verification system LCF_LSM (a now obsolete stepping stone from LCF to HOL) was inspired by Milner's Calculus of Communicating Systems (CCS).

2 Stanford LCF

"LCF" abbreviates "Logic for Computable Functions", Milner's name for a logic devised by Dana Scott in 1969, but not published until 1993 [46]. The LCF logic has terms from the typed λ-calculus and formulae from predicate calculus. Types are interpreted as Scott domains (CPOs) and the logic is intended for reasoning, using fixed-point induction, about recursively defined functions of the sort used in denotational semantic definitions.

The original LCF team at Stanford consisted of Robin Milner, assisted by Whitfield Diffie (from whom Milner learnt Lisp). Diffie subsequently became interested in cryptography, where he became well-known. A bit later Richard Weyhrauch joined the team, soon followed by Malcolm Newey. All these people collaborated on designing, implementing and using the original LCF system, now known as Stanford LCF. The resulting system is a proof checker for Scott's logic and is described by Milner as follows [34]:

> The proof-checking program is designed to allow the user interactively to generate formal proofs about computable functions and functionals over a variety of domains, including those of interest to the computer scientist - for example, integers, lists and computer programs and their semantics. The user's task is alleviated by two features: a subgoaling facility and a powerful simplification mechanism.

[1] University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge CB2 3QG, U.K.

[2] "HOL" abbreviates "Higher Order Logic".

Proofs are conducted by declaring a main goal (a formula in Scott's logic) and then splitting it into subgoals using a fixed set of subgoaling commands (such as induction to generate the basis and step). Subgoals are either solved using a simplifier or split into simpler subgoals until they can be solved directly. Data structures representing formal proofs in Scott's logic are created when proof commands are interpreted. These can consume a lot of memory.

Stanford LCF was used for a number of case studies. Weyhrauch worked on the proof of correctness of a compiling algorithm for a simple imperative language to a stack-based target language [36] and Newey on the generation of equational theories of integers and lists.

3 Edinburgh LCF

Around 1973 Milner moved to Edinburgh University and established a project to build a successor to Stanford LCF, which was subsequently dubbed Edinburgh LCF. He initially hired Lockwood Morris and Malcolm Newey (both recent PhD graduates from Stanford) as research assistants. Two problems with Stanford LCF were (i) that the size of proofs was limited by available memory and (ii) the fixed set of proof commands could not be easily extended. Milner set out to correct these deficiencies in Edinburgh LCF. For (i) he had the idea that instead of saving whole proofs, the system should just remember the results of proofs, namely theorems. The steps of a proof would be performed but not recorded, like a mathematics lecturer using a small blackboard who rubs out earlier parts of proofs to make space for later ones. To ensure that theorems could only be created by proof, Milner had the brilliant idea of using an abstract data type whose predefined values were instances of axioms and whose operations were inference rules. Strict typechecking then ensured that the only values that could be created were those that could be obtained from axioms by applying a sequence of inference rules – namely theorems. To enable the set of proof commands to be extended and customised – (ii) above – Milner, ably assisted by Morris and Newey, designed the programming language ML (an abbreviation for "Meta Language"). This was strictly typed to support the abstract type mechanism needed to ensure theorem security [35].

In Stanford LCF, the axioms and rules of inference of Scott's logic were directly encoded in the implementation of the simplification and subgoaling mechanism. The user could only construct proofs 'backwards' from goals. In Scott's unpublished paper, his logic was presented in the conventional way by giving axiom schemes and rules of inference. The direct notion of formal proof suggested by this is a sequence, each member of which is either an axiom or follows from an earlier member via a rule of inference (this notion of proof is sometimes called 'forward'). By encoding the logic as an abstract type, Edinburgh LCF directly supported forward proof. The design goal was to implement goal directed proof tools by programs in ML. To make ML convenient for this, the language was made functional so that subgoaling strategies could be represented as functions (called "tactics" by Milner) and operations for combining strategies could be

programmed as higher-order functions taking strategies as arguments and returning them as results (called "tacticals"). It was anticipated that strategies might fail (e.g. by being applied to inappropriate goals) so an exception handling mechanism was included in ML. The needs of theorem proving very strongly influenced the design of the first version of ML. Many design details were resolved by considering alternatives in the light of their use for programming proof tools. This narrow design focus resulted in a simple and coherent language.

In 1975, Morris and Newey took up faculty positions at Syracuse University and the Australian National University, respectively, and were replaced by Chris Wadsworth and myself. The design and implementation of ML and Edinburgh LCF was finalised and the book "Edinburgh LCF" [15] was written and published.[3] In 1978, the first LCF project finished, Chris Wadsworth went off trekking in the Andes (returning to a permanent position at the Rutherford Appleton Laboratory) and I remained at Edinburgh supported by a postdoctoral fellowship and with a new research interest: hardware verification. After the first LCF project finished, application studies using Edinburgh LCF continued. Milner's student Avra Cohn did a PhD on verifying programming language implementations with LCF and Brian Monahan did a PhD on the theory and mechanisation (using LCF) of datatypes. Researchers who worked with LCF included Jacek Leszczyłowski and Stefan Sokołowski.

4 Cambridge LCF

In 1981, I moved to a permanent position as Lecturer at the University of Cambridge Computer Laboratory. Another LCF project was funded by SERC, split between Edinburgh and Cambridge, with one research assistant at each site. Larry Paulson, recently graduated with a PhD from Stanford, was hired at Cambridge and David Schmidt followed by Lincoln Wallen at Edinburgh[4]. About this time, and in parallel, Gérard Huet ported the Edinburgh LCF code to Lelisp and MacLisp.[5] Paulson and Huet then established a collaboration and did a lot of joint development of LCF by sending each other magnetic tapes in the post. Huet improved and extended ML (his group subsequently developed Caml) and optimised the implementation of LCF's theory files. Paulson improved and simplified much of the Lisp code, and removed some space optimisations that had been necessary to get Edinburgh LCF to run in the limited amount of memory available on the Edinburgh DEC-10 system. Edinburgh LCF ran interpretively, but during Paulson and Huet's collaboration an ML compiler was implemented that provided a speedup by a factor of about twenty.

As part of his LCF work at Cambridge, Paulson made dramatic improvements both to our understanding of how to design and program proof tools and (in col-

[3] The author "Arthur J. Milner" of this book was the result of a bug in the way Springer-Verlag created title pages: they asked for the author's full name on a form and then used it without properly checking its appropriateness.

[4] In the 1980s work at Edinburgh broadened its emphasis from LCF to generic logical frameworks, and then to constructive type theory.

[5] Edinburgh LCF, including the ML interpreter, was implemented in Lisp.

laboration with Gérard Huet) to the implementation of LCF. The now-standard techniques of conversions [39] and theorem continuations [40, Section 8.7.2] were devised by him and then used to implement a large collection of tools. Edinburgh LCF had a monolithic simplifier provided as a primitive. Paulson redesigned and reprogrammed this as a simple and clean derived rule. Inspired by an algorithm from the book *Artificial Intelligence Programming* [3], he implemented a data structure (discrimination nets) for efficiently indexing sets of equations used in rewriting. This became an essential tool for efficiently handling large sets of rules.

Paulson also upgraded the LCF logic to include all the standard constructs of predicate calculus (Scott's logic didn't have disjunction or existential quantification). He also implemented a simple package for managing subgoals on a stack (users of Edinburgh LCF typically managed subgoals by hand by explicitly binding them to ML variables with cumbersome names like `g_2_1_3`). These developments were driven and tested by a number of major case studies, including the formalisation and checking, by Paulson [42], of a proof of correctness of the unification algorithm. The resulting new LCF system was named "Cambridge LCF" and completed around 1985. Paulson did little work on it after that. Mikael Hedlund (of the Rutherford Appleton Laboratory) then ported Cambridge LCF to Standard ML (using a new implementation of ML that he created). The resulting Standard ML based version of Cambridge LCF is documented (with supporting case studies and a tutorial of underlying theory) in Paulson's 1987 book *Logic and Computation* [40].

5 From LCF to HOL

Whilst Paulson was designing and implementing Cambridge LCF, I was mainly concerned with hardware verification. I had been impressed by how the Expansion Theorem of Milner's Calculus of Communicating Systems (CCS) [37] enabled a direct description of the behaviour of a composite agent to be calculated from the parallel composition of its individual components. This seemed like a good paradigm for deriving the behaviour of a digital system from its structural description. I invented a rather *ad hoc* notation (called "LSM" for Logic of Sequential Machines) for denoting sequential machine behaviour, together with a law manipulatively similar to CCS's Expansion Theorem. To provide a proof assistant for LSM, I lashed up a version of Cambridge LCF (called LCF_LSM) that added parsing and pretty-printer support for LSM and provided the expansion-law as an additional axiom scheme [11]. This lash-up worked quite well and even got used outside Cambridge. I used it to verify a toy microprocessor [12], subsequently called Tamarack[6] and the LSM notation was used by a group in the Royal Signals and Radar Establishment (RSRE) to specify the ill-fated Viper processor [5]. During this time Ben Moskowski, who had recently graduated from Stanford, was doing a postdoc at Cambridge. He

[6]The name "Tamarack" is due to Jeff Joyce, who reverified it in HOL [27] and fabricated a chip based on it whilst visiting Xerox Parc as a summer student.

showed me how the terms of LSM could be encoded in predicate calculus in such a way that the LSM expansion-law just becomes a derived rule (i.e. corresponds to a sequence of standard predicate calculus inferences). This approach is both more elegant and rests on a firmer logical foundation, so I switched to it and HOL was born. Incidently, not only was CCS's Expansion Theorem an inspirational stepping stone from LCF via LSM to HOL, but recently things have come 'full circle' and Monica Nesi has used HOL to provide proof support for CCS, including the mechanisation of the Expansion Theorem [38]!

The logic supported by Cambridge LCF has the usual formula structure of predicate calculus, and the term structure of the typed λ-calculus. The type system, due to Milner, is essentially Church's original one [4], but with type variables moved from the meta-language to the object language (in Church's system, a term with type variables is actually a meta-notation – a term-schema – denoting a family of terms, whereas in LCF it is a single polymorphic term). LCF's interpretation of terms as denoting members of Scott domains is overkill for hardware verification where the recursion that arises is normally just primitive recursion. For hardware verification, there is rarely the need for the extra sophistication of fixed-point (Scott) induction; ordinary mathematical induction suffices. The HOL system retains the syntax of LCF, but reinterprets types as ordinary sets, rather than Scott domains.

To enable existing LCF code to be reused, the axioms and rules of inference of HOL were not taken to be the standard ones due to Church. For example, the LCF logic has parallel substitution as a primitive rule of inference (a decision taken after experimentation when Edinburgh LCF was designed), but Church's logic has a different primitive. HOL employs the LCF substitution because I wanted to use the existing efficient code. As a result the HOL logic ended up with a rather *ad hoc* formal basis. Another inheritance from LCF is the use of a natural deduction logic (Church used a Hilbert-style formal system). However, this inheritance is, in my opinion, entirely good.

5.1 The development of HOL

Originally HOL was created for hardware verification at the register transfer level. The modelling technique to be supported represents hardware devices as relations between input and output signals, with internal signals hidden by existential quantification. Signals are represented by functions from time to values (wire states), so that higher-order relations and quantification are necessary. This immediately suggests higher-order logic as an appropriate formalism [13] (the same idea occurred earlier to Keith Hanna [19], the designer of the Veritas hardware verification system). The design of HOL was largely taken 'off the shelf' the theory being classical higher order logic and the implementation being LCF. The development of the system was, at first, primarily driven by hardware verification case studies

The first version of the HOL system was created by modifying the Cambridge LCF parser and pretty-printer to support higher order logic concrete syntax. HOL terms were encoded as LCF constructs in a way designed to support

maximum reuse of LCF code (the encoding did not represent any coherent domain-theoretic semantics). Many aspects of LCF, e.g. typechecking and theory management, were carried over unchanged to HOL. The LCF primitive axioms and inference rules were modified to be correct for higher order logic, and then the higher level theorem proving infrastructure (conversions, tactics, tacticals, subgoal package etc.) was modified to work correctly.

5.2 Primitive definitional principles

The HOL system, unlike LCF, emphasises definition rather than axiom postulation as the primary method of developing theories. Higher order logic makes possible a purely definitional development of many mathematical objects (numbers, lists, trees etc.) and this is supported and encouraged.[7]

The definitional principles provided by HOL evolved during the 1980s. Initially, constants could only be defined via equations of the form $\mathsf{c} = t$, where c was a new name and t a closed term. Types could be defined by a scheme, devised by Mike Fourman, in which new types could be introduced as names for non-empty subsets (specified by a predicate) of existing types. Somewhat later, HOL users at ICL (Roger Jones *et al.*) proposed that 'loose specifications' of constants be allowed. This was implemented by a definitional principle that allowed constants $\mathsf{c}_1, \ldots, \mathsf{c}_n$ to be introduced satisfying a property $P(\mathsf{c}_1, \ldots, \mathsf{c}_n)$, as long as $\exists x_1 \cdots x_n . P(x_1, \ldots, x_n)$ could be proved. This principle is called "constant specification" in HOL.

The first versions of the definitional principles seemed 'obviously correct', but Mark Saaltink and Roger Jones independently noticed that in fact they were unsound in the sense that making a definition did not necessarily preserve consistency.[8] It is easy to fix the problems by adding side conditions to the definitional principles. With the support of DSTO Australia, Dr Andrew Pitts was commissioned to validate HOL's definitional principles. He produced informal proofs that they could not introduce inconsistency [14, Chapter 16].

[7] Axiomatic developments, like group theory, have been attempted with some success [18] (though the LCF/HOL theory mechanism is not ideal for it and various improvements have been proposed). The facilities inherited from LCF for declaring axioms are still available in HOL.

[8] The problem was HOL's treatement of type variables. In the HOL logic, constants can have polymorphic types – i.e. types containing type variables. For example, the identity function I has type $\alpha \rightarrow \alpha$, where α is a type variable. Polymorphic constants are not explicitly parameterised on type variables, so a constant definition $\mathsf{c} = t$ in which there is a free type variable in the type of t but not in the type of c might lead to inconsistency. For example, it is easy to devise [14, page 221] a closed boolean term (i.e. formula), t say, that contains a type variable, α say, and is such that t is true for some instances of α, and false for others. A constant definition $\mathsf{c} = t$ will then be inconsistent, because it will be possible to prove c equals both true and false, by type-instantiating the definition with the two instances. In future versions of HOL it is expected that there will be explicit type variable quantification [31], i.e. terms of the form $\forall \alpha . t$ (where α is a type variable). The right hand side of definitions will be required to be closed with respect to both term and type variables. Melham has shown that this will make defining mechanisms much cleaner and also permit an elegant treatment of type specifications.

5.3 Derived definitional principles

The primitive built-in definitional principles are low-level, but high-level derived principles can be programmed in ML. These take a property one wants a new constant or type to have and then automatically define constants and/or types that have the property. For example, one early derived constant-definition principle defined arithmetical functions to satisfy any user-supplied primitive recursive equation by instantiating the primitive recursion theorem (which was previously proved from the definition of numbers).

Building on ideas originally developed by Milner, Monahan and Paulson for LCF, Melham implemented a derived type-definition principle that converts descriptions of recursive datatypes into primitive definitions and then automatically derives the natural induction and primitive recursion principles for the datatype [29].[9]

It was probably this tool that was most responsible for changing the perception of HOL from being purely a hardware verification system to being a general purpose proof assistant. For example, it became dramatically easier to embed languages inside HOL, by defining a recursive type of abstract syntax trees and a primitive recursive semantic function. Another derived definitional principle, also due to Melham [30], allows inductively defined relations to be specified by a transition system, and then a rule-induction tactic to be automatically generated. This enables operational semantics to be easily defined inside HOL. Other derived definitional principles have also been implemented, including a powerful tool by Konrad Slind for making general recursive definitions of functions [47] (which also runs in Isabelle/HOL) and at least two independent packages for creating quotient types.

5.4 Simplification

Cambridge LCF had a powerful simplifier that dealt separately with term rewriting and formula simplification. In HOL, boolean terms played the role of formulae, so a separate syntactic class of formulae was not needed and hence separate sets of tools for formula and term simplification were no longer necessary. Unfortunately, when I modified Paulson's simplifier for use in HOL, I never got around to making the condition-handling (backchaining) parts of his code work, so HOL ended up without conditional simplification. At the time I justified my laziness with the thought that this would not be important, because in LCF conditional simplification had mainly been used to manage definedness and strictness assumptions arising from the bottom element ($\bot$) of domain theory. Such assumptions do not arise in HOL. However, with hindsight, it is clear that Paulson's superior simplifier would have been very useful for HOL applications, and would have saved many people much low-level proof hacking. Over the years several people have contributed conditional simplification packages as

[9] The various datatype packages in LCF were not definitional – LCF did not distinguish definitions from arbitary axioms. In LCF, new constants and types were characterised by asserting new axioms. Melham's package for HOL automatically generates constant and type definitions and then proves theorems corresponding to the characterising axioms used in LCF.

optional add-ons to HOL, but only recently has conditional simplification been added to the core of the system. It is now part of a new simplifier that integrates rewriting, conversions and decision procedures into a single tool.[10]

6 Versions of HOL

The HOL system has always been very open and many people have contributed to its development. Several groups have built their own versions of the system, essentially starting from scratch. This has good and bad aspects: perhaps some effort has been wasted through duplication and there may be a bit of confusion about which version to use, but on the other hand poor design decisions have been rectified and new ideas (e.g. Mizar mode – see 7.3) have had an opportunity to get incorporated. The latest versions of HOL incorporate ideas from other successful systems, like Isabelle, PVS and Mizar.

6.1 HOL88

The core HOL system became stable in about 1988. A new release that consolidated various changes and enhancements called HOL88 was issued then.[11] We were fortunate to receive support from DSTO Australia to document HOL[12] and from Hewlett Packard to port it from Franz Lisp to Common Lisp (a job very ably done by John Carroll). The current versions of HOL and its documentation are public domain[13] and available on the Internet[14].

6.2 HOL90

In the late 1980's Graham Birtwistle of the University of Calgary started a project to reimplement HOL in Standard ML. The work was done by Konrad Slind, under Birtwistle's direction and with the collaboration of the HOL group at Cambridge. The resulting system, called HOL90, was first released around 1990. It introduced much rationalisation to the legacy-code-based HOL88, and provided a significant performance improvement. During the 1990's Slind continued to develop HOL90 in collaboration with Elsa Gunter of AT&T Bell Laboratories (which has recently become Bell Labs Innovations, Lucent Technologies). HOL90 is now the main version of HOL in use around the world, though users of HOL88 still linger on.

[10] HOL's new simplifier uses ideas from Isabelle and is being developed by Donald Syme.

[11] The release was prepared by Tom Melham and paid for by ICL.

[12] A documentation standard was designed and then each of the several hundred ML functions comprising HOL was priced at £5 or £10. Members of the Cambridge HOL users community were then invited to write documentation for money. The whole job was done painlessly in a few weeks.

[13] A licence agreement for HOL is available, but signing it is optional.

[14] `http://lal.cs.byu.edu/lal/hol-documentation.html`

6.3 ProofPower

In parallel with the development of HOL90, ICL created their own commercial version of HOL, now called ProofPower[15]. This was targetted at in-house and commercial use, especially for security applications. ProofPower supports exactly the same logic as the other HOL systems, but has different proof infrastructure that evolved to meet the needs of the targetted applications (e.g. customised theorem-proving support for the Z notation and a verification condition generator for the Z - (SPARK) Ada compliance notation).

6.4 Isabelle/HOL

Besides HOL, several other LCF-style proof assistants were developed with ML as their metalanguage (in some cases code from LCF was used as the starting point). These include a proof system for the Calculus of Contructions [10, 9], Nuprl [8], and a proof system for Martin Löf type theory [44]. These applied Milner's LCF methodology to widely different logics. To try to provide a systematic implementation methodology for 'LCF-style' systems, Paulson developed the generic prover Isabelle [41, 43]. This provided a metalogic in which the proof rules of object logics can be described declaratively (in LCF and HOL, rules are represented as ML programs – i.e. they are implemented rather than specified). At first sight Isabelle seems to provide a similar collection of proof tools as HOL, but the way they work is quite different. Metalogic rules are composed using a meta-inference rule based on higher order unification (resolution).[16] Forward and backward proof in HOL corresponds to special cases of rule composition in Isabelle. However, Milner's key idea of using ML's abstract types to ensure that theorems can only be obtained by allowable combinations of allowable rules is retained, and lifted to the metalogic level. One of the object logics developed for Isabelle by Paulson was the HOL logic. The resulting Isabelle/HOL system has a somewhat different 'look and feel' to the original HOL system, due to Isabelle's different proof infrastructure. It provides better general logic automation than HOL (via its customisable simplifier and first-order theorem proving tools)[17] and some HOL users have migrated to it.

6.5 HOL Light

Recently John Harrison and Konrad Slind have entirely reworked the design of HOL to, among other things, rationalise the primitive constants, axioms and rules of inference. For example, the logic is initially taken to be constructive, and only after considerable proof infrastructure has been defined are non-constructive principles added. This new version of HOL is called "HOL Light". It is implemented in Caml Light and runs on modest platforms (e.g. standard PCs). It is faster than the Lisp-based HOL88, but a bit slower than HOL90 run-

[15] `http://www.to.icl.fi/ICLE/ProofPower/index.html`

[16] Higher order unification is built into Isabelle, thus Isabelle's 'trusted core' is considerably more complex than HOL's.

[17] Tools based on the Isabelle simplifier, and others comparable to its first order automation facilities, are now part of HOL90 and HOL Light.

ning in modern implementations of Standard ML. HOL Light contains many new facilities, including automatic provers that separate proof search from checking. It also provides 'Mizar mode' (see 7.3) as well as the normal goal-oriented proof styles.

7 Features of HOL

HOL is characterised by a number of key features: a simple core logic, LCF-style 'full expansiveness', support for a growing diversity of proof styles and a large corpus of user-supplied theories and proof tools.

7.1 The core logic

There are only four separate kinds of primitive terms: variables, constants, function applications and λ-abstractions. Using standard techniques, other useful notations are supported on top of these by the parser and pretty-printer. For example, quantifications $Qx.t$ (where Q is $\forall$ or $\exists$) are encoded (following Church) as the application of a constant to an abstraction – i.e. as $Q(\lambda x.t)$, and local variable binding $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ is equivalent (following Landin) to $(\lambda x.t_2)t_1$. Thus all variable binding is reduced to λ-binding.

Notations including infixes, conditional terms, set abstractions, restricted quantifications (i.e. quantification over predicate subtypes), tuple notation, and tupled variable binding (e.g. $\lambda(x, y).t$) are considered to be derived forms (i.e. 'syntactic sugar'). This strategy of translating away complex notations has worked pretty well. It means that procedures designed to process all terms need often only consider four cases (variables, constants, applications and abstractions) and so can be short and semantically transparent.[18] On the other hand, the encoding of everything into just variables, constants, applications and abstractions makes the computation of the natural constituents of constructs expensive. This makes writing interfaces (e.g. pretty-printing) quite complex.[19] A particular danger with reducing complex notations to a simple core is that what the user sees printed can be remote from what the inference mechanisms actually process. Errors in the parsing and pretty printing interface can be just as dangerous as errors in the infererence rules (e.g. consider an interface that translated *true* to *false* with a pretty-printer that inverted this translation). This problem has been worried about a lot. One approach to minimising the dangers is to use trustworthy tools that generate parsers and pretty-printers from a declarative

[18] An example of this is Grundy's window inference system for hierarchical transformational reasoning (e.g. program refinement) [17]. This provides an environment on top of HOL for pointing at subterms and then transforming them 'in place' (subject, of course, to context-dependent side-conditions). Grundy was able to base his system on three primitives (one each for the function and argument constituents of applications and one for the bodies of λ-abstractions) and then have ML programs automatically compose guaranteed-sound window rules for transforming arbitrarily complex terms as syntax-directed compositions of the three primitives. If HOL had had different primitive terms for each user-level notation, then Grundy would have had to hand-build a primitive window-rule for each of them.

[19] For example, it is more work than it looks to extract the constituents x, y, t_1 and t_2 from $\texttt{let}\ (x, y)\ =\ t_1\ \texttt{in}\ t_2$, which is actually parsed to $\texttt{LET}(\texttt{UNCURRY}(\lambda x.\texttt{UNCURRY}(\lambda y.\lambda z.t_2)))t_1$.

input. A recent example of such tool is CLaReT, due to Richard Boulton.

7.2 Full expansiveness

The LCF approach to theorem proving is "fully expansive" in that all proofs are expanded into sequences of primitive inferences. At first sight this seems to be very inefficient and it has been argued that it is incompatible with acceptable performance. However, a whole programming methodology has evolved for programming efficient derived rules and tactics. For example, decision procedures for particular classes of problems have been developed. Although these expand out to primitive inferences, they are surprisingly efficient. HOL has such decision procedures for tautology checking (based on BDDs and due to John Harrison) and for a subset of arithmetic (due to Richard Boulton) which users find adequately fast and powerful for many applications. An important efficiency improving technique is to exploit the expressive power of higher order logic by encoding as single theorems facts that would have to be derived rules of inference (e.g. theorem schemas) in first order logic. Thus time-consuming repetitions of sequences of proof steps can often be avoided by proving a general theorem once and then instantiating it many times. Another programming technique is to separate out proof search from proof checking. An ML program, or even an external oracle (like a C-coded tautology checker or an algebra system [22]), can be used to find a proof. The result is validated by formal inference inside the HOL logic. One way of packaging (and automating) this separation is Boulton's technique of lazy theorems [2].[20]

There have been many developments in the implementation and use of tactics over the last twenty years. It is remarkable that Milner's original concept has turned out to be sufficiently general to support them.

7.3 A diversity of proof styles

Current releases of HOL support forward proof and goal directed proof. The latter via a stack-based subgoal package provided for Cambridge LCF by Paulson. Other styles are available as libraries (e.g. window inference and tree based subgoal package due to Sara Kalvala). One way that HOL is expected to evolve is as a tool with a fixed logic but with an ever growing variety of built-in proof styles. For example, recently a lot of excitement has been generated by the remarkable Mizar system[21] in which proofs are constructed by refining arguments expressed in a natural-language-like textbook style. It seems that this forward style is better for some things and a goal-oriented style for others. In particular, a goal-oriented style works well for verifying complex artifacts (e.g. microproces-

[20] A lazy theorem is a pair consisting of a term, together with a procedure for proving it. Lazy theorems can be created using a (fast) non-expansive decision procedure, and supplying a (slow) expansive procedure as the proof function. Such lazy theorems can be manipulated (with some additional effort) much like proper theorems, by composing the proof functions. At some point they need to be coerced into proper theorems (by running the proof part), but this can be postponed to a convenient moment – e.g. coffee time. With lazy theorems one gets the interactive speed of conventional decision procedures with the security of full-expansiveness.

[21] `http://web.cs.ualberta.ca:80/~piotr/Mizar/`

sors) where the proof can be generated via specialised algorithms, whereas the forward Mizar style seems better for developing general mathematical theories (e.g. algebra, functional analysis, topology). Many applications (e.g. floating point verification, cryptography, signal processing) require a general mathematical infrastructure to be brought to bear via problem-specific algorithms [20]. It is thus useful to provide the option of using a Mizar style for developing theories and a goal-oriented style for deploying them. To this end, John Harrison has recently added support for 'Mizar mode' in HOL [21].

7.4 Libraries and other user supplied contributions

To enable theories (and other utility code) to be shared, HOL has a rudimentary library facility. This provides a file structure and documentation format for self contained HOL developments (usually a combination of theories and theorem proving tools). Over the years many libraries were supplied by users from around the world. Although the core HOL system remained fairly stable, the set of libraries grew. Libraries currently distributed with HOL include arithmetic and tautology decision procedures, a development of group theory, a package to support inductively defined relations, theories of integers and reals (only natural numbers are predefined in the core system) theories of n-bit words, character strings, general lists and sets, well-ordered sets (transfinite induction etc.), support for UNITY and Hoare-style programming logics, tools for hardware verification and program refinement (window inference).

Libraries are intended to be fairly polished and documented to a high standard. Also distributed with HOL are "contributions", which are subject to minimal quality control. Currently distributed contributions include: CSP trace theory, proof tools for associative-commutative unification, tactics implementing Boyer and Moore's automatic proof heuristics (as described in the book *A Computational Logic*), the proof of a sequential multiplier (used as a benchmark for HOL), theories of infinite state automata, rules for simplifying conditional expressions, the definition of fixedpoints and the derivation of Scott induction, tools to support language embedding in higher order logic, Knuth-Bendix completion as a derived rule, various enhancements to the recursive types package (e.g. for nested and mutual recursion), the formalisation of a significant part of the definition of Standard ML, the application of a database query language to HOL, a compiler for generating efficient conversions from sets of equations, theories supporting partial functions and a hyperlinked guide to HOL theories.

Considering the way it was built, there have been relatively few bugs in HOL. Because the system is completely open, early bugfixes were often done by intrepid users[22] and then mailed to me for inclusion in future releases.

[22]Subtle Lisp bugs in early versions of HOL were fixed by David Shepherd of Inmos and Ton Kalker of Philips.

8 Conclusions

Over the last ten years the scale of what can be proved with HOL (as with other provers) has increased dramatically. In the early 1980's the verification of simple systems with a few registers and gates was considered significant. By the late 1980s simple microprocessors [6, 7, 16, 49] and networking hardware [23] was being verified and by the mid 1990s complex hardware structures (e.g. pipelines) were being analysed and many non-hardware applications were being attempted, including program and algorithm verification, support for process algebras and the mechanisation of classical mathematical analysis (i.e. the theory of limits, differentiation and integration). There are nearly 400 publications and 30 dissertations listed in Tom Melham's HOL bibliography[23]. There is no space to summarise all the work that has been done with HOL here, but recent work can be found in the proceedings of the conference now called[24] *Theorem Proving in Higher Order Logics* [28, 32, 45, 48] or in special issues of *The Computer Journal* [26] and *Formal Methods in System Design* [24, 25].

One noteworthy niche has been the embedding of programming and hardware description languages in HOL.[25] This was opened up by Melham's packages for recursive types and inductive relations (and enhancements to these by David Shepherd, Elsa Gunter and John Harrison). One result of this particular focus of activity has been the accumulation of wisdom (and jargon) concerning language embeddings and the development of syntax directed support tools. These tools both automate the construction of embeddings and help reduce the dangers discussed at the end of 7.1. An embedding that previously might have taken weeks/months can now often be done in hours/days.

For a proof assistant, HOL has a large user community. There have been nine HOL Users Workshops, which have gradually evolved from informal get-togethers to elaborate international meetings with refereed papers. In the future, these conferences will be targetted at the whole higher order logic theorem proving community, rather than just HOL users.

To harvest the accumulated wisdom of the whole proof assistant community and to position HOL for the results of ML2000[26], a public initiative called

[23] `http://www.dcs.glasgow.ac.uk/~tfm/hol-bib.html`

[24] Previous names include *International Conference on Higher Order Logic Theorem Proving and Its Applications*.

[25] The embedding of Standard ML has been extensively studied by Elsa Gunter, Myra VanInwegen and Donald Syme, the embedding of a large subset of C is currently being undertaken by Michael Norrish at Cambridge. Embeddings of subsets of the hardware description languages ELLA, NODEN, Silage, VHDL and Verilog have been done (or are in progress). Steve Brackin has embedded a language of security assertions and provided a GUI that hides HOL, all input and output being in terms of the security language. Flemming Andersen (partly supported by the Technical University at Lyngby in Denmark and partly by TFL) has embedded the UNITY language and provided theorem proving support for it. Joakim Von Wright and Thomas Långbacka have embedded Lamport's Temporal Logic of Actions (TLA).

[26] Under the slogan "ML2000", a cabal from the ML community have started investigating ideas for a major revision of ML. Their aim is to design a language in the ML tradition that builds on current experience with Standard ML and Caml as well as recent theoretical work on language design and semantics (such as type systems for object oriented programming).

"HOL2000" has been launched.[27] Only time will tell how this evolves. The end of the century should be an exciting time for HOL – and all thanks to Robin Milner!

9 Acknowledgements

Richard Boulton, Paul Curzon, Jim Grundy, John Harrison, Tom Melham, Robin Milner, Larry Paulson and Konrad Slind provided help in writing this paper and/or suggestions for improving it. An anonymous referee suggested several substantial improvements and clarifications.

References

[1] Graham Birtwistle and P. A. Subrahmanyam, editors. *Current Trends in Hardware Verification and Automated Theorem Proving*. Springer, 1989.

[2] R. J. Boulton. Lazy techniques for fully expansive theorem proving. *Formal Methods in System Design*, 3(1/2):25–47, August 1993.

[3] E. Charniak, C. K. Riesbeck, and D. V. McDermott. *Artificial Intelligence Programming*. Lawrence Erlbaum Associates, 1980.

[4] A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5:56–68, 1940.

[5] Avra Cohn. A proof of correctness of the VIPER microprocessor: The first level. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 27–71. Kluwer Academic Publishers, 1988.

[6] Avra Cohn. Correctness properties of the Viper block model: The second level. In Birtwistle and Subrahmanyam [1], pages 1–91.

[7] Avra Cohn. The notion of proof in hardware verification. *Journal of Automated Reasoning*, 5(2):127–139, 1989.

[8] R. L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.

[9] Thierry Coquand and Gérard Huet. Constructions: A higher order proof system for mechanizing mathematics. In Bruno Buchberger, editor, *EUROCAL '85*, volume 203 of *Lecture Notes in Computer Science*, pages 151–184, Berlin, 1985. Springer-Verlag.

[10] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[11] M. J. C. Gordon. LCF_LSM: A system for specifying and verifying hardware. Technical Report 41, University of Cambridge Computer Laboratory, 1983.

[12] M. J. C. Gordon. Proving a computer correct. Technical Report 42, University of Cambridge Computer Laboratory, 1983.

[13] M. J. C. Gordon. Why higher-order logic is a good formalism for specifying and verifying hardware. In Milne and Subrahmanyam [33], pages 153–177.

[14] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.

[15] 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.

[16] B. T. Graham. *The SECD Microprocessor: A Verification Case Study*. Kluwer, 1992.

[27] `http://lal.cs.byu.edu/lal/hol2000/hol2000.html`

[17] Jim Grundy. A window inference tool for refinement. In Clifford Bryn Jones, B. Tim Denvir, and Roger C. F. Shaw, editors, *Proceedings of the 5th Refinement Workshop*, Workshops in Computing, pages 230–254, Lloyd's Register, London, January 1992. BCS FACS, Springer-Verlag.

[18] E. L. Gunter. Doing algebra in simple type theory. Technical Report MS-CIS-89-38, Dept. of Computer and Information Science, Moore School of Engineering, University of Pennsylvania, June 1989.

[19] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: a case study. In Milne and Subrahmanyam [33], pages 179–213.

[20] John Harrison. Constructing the real numbers in HOL. *Formal Methods in System Design*, 5:35–59, 1994.

[21] John Harrison. A mizar mode for HOL. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 203–220, Turku, Finland, 1996. Springer-Verlag.

[22] John Harrison and Laurent Théry. Extending the HOL theorem prover with a computer algebra system to reason about the reals. In Joyce and Seger [28], pages 174–184.

[23] J. Herbert. Case study of the Cambridge Fast Ring ECL chip using HOL. Technical Report 123, Computer Laboratory, University of Cambridge, UK, 1988.

[24] *Formal Methods in System Design*, volume 3, number 1/2. Special issue on Higher Order Logic Theorem Proving and its Applications, August 1993.

[25] *Formal Methods in System Design*, volume 5, number 1/2. Special issue on Higher Order Logic Theorem Proving and its Applications, July/August 1993.

[26] *The Computer Journal*, volume 38, number 2. Special issue on Higher Order Logic Theorem Proving and its Applications, 1995.

[27] G. Birtwistle J. Joyce and M. J. C. Gordon. Proving a computer correct in higher order logic. Technical Report 100, University of Cambridge Computer Laboratory, 1986.

[28] Jeffrey J. Joyce and Carl Seger, editors. volume 780 of *Lecture Notes in Computer Science*, UBC, Vancouver, Canada, 1993. Springer-Verlag.

[29] T. F. Melham. Automating recursive type definitions in higher order logic. In Birtwistle and Subrahmanyam [1], pages 341–386.

[30] T. F. Melham. A package for inductive relation definitions in hol. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 32–37. IEEE Computer Society Press, August 1991.

[31] T. F. Melham. The HOL logic extended with quantification over type variables. *Formal Methods in System Design*, 3(1/2):7–24, August 1993.

[32] Thomas F. Melham and Juanito Camilleri, editors. *Higher Order Logic Theorem Proving and Its Applications: 7th International Workshop*, LNCS 859. Springer, September 1994.

[33] G. Milne and P. A. Subrahmanyam, editors. *Formal Aspects of VLSI Design*. North-Holland, 1986.

[34] R. Milner. Logic for computable functions; description of a machine implementation. Technical Report STAN-CS-72-288, A.I. Memo 169, Stanford University, 1972.

[35] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[36] R. Milner and R. Weyhrauch. Proving compiler correctness in a mechanized logic. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence 7*, pages 51–70. Edinburgh University Press, 1972.

[37] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[38] Monica Nesi. A formalization of the process algebra CCS in higher order logic. Technical Report 278, University of Cambridge, Computer Laboratory, December 1992.

[39] L. C. Paulson. A higher-order implementation of rewriting. *Science of Computer Programming*, 3:119–149, 1983.

[40] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[41] L. C. Paulson. Isabelle: The next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science*, pages 361–386. Academic Press, 1990.

[42] Lawrence C. Paulson. Verifying the unification algorithm in LCF. *Science of Computer Programming*, 5:143–170, 1985.

[43] Lawrence C. Paulson. *Isabelle: A Generic Theorem Prover*. Springer, 1994. LNCS 828.

[44] K. Petersson. A programming system for type theory. Technical Report 21, Department of Computer Science, Chalmers University, Göteborg, Sweden, 1982.

[45] E. Thomas Schubert, Phillip J. Windley, and James Alves-Foss, editors. *Higher Order Logic Theorem Proving and Its Applications 8th International Workshop*, LNCS 971. Springer, September 1995.

[46] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Annotated version of the 1969 manuscript.

[47] Konrad Slind. Function definition in higher order logic. In *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96*, number 1125 in Lecture Notes in Computer Science, Turku, Finland, August 1996. Springer Verlag.

[48] Joakim von Wright, Jim Grundy, and John Harrison, editors. volume 1125 of *Lecture Notes in Computer Science*, Turku, Finland, 1996. Springer-Verlag.

[49] P. J. Windley. The practical verification of microprocessor designs. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 32–37. IEEE Computer Society Press, August 1991.

Mike Gordon
October 31, 1996