Towards self-verification of HOL Light

John Harrison

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

Abstract. The HOL Light prover is based on a logical kernel consisting of about 400 lines of mostly functional OCaml, whose complete formal verification seems to be quite feasible. We would like to formally verify (i) that the abstract HOL logic is indeed correct, and (ii) that the OCaml code does correctly implement this logic. We have performed a full verification of an imperfect but quite detailed model of the basic HOL Light core, without definitional mechanisms, and this verification is entirely conducted with respect to a set-theoretic semantics within HOL Light itself. We will duly explain why the obvious logical and pragmatic difficulties do not vitiate this approach, even though it looks impossible or useless at first sight. Extension to include definitional mechanisms seems straightforward enough, and the results so far allay most of our practical worries.

1 Introduction: quis custodiet ipsos custodes?

Mathematical proofs are subjected to peer review before publication, but there are plenty of cases where published results turned out to be faulty [13, 4]. Such errors seem more likely in mathematical correctness proofs of algorithms, protocols etc. These tend to be more messy and intricate than (most) proofs in pure mathematics, and those performing the proofs are often not primarily trained as mathematicians. So while there are still some voices of dissent [6], there is a general consensus in the formal verification world that correctness proofs should be at least checked, and perhaps partly or wholly generated, by computer. In pure mathematics a similar opinion is still controversial, but we expect it to slowly percolate into the mathematical mainstream over the coming decades.

One obvious and common objection to computer-checked proofs is: why should we believe that they are any more reliable than human proofs? Well, for most practical purposes, computers can be considered mechanically infallible. Though 'soft errors' resulting from particle bombardment are increasingly significant as miniaturization advances, techniques for controlling these and other related effects are well-established and already in widespread use for high-integrity systems. The issue is not so much one of mechanical reliability, but rather the correctness of the proof-checking program itself, as well as potentially the stack of software it runs on. We may be willing to accept a machine-checked proof that we couldn't conceivably 'survey' ourselves, provided we understand and have confidence in the checking program — in this sense a proof checker provides intellectual leverage [16]. But how can we, or why should we? Who checks the checker?

2 LCF

Many practitioners consider worries about the fallibility of provers somewhat pointless. Experience shows unambiguously that typical mainstream proof checkers *are* far more reliable than human hand proofs, and abstract theorizing to the contrary is apt to look like empty chatter. Yet bugs in proof checkers are far from being unknown, and on at least one occasion, there was an announcement that an open problem had been solved by a theorem prover, later traced to a bug in the prover. For example, versions of HOL [9] have in the past had errors of two kinds:[1]

- Errors in the underlying logic, e.g. early versions allowed constant definitions with type variables occurring in the definiens but not the constant.
- Errors in the implementation, e.g. functions implementing logical operations were found not to rename variables to avoid free variable capture.

So what if we want to achieve the highest levels of confidence? We have no fully satisfactory answer to the thoroughgoing skeptic who doubts the integrity of the implementation language, compiler, operating system or hardware. But at least let us assume those are correct and consider how we might reassure ourselves about the proof checker itself, proving the absence of logical *or* implementation errors.

Since serious proof checkers are large and complex systems of software, their correctness is certainly open to doubt. However, there are established approaches to this problem. Some systems satisfy the *de Bruijn criterion* [2]: they can output a proof that is checkable by a much simpler program. Others based on the LCF approach [10] generate all theorems internally using a small logical kernel: only this is allowed to create objects of the special type 'theorem', just as only the kernel of an operating system is allowed to execute in privileged mode. From a certain point of view, one can say that an LCF prover satisfies the de Bruijn criterion, except that the proof exists only ephemerally and is checked by the kernel as it is created. And it is straightforward to instrument an LCF kernel so that it does actually output separately checkable proofs [22].

The original Edinburgh LCF system was designed to support proofs in a special 'Logic of Computable Functions' [19], hence the name LCF. But the key idea, as Gordon [8] emphasizes, is equally applicable to more orthodox logics supporting conventional mathematics, and subsequently many 'LCF-style' proof checkers have been designed using the same principles. In particular, the original

---

[1] In the absence of a highly rigorous abstract specification of the logic, it's not always easy to categorize errors in this way, but these examples seem clear.

HOL system [9] and its descendant HOL Light [11] are LCF-style provers. HOL Light is constructed on top of a logical kernel consisting of only around 400 lines of Objective CAML. Thus, if we accept that the interface to the trusted kernel is correct, we need only verify those 400 lines of code. In the present paper, we describe significant though imperfect progress towards this goal.

3 On self-verification

Tarski's theorem on the undefinability of truth tells us that no logical system (capable of formalizing a certain amount of arithmetic) can formalize its own semantics, and Gödel's second incompleteness theorem tells us that it cannot prove its own consistency in any way at all — unless of course it *isn't* consistent, in which case it can prove anything [21]. So, regardless of implementation details, if we want to prove the consistency of a proof checker, we need to use a logic that in at least some respects goes beyond the logic the checker itself supports.

The most obvious approach, therefore, would be to verify HOL Light using a system whose logic is at least strong enough to formalize HOL Light's semantics, e.g. Mizar [18] based on Tarski-Grothendieck set theory. Instead, simply on the grounds of personal expertise with it, we have chosen to verify HOL Light *in itself.* Of course, in the light of the above observations, we cannot expect to prove consistency of HOL in itself, $\vdash_{\texttt{HOL}} Con(\texttt{HOL})$. Instead, we have proven two similar results: consistency of HOL within a stronger variant of HOL, and of a weaker variant of HOL within ordinary HOL:[2]

- $I \vdash_{\texttt{HOL}} Con(\texttt{HOL})$ for a new axiom I about sets.
- $\vdash_{\texttt{HOL}} Con(\texttt{HOL} - \{\infty\})$ where $\texttt{HOL} - \{\infty\}$ is HOL with no axiom of infinity.

One can still take the view that these results are pointless, but they cover most of the problems we worry about. Almost all implementation bugs in HOL Light and other versions of HOL have involved variable renaming, and manifest themselves in a contradiction regardless of whether we assume the axiom of infinity. So having a correctness proof of something close to the actual *implementation* of $\texttt{HOL} - \{\infty\}$, rather than merely the abstract logic, is a real reassurance.

Naturally, it is possible that a soundness bug in HOL Light could mean that these correctness statements themselves are not true, but have only been 'proved' by means of this bug! There are two counterarguments. Intuitively, it seems unlikely that some logical or implementation bug, never spotted in any other domain, should just happen to manifest itself in the proof of consistency. And HOL Light is able to generate proof logs that can be checked in Isabelle/HOL, thanks to work by Steven Obua. Thus, having a proof in HOL Light, we effectively have a proof in Isabelle/HOL too, which implements a similar logic but is quite different in terms of internal organization and so unlikely to feature the same implementation bugs.

---

[2] Thanks to Rob Arthan for pointing out this kind of possibility.

4 HOL Light foundations and axioms

HOL Light's logic is simple type theory [3, 1] 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 $\sigma \to \tau$.[3]

For the core HOL logic, there is essentially only one predefined logical constant, equality $(=)$ with polymorphic type $\alpha \to \alpha \to \mathrm{bool}$. However to state one of the mathematical axioms we also include another constant $\varepsilon : (\alpha \to \mathrm{bool}) \to \alpha$, 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 [12].

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

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

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

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

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

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

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

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

---

[3] 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.

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

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

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. Our notation for term and type instantiation assumes capture-avoiding substitution, which we discuss in detail later.

All the usual logical constants are defined in terms of equality — see below for exactly what we mean by *defined*. 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 [5].
- The axiom of infinity `INFINITY_AX`, discussed further below.

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. However, in our verification we don't formalize the rule of definition, instead regarding the definitions of the connectives as 'meta-level' definitions. When we write, say, $\bot$, it is merely an abbreviation for the term $\forall p.\, p$ and so on. We took this path to avoid technical complications over the changing signature of the logic, but eventually we want to generalize our proof to cover the actual HOL definitional principles. Neither do we presently formalize the rule of type definition, though we would eventually like to do so.

5 Added and removed axioms

Since in our self-verifications we either remove the axiom of infinity '∞' or add a new axiom I, we will explain these carefully. The HOL Light axiom of infinity asserts that the type `ind` of individuals is Dedekind-infinite, i.e. that there is a function from ind to itself that is injective but not surjective:

```
|- ∃f:ind->ind. ONE_ONE f ∧ ¬ONTO f
```

where the subsidiary concepts are defined as follows:

```
|- ∀f. ONE_ONE f ⇔ (∀x1 x2. f x1 = f x2 ⟹ x1 = x2)
|- ∀f. ONTO f ⇔ (∀y. ∃x. y = f x)
```

This is the only rule or axiom that says anything specifically about `ind`. If we exclude it, we can find a model for the HOL logic where `ind` is modelled by (say) a 1-element set, and `bool` as usual by a 2-element set. Then any type we can construct from those using the function space constructor will also have an interpretation as a finite set. Thus we can find a model for the entire type hierarchy inside any infinite set, which we have in the full HOL logic.

But to model all of HOL including its axiom of infinity, we must take an infinite set, say $\mathbb{N}$, to model the type of individuals. We then need to be able to model at least the infinite hierarchy of types ind, ind → bool, (ind → bool) → bool and so on, so we need a set for the universe of types that can contain $\wp^n(\mathbb{N})$, the n-fold application of the power set operation to $\mathbb{N}$, for all $n \in \mathbb{N}$. Since for successive n these have cardinality $\aleph_0$, $2^{\aleph_0}$, $2^{2^{\aleph_0}}$ and so on, we cannot prove in HOL that there is any set large enough. So we add a new axiom I that gives us a 'larger' universe of sets. In the traditional terminology of cardinal arithmetic it asserts that there is a cardinal ι with the property that it is strictly larger than the cardinality of $\mathbb{N}$ and is closed under exponentiation applied to smaller cardinals: $\aleph_0 < \iota \wedge (\forall \kappa.\ \kappa < \iota \implies 2^\kappa < \iota)$. This is unproblematic in ZF set theory (e.g. take ι to be the cardinal of $V_{\omega+\omega}$ in the hierarchy of sets) so there is nothing dubious or recherché about our new axiom.

In order to deal with the two cases of removing and adding an axiom almost entirely uniformly, we start each proof by defining a type `ind_model` to model the type `ind`, as well as the type `I` that will model the whole type universe. In proving $I \vdash Con(HOL)$, we introduce such types and assert our higher axiom of infinity for them:

```
|- (:ind_model) <_c (:I) ∧
   (∀s:A->bool. s <_c (:I) ⟹ {t | t SUBSET s} <_c (:I)
```

Here '`(:I)`' is a HOL Light shorthand for the universal set on type `I`, and '$<_c$' is strict cardinal comparison, defined as the irreflexive form of non-strict cardinal comparison, itself defined in terms of the existence of an injective map from one set to the other.[4]

In the case of proving $\vdash_{\texttt{HOL}} Con(\texttt{HOL} - \{\infty\})$, we just *define* a type `ind_model` in bijection with a finite nonempty set and `I` in bijection with $\mathbb{N}$. In this case we can easily *prove* the statement about cardinal closure, instead of taking it as an axiom. The subsequent proofs are all completely identical based on this cardinality property, except that right at the end we need in one case to show how we can model the axiom of infinity.

6 Formalized syntax

The various OCaml types representing logical entities of types and terms are formalized inside the HOL logic using analogous recursive type definitions. However, there is an important difference, which we will explain for types first. In the code, the type of HOL types is declared by the following OCaml recursive type definition:

```
type hol_type = Tyvar of string
              | Tyapp of string *  hol_type list
```

The second clause allows a type constructor with any name and arity. However, the constructors themselves are hidden by an abstract type interface, which permits only types using type constructors that have been declared. In the initial state these amount to just the base types `bool`, `ind` and the binary function space constructor `fun`, but later type definitions can extend the list. In the HOL formalization, we do not consider the potentially extensible type signature, and just 'hardwire' the base types we will consider:

```
define_type "type = Tyvar string
                  | Bool
                  | Ind
                  | Fun type type";;
```

Similarly, the basic type of HOL terms is defined in OCaml without any well-typedness restriction, with any term as the "bound variable" of a lambda-abstraction, with these restrictions imposed by the abstract type interface.

```
type term = Var of string * hol_type
          | Const of string * hol_type
          | Comb of term * term
          | Abs of term * term
```

---

[4] In simple type theory, it is problematic defining a general type of cardinals, but many arguments can be rephrased in terms of cardinal comparison and set operations [7].

In the HOL formalization, we wire in the two primitive constants, where ‘Equal α’ represents $(=) : \alpha \to \alpha \to \text{bool}$ and ‘Select α’ represents $(\varepsilon) : (\alpha \to \text{bool}) \to \alpha$, and we syntactically force the bound variable of a lambda-abstraction to be a (typed) variable and not any other kind of term:

```
define_type "term = Var string type
                  | Equal type | Select type
                  | Comb term term
                  | Abs string type term";;
```

This allows ill-typed terms that could not be constructed using the abstract type interface of HOL Light, so we often need to state side-conditions connected with well-typedness on our theorems. This notion is defined as

```
 |- welltyped tm ⇔ ∃ty. tm has_type ty
```

where the typing judgement, written infix, is defined inductively as follows; every welltyped term then has a unique type, extracted by a function `typeof`.

```
 |- (∀n ty. (Var n ty) has_type ty) ∧
    (∀ty. (Equal ty) has_type (Fun ty (Fun ty Bool))) ∧
    (∀ty. (Select ty) has_type (Fun (Fun ty Bool) ty)) ∧
    (∀s t dty rty. s has_type (Fun dty rty) ∧ t has_type dty
                   ⟹ (Comb s t) has_type rty) ∧
    (∀n dty t rty. t has_type rty ⟹ (Abs n dty t) has_type (Fun dty rty))`;;
```

Subject to these systematic differences, we model much of the OCaml code in the core faithfully. Most syntax functions are purely functional, and we “naively” transcribe them into corresponding definitional theorems in the logic, following [20]. In general, recursive functions in OCaml may fail to terminate, and this aspect is not adequately modelled by our encoding.[5] In practice all the functions we use do terminate, and without some inductive argument we would not be able to prove anything non-trivial about them. So this distinction is somewhat academic, and generally speaking the structural similarity is very clear. (It’s particularly important to emphasize this point, since most of our discussion here is devoted to differences.) For example, the function that performs a union of term lists modulo alpha-equivalence in OCaml is:

```
let rec term_union l1 l2 =
  match l1 with
    [] -> l2
  | (h::t) -> let subun = term_union t l2 in
              if exists (aconv h) subun then subun else h::subun;;
```

and the HOL formalization is:

```
 |- (TERM_UNION [] l2 = l2) ∧
    (TERM_UNION (CONS h t) l2 =
          let subun = TERM_UNION t l2 in
          if EX (ACONV h) subun then subun else CONS h subun)
```

---

[5] HOL Light’s derived rules can prove consistency of various recursive definitions, in particular all tail-recursive ones (I owe the observation that these are always consistent to J Moore). This does *not* imply termination of the analogous functional program.

At the other end of the spectrum, the worst case for the correspondence between code and HOL formalization is the type instantiation function, which replaces type variables $\alpha_1, \ldots, \alpha_n$ with other types $\sigma_1, \ldots, \sigma_n$ in some term. The OCaml code involves exceptions and pointer-equality tests:

```
let rec inst env tyin tm =
  match tm with
    Var(n,ty)   -> let ty' = type_subst tyin ty in
                   let tm' = if ty' == ty then tm else Var(n,ty') in
                   if rev_assocd tm' env tm = tm then tm'
                   else raise (Clash tm')
  | Const(c,ty) -> let ty' = type_subst tyin ty in
                   if ty' == ty then tm else Const(c,ty')
  | Comb(f,x)   -> let f' = inst env tyin f and x' = inst env tyin x in
                   if f' == f & x' == x then tm else Comb(f',x')
  | Abs(y,t)    -> let y' = inst [] tyin y in
                   let env' = (y,y')::env in
                   try let t' = inst env' tyin t in
                       if y' == y & t' == t then tm else Abs(y',t')
                   with (Clash(w') as ex) ->
                   if w' <> y' then raise ex else
                   let ifrees = map (inst [] tyin) (frees t) in
                   let y'' = variant ifrees y' in
                   let z = Var(fst(dest_var y''),snd(dest_var y)) in
                   inst env tyin (Abs(z,vsubst[z,y] t))
```

The `tyin` argument is an association list $[\sigma_1, \alpha_1; \cdots; \sigma_n, \alpha_n]$ specifying the desired instantiation, `tm` is the term to instantiate, and `env` is used to keep track of correspondences between original and instantiated variables to detect name clash problems. Note first that the recursive cases are optimized to avoid rebuilding the same term. For example, the case for `Comb(f,x)` checks if the instantiated subterms `f'` and `x'` are pointer identical (‘`==`’) to the originals, and if so just returns the full original term. This optimization is not, and cannot be, reflected in our naive model.

The main complexity in this function is detecting and handling variable capture. For example, the instantiation of α to bool in the constant function $\lambda x : \text{bool}.\ x : \alpha$ would, if done naively, result in the identity function $\lambda x : \text{bool}.\ x : \text{bool}$. We want to ensure instead that we get something like $\lambda x' : \text{bool}.\ x : \text{bool}$. So each time a variable is type-instantiated (first clause) we check that it is consistent with the list `env`, which roughly means that if after instantiation it is bound by some abstraction, it was already bound by the same one before. If this property fails, an exception `Clash` is raised with the problem term. This exception is supposed to be caught by exactly the outer recursive call for that abstraction, which renames the variable appropriately and tries again (last line).

Exceptions also have no meaning in our naive model. Instead, we include the possibility of exceptions by extending the return type of the function in our HOL formalization to a disjoint sum type defined by:

```
define_type "result = Clash term | Result term";;
```

In all expressions we manually ‘propagate’ the `Clash` exception with the help of discriminator (`IS_RESULT` and `IS_CLASH`) and extractor (`RESULT` and `CLASH`) functions. These are also used at the end to take us back to a simple analog `INST` of the OCaml code's main `inst` function.[6] With all these caveats, the overall structure should faithfully model the OCaml code:

```
|- (INST_CORE env tyin (Var x ty) =
      let tm = Var x ty
      and tm’ = Var x (TYPE_SUBST tyin ty) in
     if REV_ASSOCD tm’ env tm = tm then Result tm’ else Clash tm’) ∧
   (INST_CORE env tyin (Equal ty) = Result(Equal(TYPE_SUBST tyin ty))) ∧
   (INST_CORE env tyin (Select ty) = Result(Select(TYPE_SUBST tyin ty))) ∧
   (INST_CORE env tyin (Comb s t) =
        let sres = INST_CORE env tyin s in
        if IS_CLASH sres then sres else
        let tres = INST_CORE env tyin t in
        if IS_CLASH tres then tres else
        let s’ = RESULT sres and t’ = RESULT tres in
        Result (Comb s’ t’)) ∧
   (INST_CORE env tyin (Abs x ty t) =
        let ty’ = TYPE_SUBST tyin ty in
        let env’ = CONS (Var x ty,Var x ty’) env in
        let tres = INST_CORE env’ tyin t in
        if IS_RESULT tres then Result(Abs x ty’ (RESULT tres)) else
        let w = CLASH tres in
        if ¬(w = Var x ty’) then tres else
        let x’ = VARIANT (RESULT(INST_CORE [] tyin t)) x ty’ in
        INST_CORE env tyin (Abs x’ ty (VSUBST [Var x’ ty,Var x ty] t)))
```

The termination of this function needs a careful argument. The last line can result in a recursive call on a term of the same size, but the choice of new variable means that the subcall will then not raise the same exception that would lead to yet another subcall from this level.

We now introduce a handy abbreviation for equations (an exact counterpart to a function `mk_eq` in the OCaml code):

```
|- (s === t) = Comb (Comb (Equal(typeof s)) s) t
```

and are ready to model the HOL Light deductive system using an inductively defined 'is provable' predicate '`|-`'. For reasons of space, we only show clauses for rules `REFL`, `TRANS` and `INST_TYPE`, but none of them are complex or surprising:

```
|- (∀t. welltyped t ⟹ [] |- t === t) ∧
   (∀asl1 asl2 l m1 m2 r.
         asl1 |- l === m1 ∧ asl2 |- m2 === r ∧ ACONV m1 m2
         ⟹ TERM_UNION asl1 asl2 |- l === r) ∧
   ...
   (∀tyin asl p. asl |- p ⟹ MAP (INST tyin) asl |- INST tyin p) ∧
   ...
```

7 Set theory

We next develop a HOL type `V` of 'sets' big enough to model all types. The sets are arranged in levels somewhat analogous to the Zermelo-Fraenkel hierarchy,

---

[6] The main recursion for `inst` shown above is used internally in the definition of the main `inst`, which simply provides the empty list as the initial `env` argument. The choice of names is a bit confusing: this is used in the inference rule `INST_TYPE`; the rule `INST` is term instantiation and the corresponding term operation is called `VSUBST`.

each containing all subsets of the levels below it. The membership symbol in `V` is written as an infix `<:`, and has type $V \to V \to \mathrm{bool}$. (This is quite distinct from the usual HOL set/predicate membership operation `IN` with type $\alpha \to (\alpha \to \mathrm{bool}) \to \alpha$.) Many of the axioms and constructs familiar from ZF set theory appear, e.g 's suchthat p' is the subset of elements of s satisfying p, whose existence is assured by the ZF separation axiom:

```
|- (level(s suchthat p) = level s) ∧ ∀x. x <: s suchthat p ⇔ x <: s ∧ p x
```

Similarly we have a choice function `ch` satisfying:

```
|- ∀s. (∃x. x <: s) ⟹ ch(s) <: s
```

But we have no need of 'mixed level' sets like $\{\emptyset, \{\emptyset\}\}$, so we make the hierarchy non-cumulative, with the levels all distinct. This means that there are multiple empty sets at all levels of the hierarchy, so we don't have simple extensionality. We also have a primitive notion of pairing (it is not defined as is usually done in ZF set theory), and we start with two basic sets of 'ur-elements' `boolset` (with elements `true` and `false`) and `indset` to model the base HOL types. We will not show the technical details of the construction, since they are not particularly interesting or challenging. We just observe some notation for later use.

The set of functions from set `s` to set `t` (constructed much as in ZF set theory, as a certain set of ordered pairs) is denoted by `funspace s t`, and function application is `apply`. We also define a set-theoretic analog `abstract` of lambda-abstraction to allow us to construct certain functions explicitly. Here are a few relevant lemmas to help the reader to get a picture of the setup.

```
|- x <: s ∧ f(x) <: t ⟹ (apply(abstract s t f) x = f(x))

|- x <: s ∧ f <: funspace s t ⟹ apply f x <: t

|- (∀x. x <: s ⟹ f(x) <: t) ⟹ abstract s t f <: funspace s t
```

Note that everything in the construction of this set-theoretic hierarchy is based on the key cardinality property we noted earlier; no other axioms are used. Of course, this property was designed exactly to allow the construction of such a type.

8 Formalized semantics

HOL is a fairly simple logic, and it isn't so difficult to give it a set-theoretic semantics. However, the presence of polymorphic type variables makes it a bit trickier than it first appears. Our approach is inspired by the semantics given by Andy Pitts [9], though we use more traditional valuation-based formulation rather than using contexts, since it seems (to us) technically simpler. The semantics is parameterized throughout by a valuation τ : `string` $\to$ V of the type variables. We require only that it always returns a nonempty set:

```
|- type_valuation tau ⇔ ∀x. (∃y. y <: tau x)
```

Given such a type valuation, each HOL type is allocated a corresponding set in `V` using the following straightforward definition:

```
|- (typeset tau (Tyvar s) = tau(s)) ∧
   (typeset tau Bool = boolset) ∧
   (typeset tau Ind = indset) ∧
   (typeset tau (Fun a b) = funspace (typeset tau a) (typeset tau b))
```

Now we come to the semantics of terms. As well as the valuation τ of type variables, this has as another parameter a valuation σ of term variables, or more precisely of name-type pairs. This should always be consistent with τ, i.e. should map each variable-type pair into the set corresponding to that type:

```
|- term_valuation tau sigma ⇔ ∀n ty. sigma(n,ty) <: typeset tau ty
```

The definition of the semantics is:

```
|- (semantics sigma tau (Var n ty) = sigma(n,ty)) ∧
   (semantics sigma tau (Equal ty) =
        abstract (typeset tau ty) (typeset tau (Fun ty Bool))
                 ((λx. abstract (typeset tau ty) (typeset tau Bool)
                                (λy. boolean(x = y)))) ∧
   (semantics sigma tau (Select ty) =
        abstract (typeset tau (Fun ty Bool)) (typeset tau ty)
                 (λs. if ∃x. x <: ((typeset tau ty) suchthat (holds s))
                      then ch ((typeset tau ty) suchthat (holds s))
                      else ch (typeset tau ty))) ∧
   (semantics sigma tau (Comb s t) =
        apply (semantics sigma tau s) (semantics sigma tau t)) ∧
   (semantics sigma tau (Abs n ty t) =
        abstract (typeset tau ty) (typeset tau (typeof t))
                 (λx. semantics (((n,ty) |-> x) sigma) tau t))
```

The first clause is easy: just apply the valuation σ. The fourth and fifth clauses are also fairly natural: they just map application and abstraction into their set-theoretic counterparts. The semantics of a term $\lambda n : ty.\ t$ is a function taking an argument x that recursively evaluates the semantics of t in a modified valuation with $n : ty$ mapped to x but which is otherwise the same as σ. (The modification is done by an infix function update '`|->`'.) The second and third clauses look involved only because we actually need to interpret $=$ and ε as functions of the appropriate type, but they just give the right sets for the obvious equality and choice functions. (If it would otherwise be applied to an empty set, we force the choice operator to pick any element of the right type.) When the equality constant is actually used in an equation in the usual way, the semantics, with reasonable side-conditions, is about what we would expect. Note that here and above `boolean` just maps a HOL boolean into the corresponding member of `boolset` in `V`:[7]

---

[7] The typing condition is a shorthand for saying that both subterms are welltyped; an equation always has boolean type if so.

```
|- (s === t) has_type Bool ∧ type_valuation tau ∧ term_valuation tau sigma
   ⟹ (semantics sigma tau (s === t) =
        boolean(semantics sigma tau s = semantics sigma tau t))
```

We proceed with various lemmas about how the semantics interacts with syntactic operations. The most complex governs the type instantiation operation whose definition we considered earlier:

```
|- ∀tyin tm sigma tau.
        welltyped tm ∧ type_valuation tau ∧ term_valuation tau sigma
        ⟹ (semantics sigma tau (INST tyin tm) =
             semantics
                (λ(x,ty). sigma(x,TYPE_SUBST tyin ty))
                (λs. typeset tau (TYPE_SUBST tyin (Tyvar s))) tm)
```

where `TYPE_SUBST` is substitution of types for type variables within a *type*, defined by straightforward recursion. Finally, we define the semantic notion of entailment:

```
|- asms |= p ⇔ ALL (λa. a has_type Bool) (CONS p asms) ∧
                 ∀sigma tau. type_valuation tau ∧ term_valuation tau sigma ∧
                             ALL (λa. semantics sigma tau a = true) asms
                             ⟹ (semantics sigma tau p = true)
```

and hence by induction, considering the various inference rules, we deduce that HOL is sound:

```
|- ∀asl p. asl |- p ⟹ asl |= p
```

and consistent in the sense that there is an unprovable formula:

```
|- ∃p. p has_type Bool ∧ ¬([] |- p)
```

9 Conclusions and related work

We believe that this is the first time anything close to the implementation of a ‘real’ theorem prover has been verified against a semantic model, though syntactic features of the HOL logic have been formalized before [23], and full correctness for a first-order proof checker [14] and a simple first-order tableau prover [17] have been verified. We believe that a proof based on a semantics is more valuable than one relative to an abstract description of the same deductive system: even the abstract definitions of notions like capture-avoiding substitution are somewhat involved, and it is much more satisfactory to characterize them by their (relatively) simple semantics — cf. the key theorem about the semantics of `INST` above. On the other hand, it might be a fruitful separation of concerns to use a more abstract description of the logic as an intermediate step between the implementation and the semantics.

As for practical consequences, we are genuinely pleased to have finally convinced ourselves that the variable renaming methods (notably the rather involved mechanism in type instantiation) are correct. This is where our practical worries lay. Still, we view the present work only as a proof of concept: we have shown that all the key things can be made to work as we would wish; there is not much intellectual work involved in taking it further. But there are many obvious shortcomings in the work so far that need to be addressed. First of all, we need to properly model the full extensible signatures and the definitional principles that extend them. It would also be desirable to use a more detailed model of the implementation language. Our present work certainly does little to guard against language issues. In fact there is one that we know about: OCaml's strings are mutable, and this leads to imperfect protection of the abstract types of types and terms.

Another avenue for future work would be to extend the semantics to cover extensions to the logic, such as the introduction of quantifiers over type variables suggested in [15]. Tom Ridge has already updated large parts of HOL Light to incorporate them, and we believe the extension of the semantics is straightforward.

Acknowledgements

I would like to thank Rob Arthan, who first pointed out the feasibility of this kind of self-verification, and John Matthews and Tom Ridge who suggested extending this work to cover quantified type variables. Members of WG 2.3 provided some valuable feedback on an earlier version of this work. The anonymous referees, who evidently read the submitted version with great care, provided helpful remarks and caught several errors.

References

1. P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof.* Academic Press, 1986.
2. H. Barendregt. The impact of the lambda calculus on logic and computer science. *Bulletin of Symbolic Logic*, 3:181–215, 1997.
3. A. Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.
4. P. J. Davis. Fidelity in mathematical discourse: Is one and one really two? *The American Mathematical Monthly*, 79:252–263, 1972.
5. R. Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51:176–178, 1975.
6. E. W. Dijkstra. Formal techniques and sizeable programs (EWD563). In E. W. Dijkstra, editor, *Selected Writings on Computing: A Personal Perspective*, pages 205–214. Springer-Verlag, 1976. Paper prepared for Symposium on the Mathematical Foundations of Computing Science, Gdansk 1976.
7. T. Forster. *Reasoning about theoretical entities*, volume 3 of *Advances in Logic.* World Scientific, 2003.
8. M. J. C. Gordon. Representing a logic in the LCF metalanguage. In D. Néel, editor, *Tools and notions for program construction: an advanced course*, pages 163–185. Cambridge University Press, 1982.

9. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
10. 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.
11. J. Harrison. HOL Light: A tutorial introduction. In M. Srivas and A. 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.
12. J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1986.
13. M. Lecat. *Erreurs de Mathématiciens*. Brussels, 1935.
14. W. McCune and O. Shumsky. Ivy: A preprocessor and proof checker for first-order logic. In M. Kaufmann, P. Manolios, and J. S. Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, pages 265–281. Kluwer, 2000.
15. T. F. Melham. The HOL logic extended with quantification over type variables. In L. J. M. Claesen and M. J. C. Gordon, editors, *Proceedings of the IFIP TC10/WG10.2 International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume A-20 of *IFIP Transactions A: Computer Science and Technology*, pages 3–18, IMEC, Leuven, Belgium, 1992. North-Holland.
16. R. Pollack. How to believe a machine-checked proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, 1998. Also available on the Web as `http://www.brics.dk/~pollack/export/believing.ps.gz`.
17. T. Ridge. A mechanically verified, efficient, sound and complete theorem prover for first order logic. Available via `http://homepages.inf.ed.ac.uk/s0128214/`, 2005.
18. P. Rudnicki. An overview of the MIZAR project. Available on the Web as `http://web.cs.ualberta.ca/~piotr/Mizar/MizarOverview.ps`, 1992.
19. D. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Annotated version of a 1969 manuscript.
20. K. Slind. *Reasoning about terminating functional programs*. PhD thesis, Institut für Informatik, Technische Universität München, 1999. Available from `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/1999/slind.html`.
21. R. M. Smullyan. *Gödel's Incompleteness Theorems*, volume 19 of *Oxford Logic Guides*. Oxford University Press, 1992.
22. W. Wong. Recording HOL proofs. Technical Report 306, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK, 1993.
23. J. von Wright. Representing higher-order logic proofs in HOL. In T. F. Melham and J. Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 7th International Workshop*, volume 859 of *Lecture Notes in Computer Science*, pages 456–470, Valletta, Malta, 1994. Springer-Verlag.