HOL Light: An Overview

John Harrison

Intel Corporation, JF1-13
2111 NE 25th Avenue
Hillsboro OR 97124
johnh@ichips.intel.com

Abstract. HOL Light is an interactive proof assistant for classical higher-order logic, intended as a clean and simplified version of Mike Gordon's original HOL system. Theorem provers in this family use a version of ML as both the implementation and interaction language; in HOL Light's case this is Objective CAML (OCaml). Thanks to its adherence to the so-called 'LCF approach', the system can be extended with new inference rules without compromising soundness. While retaining this reliability and programmability from earlier HOL systems, HOL Light is distinguished by its clean and simple design and extremely small logical kernel. Despite this, it provides powerful proof tools and has been applied to some non-trivial tasks in the formalization of mathematics and industrial formal verification.

1 LCF, HOL and HOL Light

Both HOL Light and its implementation language OCaml can trace their origins back to Edinburgh LCF, developed by Milner and his research assistants in the 1970s [6]. The LCF approach to theorem proving involves two key ideas:

- All proofs are ultimately performed in terms of a small set of primitive inferences, so provided this small logical 'kernel' is correct the results should be reliable.
- The entire system is embedded inside a powerful functional programming language, which can be used to program new inference rules. The type discipline of the programming language is used to ensure that these ultimately reduce to the primitives.

The original Edinburgh LCF was a theorem prover for Scott's Logic of Computable Functions [16], hence the name LCF. But as emphasized by Gordon [4], the basic LCF approach is applicable to any logic, and now there are descendents implementing a variety of higher order logics, set theories and constructive type theories. In particular, members of the HOL family [5] implement a version of classical higher order logic, hence the name HOL. They take the LCF approach a step further in that all theory developments are pursued 'definitionally'. New mathematical structures, such as the real numbers, may be defined only by exhibiting a model for them in the existing theories (say as Dedekind

S. Berghofer et al. (Eds.): TPHOLs 2009, LNCS 5674, pp. 60–66, 2009.

cuts of rationals). New constants may only be introduced by definitional extension (roughly speaking, merely being a shorthand for an expression in the existing theory). This fits naturally with the LCF style, since it ensures that all extensions, whether of the deductive system or the mathematical theories, are consistent per construction.

2 HOL Light's Logical Foundations

HOL Light's logic is simple type theory [1,2] with polymorphic type variables. The terms of the logic are those of simply typed lambda calculus, with formulas being terms of boolean type, rather than a separate category. Every term has a single welldefined type, but each constant with polymorphic type gives rise to an infinite family of constant terms. There are just two primitive types: `bool` (boolean) and `ind` (individuals), and given any two types σ and τ one can form the function type $\sigma \to \tau$.[1]

For the core HOL logic, there is essentially only one predefined logical constant, equality ($=$) with polymorphic type $\alpha \to \alpha \to$ bool. However to state one of the mathematical axioms we also include another constant $\varepsilon : (\alpha \to \text{bool}) \to \alpha$, explained further below. For equations, we use the conventional concrete syntax $s = t$, but this is just surface syntax for the λ-calculus term $((=)s)t$, where juxtaposition represents function application. For equations between boolean terms we often use $s \Leftrightarrow t$, but this again is just surface syntax.

The HOL Light deductive system governs the deducibility of one-sided sequents $\Gamma \vdash p$ where p is a term of boolean type and Γ is a set (possibly empty) of terms of boolean type. There are ten primitive rules of inference, rather similar to those for the internal logic of a topos [14].

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \texttt{TRANS}$$

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \texttt{MK\_COMB}$$

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\ s) = (\lambda x.\ t)}\ \texttt{ABS}$$

$$\frac{}{\vdash (\lambda x.\ t)x = t}\ \texttt{BETA}$$

$$\frac{}{\{p\} \vdash p}\ \texttt{ASSUME}$$

[1] In Church's original notation, also used by Andrews, these are written o, ι and $\tau\sigma$ respectively. Of course the particular concrete syntax has no logical significance.

$$\frac{\Gamma \vdash p \Leftrightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \texttt{EQ\_MP}$$

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p \Leftrightarrow q}\ \texttt{DEDUCT\_ANTISYM\_RULE}$$

$$\frac{\Gamma[x_1, \ldots, x_n] \vdash p[x_1, \ldots, x_n]}{\Gamma[t_1, \ldots, t_n] \vdash p[t_1, \ldots, t_n]}\ \texttt{INST}$$

$$\frac{\Gamma[\alpha_1, \ldots, \alpha_n] \vdash p[\alpha_1, \ldots, \alpha_n]}{\Gamma[\gamma_1, \ldots, \gamma_n] \vdash p[\gamma_1, \ldots, \gamma_n]}\ \texttt{INST\_TYPE}$$

In `MK_COMB` it is necessary for the types to agree so that the composite terms are well-typed, and in `ABS` it is required that the variable x not be free in any of the assumptions Γ, while our notation for term and type instantiation assumes capture-avoiding substitution. All the usual logical constants are defined in terms of equality. The conventional syntax $\forall x.\, P[x]$ for quantifiers is surface syntax for $(\forall)(\lambda x.\, P[x])$, and we also use this 'binder' notation for the ε operator.

$$\begin{aligned}
\top &=_{def} (\lambda p.\, p) = (\lambda p.\, p)\\
\wedge &=_{def} \lambda p.\, \lambda q.\, (\lambda f.\, f\ p\ q) = (\lambda f.\, f\ \top\ \top)\\
\Longrightarrow &=_{def} \lambda p.\, \lambda q.\, p \wedge q \Leftrightarrow p\\
\forall &=_{def} \lambda P.\, P = \lambda x.\, \top\\
\exists &=_{def} \lambda P.\, \forall q.\, (\forall x.\, P(x) \Longrightarrow q) \Longrightarrow q\\
\vee &=_{def} \lambda p.\, \lambda q.\, \forall r.\, (p \Longrightarrow r) \Longrightarrow (q \Longrightarrow r) \Longrightarrow r\\
\bot &=_{def} \forall p.\, p\\
\neg &=_{def} \lambda p.\, p \Longrightarrow \bot\\
\exists! &=_{def} \lambda P.\, \exists P \wedge \forall x.\, \forall y.\, P\ x \wedge P\ y \Longrightarrow x = y
\end{aligned}$$

These definitions allow us to derive all the usual (intuitionistic) natural deduction rules for the connectives in terms of the primitive rules above. All of the core 'logic' is derived in this way. But then we add three mathematical axioms:

- The axiom of extensionality, in the form of an eta-conversion axiom `ETA_AX`: $\vdash (\lambda x.\, t\ x) = t$. We could have considered this as part of the core logic rather than a mathematical axiom; this is largely a question of taste.
- The axiom of choice `SELECT_AX`, asserting that the Hilbert operator ε is a choice operator: $\vdash P\ x \Longrightarrow P((\varepsilon)P)$. It is only from this axiom that we can deduce that the HOL logic is classical [3].
- The axiom of infinity `INFINITY_AX`, which implies that the type `ind` is infinite.

In addition, HOL Light includes two principles of definition, which allow one to extend the set of constants and the set of types in a way guaranteed to preserve consistency. The rule of constant definition allows one to introduce a new constant c and an axiom $\vdash c = t$, subject to some conditions on free variables and polymorphic types in t, and provided no previous definition for c has been introduced. All the definitions of the logical connectives above are introduced in this way. Note that this is 'object-level' definition: the constant and its defining axiom exists in the object logic. Nevertheless, the definitional principles are designed so that they always give a conservative (in particular consistency-preserving) extension of the logic.

3 The HOL Light Implementation

Like other LCF provers, HOL Light is in essence simply a large ML program that defines data structures to represent logical entities, together with a suite of functions to manipulate them in a way guaranteeing soundness. The most important data structures belong to one of the datatypes `hol_type`, `term` and `thm`, which represent types, terms (including formulas) and theorems respectively. The user can write arbitrary programs to manipulate these objects, and it is by creating new objects of type `thm` that one proves theorems. HOL's notion of an 'inference rule' is simply a function with return type `thm`.

In order to guarantee logical soundness, however, all these types are encapsulated as abstract types. In particular, the only way of creating objects of type `thm` is to apply one of the 10 very simple inference rules listed above or to make a new term or type definition. Thus, whatever the circuitous route by which one arrives at it, the validity of any object of type `thm` rests only on the correctness of the rather simple primitive rules (and of course the correctness of OCaml's type checking etc.).

To illustrate how inference rules are represented as functions in OCaml, suppose that two theorems of the form $\Gamma \vdash s = t$ and $\Delta \vdash t = u$ have already been proved and bound to the OCaml variables `th1` and `th2` respectively. In abstract logical terms, the rule `TRANS` ensures that the theorem $\Gamma \cup \Delta \vdash s = u$ is derivable. In terms of the HOL implementation, one can apply the OCaml function `TRANS`, of type `thm -> thm -> thm`, to these two theorems as arguments, and hence bind name `th3` to that theorem $\Gamma \cup \Delta \vdash s = u$:

```
let th3 = TRANS th1 th2;;
```

One doesn't normally use such low-level rules much, but instead interacts with HOL via a series of higher-level derived rules, using built-in parsers and printers to read and write terms in a more natural syntax. For example, if one wants to bind the name `th6` to the theorem of real arithmetic that when $|c - a| < e$ and $|b| \leq d$ then $|(a + b) - c| < d + e$, one simply does:

```
let th6 = REAL_ARITH
 `abs(c - a) < e ∧ abs(b) <= d ⟹ abs((a + b) - c) < d + e`;;
```

If the purported fact in quotations turns out not to be true, then the rule will fail by raising an exception. Similarly, any bug in the derived rule (which represents several dozen pages of code written by the present author) would lead to an exception.[2] But we can be rather confident in the truth of any theorem that is returned, since it must have been created via applications of primitive rules, even though the precise choreographing of these rules is automatic and of no concern to the user. What's more, users can write their own special-purpose proof rules in the same style when the standard ones seem inadequate — HOL is fully programmable, yet retains its logical trustworthiness when extended by ordinary users.

Among the facilities provided by HOL is the ability to organize proofs in a mixture of forward and backward steps, which users often find more congenial. The user invokes so-called *tactics* to break down the goal into more manageable subgoals. For example, in HOL's inbuilt foundations of number theory, the proof that addition of natural numbers is commutative is written as follows (the symbol ∀ means 'for all'):

```
let ADD_SYM = prove
 (‘∀m n. m + n = n + m‘,
  INDUCT_TAC THEN
  ASM_REWRITE_TAC[ADD_CLAUSES]);;
```

The tactic `INDUCT_TAC` uses mathematical induction to break the original goal down into two separate goals, one for $m = 0$ and one for $m + 1$ on the assumption that the goal holds for m. Both of these are disposed of quickly simply by repeated rewriting with the current assumptions and a previous, even more elementary, theorem about the addition operator. The identifier `THEN` is a so-called *tactical*, i.e. a function that takes two tactics and produces another tactic, which applies the first tactic then applies the second to any resulting subgoals (there are two in this case).

For another example, we can prove that there is a unique x such that $x = f(g(x))$ if and only if there is a unique y with $y = g(f(y))$ using a single standard tactic `MESON_TAC`, which performs model elimination [15] to prove theorems about first order logic with equality. As usual, the actual proof under the surface happens by the standard primitive inference rules.

```
let WISHNU = prove
 (‘(∃!x. x = f (g x)) ⇔ (∃!y. y = g(f y))‘,
  MESON_TAC[]);;
```

These and similar higher-level rules certainly make the construction of proofs manageable whereas it would be almost unbearable in terms of the primitive rules alone. Nevertheless, we want to dispel any false impression given by the simple examples above: proofs often require long and complicated sequences of rules. The

[2] Or possibly to a true but different theorem being returned, but this is easily guarded against by inserting sanity checks in the rules.

construction of these proofs often requires considerable persistence. Moreover, the resulting proof scripts can be quite hard to read, and in some cases hard to modify to prove a slightly different theorem. One source of these difficulties is that the proof scripts are highly procedural — they are, ultimately, OCaml programs, albeit of a fairly stylized form. There are arguments in favour of a more declarative style for proof scripts, but the procedural approach has its merits too, particularly in applications using specialized derived inference rules [9].

4 HOL Light Applications

Over the years, HOL Light has been used for a wide range of applications, and in concert with this its library of pre-proved formalized mathematics and its stock of more powerful derived inference rules have both been expanded. As well as the usual battery of automated techniques like first-order reasoning and linear arithmetic, HOL Light has been used to explore and apply unusual and novel decision procedures [12,17].

In verification, HOL Light has been used at Intel to verify a number of complex floating-point algorithms including division, square root and transcendental functions [11]. HOL Light seems well-suited to applications like this. It has a substantial library of formalized real analysis, which is used incessantly when justifying the correctness of such algorithms. The flexibility and programmability that the LCF approach affords are also important here since one can write custom derived rules for special tasks like accumulating bounds on rounding errors or enumerating the solutions to Diophantine equations of special kinds.

As for the formalization of mathematics, HOL Light has from the very beginning had a useful formalization of real analysis [10]. More recently this has been substantially developed to cover multivariate analysis in Euclidean space and complex analysis. As well as the miscellany of theorems noted in the list at `http://www.cs.ru.nl/~freek/100/`, HOL Light has been used to formalize some particularly significant results such as the Jordan Curve Theorem [8] and the Prime Number Theorem [13]. HOL Light is also heavily used in the Flyspeck Project [7] to formalize the proof of the Kepler sphere-packing conjecture, possibly the most ambitious formalization project to date.

References

1. Andrews, P.B.: An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof. Academic Press, London (1986)
2. Church, A.: A formulation of the Simple Theory of Types. Journal of Symbolic Logic 5, 56–68 (1940)
3. Diaconescu, R.: Axiom of choice and complementation. Proceedings of the American Mathematical Society 51, 176–178 (1975)
4. Gordon, M.J.C.: Representing a logic in the LCF metalanguage. In: Néel, D. (ed.) Tools and notions for program construction: an advanced course, pp. 163–185. Cambridge University Press, Cambridge (1982)

5. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: a theorem proving environment for higher order logic. Cambridge University Press, Cambridge (1993)
6. Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF. LNCS, vol. 78. Springer, Heidelberg (1979)
7. Hales, T.C.: Introduction to the Flyspeck project. In: Coquand, T., Lombardi, H., Roy, M.-F. (eds.) Mathematics, Algorithms, Proofs. Dagstuhl Seminar Proceedings, vol. 05021. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany (2006)
8. Hales, T.C.: The Jordan curve theorem, formally and informally. The American Mathematical Monthly 114, 882–894 (2007)
9. Harrison, J.: Proof style. In: Giménez, E., Paulin-Mohring, C. (eds.) TYPES 1996. LNCS, vol. 1512, pp. 154–172. Springer, Heidelberg (1998)
10. Harrison, J.: Theorem Proving with the Real Numbers. Springer, Heidelberg (1998); Revised version of author's PhD thesis
11. Harrison, J.: Floating-point verification using theorem proving. In: Bernardo, M., Cimatti, A. (eds.) SFM 2006. LNCS, vol. 3965, pp. 211–242. Springer, Heidelberg (2006)
12. Harrison, J.: Verifying nonlinear real formulas via sums of squares. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 102–118. Springer, Heidelberg (2007)
13. Harrison, J.: Formalizing an analytic proof of the Prime Number Theorem (dedicated to Mike Gordon on the occasion of his 60th birthday). Journal of Automated Reasoning (to appear, 2009)
14. Lambek, J., Scott, P.J.: Introduction to higher order categorical logic. Cambridge studies in advanced mathematics, vol. 7. Cambridge University Press, Cambridge (1986)
15. Loveland, D.W.: Mechanical theorem-proving by model elimination. Journal of the ACM 15, 236–251 (1968)
16. Scott, D.: A type-theoretical alternative to ISWIM, CUCH, OWHY. Theoretical Computer Science 121, 411–440 (1993); Annotated version of a 1969 manuscript
17. Solovay, R.M., Arthan, R., Harrison, J.: Some new results on decidability for elementary algebra and geometry. ArXiV preprint 0904.3482 (2009); submitted to Annals of Pure and Applied Logic,
`http://arxiv.org/PS_cache/arxiv/pdf/0904/0904.3482v1.pdf`