HOL with Definitions: Semantics, Soundness, and a Verified Implementation

Ramana Kumar[1], Rob Arthan[2], Magnus O. Myreen[1], and Scott Owens[3]

[1] Computer Laboratory, University of Cambridge, UK
[2] School of EECS, Queen Mary, University of London, UK
[3] School of Computing, University of Kent, UK

Abstract. We present a mechanised semantics and soundness proof for the HOL Light kernel including its definitional principles, extending Harrison's verification of the kernel without definitions. Soundness of the logic extends to soundness of a theorem prover, because we also show that a synthesised implementation of the kernel in CakeML refines the inference system. Our semantics is the first for Wiedijk's stateless HOL; our implementation, however, is stateful: we give semantics to the stateful inference system by translation to the stateless. We improve on Harrison's approach by making our model of HOL parametric on the universe of sets. Finally, we prove soundness for an improved principle of constant specification, in the hope of encouraging its adoption. This paper represents the logical kernel aspect of our work on verified HOL implementations; the production of a verified machine-code implementation of the whole system with the kernel as a module will appear separately.

1 Introduction

In this paper, we present a mechanised proof of the soundness of higher-order logic (HOL), including its principles for defining new types and new polymorphic constants, and describe production of a verified implementation of its inference rules. This work is part of a larger project, introduced in our rough diamond last year [11], to produce a verified machine-code implementation of a HOL prover. This paper represents the top half of the project: soundness of the logic, and a verified implementation of the logical kernel in CakeML [7].

What is the point of verifying a theorem prover and formalising the semantics of the logic it implements? One answer is that it raises our confidence in the correctness of the prover. A prover implementation usually sits at the centre of the trusted code base for verification work, so effort spent verifying the prover multiplies outwards. Secondly, it helps us understand our systems (logical and software), to the level of precision possible only via formalisation. Finally, a theorem prover is a non-trivial piece of software that admits a high-level specification and whose correctness is important: we see it as a catalyst for tools and methods aimed at developing complete verified systems, readying them for larger systems with less obvious specifications.

The first soundness proof we present here is for Wiedijk's stateless HOL [16], in which terms carry their definitions; by formalising we hope to clarify the semantics of this system. We then show that traditional stateful HOL, where terms are understood in a context of definitions, is sound by a translation to the stateless inference system.

We build on Harrison's proof of the consistency of HOL without definitions [4], which shares our larger goal of verifying concrete HOL prover implementations, and advance this project by verifying an implementation of the HOL Light [5] kernel in CakeML, an ML designed to support fully verified applications. We discuss the merits of Harrison's model of set theory defined within HOL, and provide an alternative not requiring axiomatic extensions.

Our constant specification rule generalises the one found in the various HOL systems, adding support for implicit definitions with fewer constraints and no new primitives. We lack space here to justify its design in full detail, but refer to a proposal [2] by the second author. We hope our proof of its soundness will encourage its adoption.

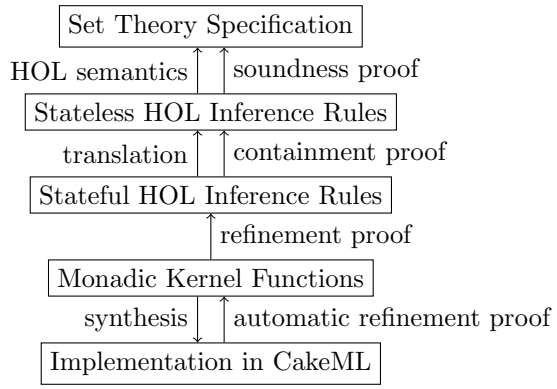
The specific contributions of this paper are:

- a formal semantics for Wiedijk's stateless HOL (§4), against a new specification of set theory (§3),
- a proof of soundness (§4.2) for stateless HOL, including type definitions, a new rule for constant specification, and the three axioms used in HOL Light,
- a proof of soundness for stateful HOL by translation to stateless (§5), and
- a verified implementation of the HOL Light kernel in CakeML (§6) that should be a suitable basis for a verified implementation of the prover in machine-code.

All our definitions and proofs have been formalised in the HOL4 theorem prover [14] and are available from `https://cakeml.org`.[1]

2 Approach

At a high level, our semantics and verified implementation fit together as follows.



[1] Specifically, the `hol-light` directory of `https://github.com/xrchz/vml`.

The overall theorems we obtain are about evaluating the CakeML implementations of the HOL Light kernel functions in CakeML's operational semantics. For each kernel function, we prove that if the function is run in a good state on good arguments, it terminates in a good state and produces good results. Here "good" refers to our refinement invariants. In particular, a good value of type "thm" must refine a sequent in stateful HOL that translates to a sequent in stateless HOL that is valid according to the set-theoretic semantics.

We prove these results by composing the four proof layers in the diagram. Starting from the top, the HOL semantics interprets stateless HOL sequents in set theory, from which we obtain a definition of validity. The soundness proof says that each of the stateless HOL inference rules preserves validity of sequents.

In stateless HOL, defined types and terms carry syntactic tags describing their definitions, whereas in stateful HOL there is a context of definitions that is updated when a new definition is made. Our translation from stateful to stateless takes the definitions from the context and inlines them into the tags. Our containment proof then shows that whenever the stateful system proves a sequent, the stateless system proves the translation of the sequent.

As outlined in our rough diamond [11], we define shallowly-embedded HOL functions, using a state-exception monad, for each of the HOL Light kernel functions. These "monadic kernel functions" are written following the original OCaml code closely, then we prove that they implement the stateful inference rules. Specifically, if one of these functions is applied to good arguments, it terminates with a good result; any theorem result must refine a sequent that is provable in the stateful system.

Finally, using the method developed by Myreen and Owens [10] we synthesise CakeML implementations of the monadic kernel functions. This automatic translation from shallowly- to deeply-embedded code is proof-producing, and we use the certificate theorems to complete the refinement proof.

In the context of our larger project, the next steps include: a) proving, against CakeML's semantics, that our implementation of the kernel can be wrapped in a module to protect the key property, provability, of values of type "thm"; and b) using CakeML's verified compiler to generate a machine-code implementation of the kernel embedded in an interactive read-eval-print loop that is verified to never print a false theorem.

3 Set Theory

A rigorous but informal account of the semantics of HOL, due to Pitts, is given in the HOL4 documentation [12]. It assigns meanings to HOL constructs in a universe of sets satisfying Zermelo's axioms. We wish to do the same with a semantics developed using conservative extensions in HOL. Gödel's second incompleteness theorem implies that we cannot actually define a model of Zermelo set theory. However, we can define what properties such a model must have and for us this is sufficient. It is convenient to separate out the axioms of choice

and infinity. A specification along these lines was developed previously by one of us [1] but without any formal proofs. We begin by defining a predicate

```
is_set_theory (mem :𝒰 -> 𝒰 -> bool)
```

that says whether a membership relation defined on some universe $\mathcal{U}$ (represented by a type variable) satisfies the Zermelo axioms other than choice and infinity, namely the axioms of extensionality, separation (a.k.a. comprehension or specification), power set, union, and pairing. As we are working in HOL, we can use propositional functions in place of the metavariables required in a first-order presentation:

Definition 1 (Specification of Set Theory Axioms).

```
is_set_theory mem ⟺
  extensional mem ∧ (∃sub. is_separation mem sub) ∧
  (∃power. is_power mem power) ∧ (∃union. is_union mem union) ∧
  ∃upair. is_upair mem upair
extensional mem ⟺
  ∀x y. x = y ⟺ ∀a. mem a x ⟺ mem a y
is_separation mem sub ⟺
  ∀x P a. mem a (sub x P) ⟺ mem a x ∧ P a
is_power mem power ⟺
  ∀x a. mem a (power x) ⟺ ∀b. mem b a ⇒ mem b x
is_union mem union ⟺
  ∀x a. mem a (union x) ⟺ ∃b. mem a b ∧ mem b x
is_upair mem upair ⟺
  ∀x y a. mem a (upair x y) ⟺ a = x ∨ a = y
```

A relation *mem* satisfying the above axioms is sufficient to define the semantics of HOL without Hilbert choice or the axiom of infinity, that is, for the (polymorphic) simply typed λ-calculus with equality. For the remaining features of HOL, we need two more parameters: a choice function, and a distinguished infinite set for the individuals. We specify a complete model as follows.[2]

Definition 2 (Specification of a Model for HOL).

```
is_model (mem,indset,ch) ⟺
  is_set_theory mem ∧ is_infinite mem indset ∧ is_choice mem ch
is_choice mem ch ⟺ ∀x. (∃a. mem a x) ⇒ mem (ch x) x
is_infinite mem s ⟺ infinite {a | mem a s}
```

3.1 Derived Concepts

In order to reuse Harrison's proofs [4] as much as possible, we define various constructions above our set theory model and prove the same theorems he did to characterise them. These theorems form the interface to set theory above which one can give a semantics to HOL. To save space, we do not list them all.

[2] `infinite (p :α -> bool)` abbreviates ¬`FINITE` p, with finiteness defined inductively for sets-as-predicates in HOL4's library.

For function spaces, function application, and abstraction of a HOL function, we use the standard set-theoretic method of identifying functions with their graphs. For Booleans we define a distinguished set of two elements and name its members. We often use abbreviations to hide the *mem* argument to a function, for example `funspace` s t below actually abbreviates `funspace0` *mem* s t.

$$\begin{array}{l}\vdash \texttt{is\_set\_theory}\ \mathit{mem} \Rightarrow \\ \quad (\forall f\ s\ t. \\ \quad\quad ((\forall x.\ \mathit{mem}\ x\ s \Rightarrow \mathit{mem}\ (f\ x)\ t) \Rightarrow \\ \quad\quad\quad \mathit{mem}\ (\texttt{abstract}\ s\ t\ f)\ (\texttt{funspace}\ s\ t)) \wedge \\ \quad\quad \forall x.\ \mathit{mem}\ x\ s \wedge \mathit{mem}\ (f\ x)\ t \Rightarrow \texttt{apply (abstract}\ s\ t\ f)\ x = f\ x) \wedge \\ \quad \forall x.\ \mathit{mem}\ x\ \texttt{boolset} \iff x = \texttt{true} \vee x = \texttt{false}\end{array}$$

3.2 Consistency

We wish to know that `is_model` is not an empty predicate, to protect against simple mistakes in the definition, and because the existence of a model will be an assumption on our soundness theorems. Since actually building a model in HOL would allow HOL to prove its own consistency, we will have to settle for something less. However, we wish to avoid axiomatic extensions if possible.

We tried using Harrison's construction [4] as witness, but unfortunately it uses what amounts to a type system to define a coherent notion of membership in terms of injections into a universe. (Harrison calls the types "levels".) For simplicity and familiarity our `is_set_theory` characterises an untyped set theory. In particular, we need extensionality to hold for all sets, while in Harrison's model empty sets of distinct types are distinct.

So instead we use a standard encoding of the hereditarily finite sets in HOL as natural numbers, which takes the universe to be the natural numbers, and *mem* n m to hold if the mth bit in the binary numeral for n is 1. With this model, we can introduce, by conservative extension, a universe type that satisfies `is_set_theory`, and, under the assumption that it contains an infinite set, that it satisfies `is_model` too. To be able to consistently assume the existence of infinite sets, the universe type has a free type variable.

Specifically, we define the universe as an arbitrary subset of α `+ num` for which a suitable membership relation exists. We prove the existence of such a subset, namely all the numbers in the right of the sum, by using the standard encoding, which it is straightforward to show satisfies the set-theoretic axioms. Thus, we prove the following:

$$\begin{array}{l}\vdash \exists (P : \alpha\ \texttt{+ num -> bool})\ (\mathit{mem} : \alpha\ \texttt{+ num -> } \alpha\ \texttt{+ num -> bool}). \\ \quad \texttt{is\_set\_theory\_pred}\ P\ \mathit{mem}\end{array}$$

where `is_set_theory_pred` P is like `is_set_theory` but with all quantification relativised to P. We then feed this theorem into HOL4's constant and type specification machinery to derive a new type α `V` for our universe, and its membership relation `V_mem`. Our main lemma follows directly:

$$\vdash \texttt{is\_set\_theory V\_mem}$$

There is a natural choice function for non-empty sets in the standard encoding of finite sets, namely, the most significant bit in the binary numeral. But there are no infinite sets, so, as we would expect from Gödel's second incompleteness theorem at this point, no model for the set of individuals.

Now we use the facts that our specification of `(V_mem :`α `V -> `α `V -> bool)` is loose—the only things that can be proved about it come from `is_set_theory` and not the specific construction of the model—and the type α `V`, being parametric, has no provable cardinality bound. Hence if τ is an unspecified type, it is consistent to assume that τ `V` includes infinite sets. We specify `V_indset` as an arbitrary infinite set under the assumption that one exists. We can then prove our desired theorem:

Theorem 1 (Example Model).

```
⊢ (∃I. is_infinite V_mem I) ⇒
    is_model (V_mem,V_indset,V_choice)
```

An alternative to the general `is_model` characterisation of a suitable set-theoretic model is to define a particular universe of sets and then prove that it has all the desired properties. This is the approach taken by Harrison [4], who constructs by conservative extension a monomorphic type `V` equipped with a membership relation satisfying a typed analogue of our `is_set_theory`. `V` is countably infinite and it would be inconsistent to assert that it is a model of the axiom of infinity. Harrison observes that one could adapt his formalisation to give a model of the axiom of infinity using a non-conservative extension. Our approach allows us to work by conservative extension while remaining consistent with an assumption of the axiom of infinity.

4 Stateless HOL

Traditional implementations of HOL are stateful because they support the definition of new type and term constants by updating a context. Wiedijk [16] showed that this is not necessary if defined constants carry their definitions with them. Since there is no state, it was an appealing target for extension of Harrison's definitionless semantics [4].

4.1 Inference System

The distinguishing feature of stateless HOL syntax is the presence of tags, `const_tag` and `type_op`, on constants and types, containing information about how they were defined or whether they are primitive. Because of these tags, the datatypes for terms and types are mutually recursive.

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

```
type = Tyvar of string | Tyapp of type_op * type list
type_op = Typrim of string * num | Tydefn of string * term
const_tag = Prim
          | Defn of num * (string × term) list * term
          | Tyabs of string * term
          | Tyrep of string * term
```

With the `Typrim` *name arity* and `Prim` tags we can build up HOL's primitive type operators and constants without baking them into the syntax, as the following abbreviations show.

`Bool`	for	`Tyapp (Typrim "bool" 0) []`
`Ind`	for	`Tyapp (Typrim "ind" 0) []`
`Fun` *x y*	for	`Tyapp (Typrim "fun" 2) [`*x*`;` *y*`]`
`Equal` *ty*	for	`Const "=" (Fun` *ty* `(Fun` *ty* `Bool)) Prim`
`Select` *ty*	for	`Const "@" (Fun (Fun` *ty* `Bool)` *ty*`) Prim`
s `===` *t*	for	`Comb (Comb (Equal (typeof` *s*`))` *s*`)` *t*

We will explain the tags for non-primitives when we describe the definitional rules, after introducing the inference system. We use similar notation to Harrison [4] wherever possible, for example we define well-typed terms and prove ⊢ `welltyped` *tm* ⟺ *tm* `has_type (typeof` *tm*`)`, and we define the following concepts: `closed` *tm*, indicating that *tm* has no free variables; `tvars` *tm* and `tyvars` *ty* collecting the type variables appearing in a term or type, and `tyinst` *tyin ty* instantiating type variables in a type.

We were able to reuse most of Harrison's stateful HOL rules for the stateless HOL inference system, defining provable sequents[3] inductively with a few systematic modifications. Changes were required to handle the fact that stateless syntax permits terms whose definitions are unsound because their tags do not meet the side-conditions required by the definitional principles. Therefore, we also define predicates picking out good types and terms, in mutual recursion with the provability relation.

For the most part, we define good terms implicitly as those appearing in provable sequents. We also need rules for the primitives, and for (de)constructing good terms and types. A few examples are shown:

$$\frac{\texttt{hs |- c} \quad \texttt{member t (c::hs)}}{\texttt{term\_ok t}} \qquad \frac{\texttt{type\_ok } ty_1 \quad \texttt{type\_ok } ty_2}{\texttt{type\_ok (Fun } ty_1 \ ty_2\texttt{)}} \qquad \frac{\texttt{term\_ok (Comb } t_1 \ t_2\texttt{)}}{\texttt{term\_ok } t_1} \qquad \frac{\texttt{term\_ok } tm \quad tm \texttt{ has\_type } ty}{\texttt{type\_ok } ty}$$

We continue by specifying the inference rules as in [4], but restricted to good terms and types. For example, REFL, ASSUME, and INST_TYPE:

$$\frac{\texttt{term\_ok } t}{\texttt{[] |- } t \texttt{ === } t} \qquad \frac{\texttt{term\_ok } p \quad p \texttt{ has\_type Bool}}{\texttt{[}p\texttt{] |- } p} \qquad \frac{hs \texttt{ |- } c \quad \texttt{every type\_ok (map fst } tyin\texttt{)}}{\texttt{map (INST } tyin\texttt{) } hs \texttt{ |- INST } tyin \ c}$$

[3] We write the relation we define as *hs* `|-` *c*. By contrast ⊢ *p* refers to theorems proved in HOL4.

To finish the inference system, we add the rules that extend Harrison's system – the principles of definition and the axioms.

Type Definitions To define a new type in HOL, one chooses an existing type, called the representing type, and defines a subset via a predicate. HOL types are non-empty, so the principle of type definition requires a theorem as input that proves that the chosen subset of the representing type is non-empty.

In the stateless syntax for types, the tag `Tydefn` *name p* is found on a defined type operator. It contains the name of the new type and the predicate for which a theorem of the form `[] |- Comb` *p w* was proved as evidence that the representing type is inhabited.

The rule for defining new types also introduces two new constants representing injections between the new type and the representing type. In the syntax, these constants are tagged by `Tyabs` *name p* or `Tyrep` *name p*, with the name of the new type and the predicate as above defining the subset of the representing type. To show that these constants are injections and inverses, the rule produces two theorems. We show the complete provability clause for one of the theorems below.

```
closed p ∧ [] |- Comb p w ∧ rty = domain (typeof p) ∧
aty = Tyapp (Tydefn name p) (map Tyvar (sort (tvars p))) ⇒
  [] |-
  Comb (Const abs (Fun rty aty) (Tyabs name p))
    (Comb (Const rep (Fun aty rty) (Tyrep name p))
       (Var x aty)) === Var x aty
```

Because the new type and the two new constants appear in this theorem, there is no need to explicitly give rules showing that they are `type_ok` and `term_ok`.

Constant Specifications Wiedijk [16] follows HOL Light in only admitting an equational definitional principle as a primitive, unlike other implementations of HOL which also provide a principle of constant specification that takes a theorem of the form $\exists x_1, \ldots, x_k . P$ and introduces new constants $c_1, \ldots, c_k$ with $\vdash P[c_1/x_1, \ldots, c_k/x_k]$ as their defining axiom. This is subject to certain restrictions on the types of the c_i. (The constant specification principle is supported in HOL Light, but as a derived rule, which, unfortunately, introduces an additional, less abstract form of the defining axiom.)

A disadvantage of this principle is that it presupposes a suitable definition of the existential quantifier, whereas we wish to give the semantics of the HOL language and use conservative extensions to define the logical operators. Our new constant specification mechanism overcomes this disadvantage and is less restrictive about the types of the new constants. See [2] for a fuller discussion of the new mechanism and the motivation for it. We describe it in mathematical notation rather than HOL4 syntax because the formalisation makes unwieldy but uninsightful use of list operations, since the rule may introduce multiple constants.

Given a theorem of the form $\{x_1 = t_1, \ldots, x_n = t_n\}$ `|-` p, where the free variables of p are contained in $\{x_1, \ldots, x_n\}$, we obtain new constants $\{c_1, \ldots, c_n\}$ and a theorem `|-` $p[c_1/x_1, \ldots, c_n/x_n]$. The side-conditions are that the variables $x_1, \ldots, x_n$ are distinct and the type variables of each t_i are contained in its type.

In the stateless syntax, we use the tag `Defn` *i xts p* for the *i*th constant introduced by this rule when it is applied to the theorem with hypotheses `map (λ(x,t). Var x (typeof t) === t)` *xts* and conclusion *p*,

Since the rule allows new constants to be introduced without appearing in any new theorems, we also add a clause for the new constants asserting `term_ok (Const` *x ty* `(Defn` *i xts p*`))`.

Axioms We include the three mathematical axioms—ETA_AX (not shown), SELECT_AX, and INFINITY_AX—in our inference system directly:

```
p has_type (Fun ty Bool) ∧ h |- Comb p w ⇒
  h |- Comb p (Comb (Select ty) p)

[] |-
EXISTS "f" (Fun Ind Ind)
  (AND (ONE_ONE Ind Ind (Var "f" (Fun Ind Ind)))
     (NOT (ONTO Ind Ind (Var "f" (Fun Ind Ind)))))
```

Here `EXISTS`, `AND`, and so forth are abbreviations for defined constants in stateless HOL, and are built up following standard definitions of logical constants. For example,

```
NOT =
  Comb
    (Const1 "~" (Fun Bool Bool)
       (Abs "p" Bool (IMPLIES (Var "p" Bool) FALSE)))
```

where `Const1` *name ty rhs* abbreviates

`Const` *name ty* `(Defn 0 [(`*name*`,`*rhs*`)] (Var` *name* `(typeof` *rhs*`) ===` *rhs*`))`

and shows how the rule for new specification subsumes the traditional rule for defining a constant to be equal to an existing term.

4.2 Semantics

Just as we reused much of the inference system, we were able to reuse most of Harrison's proofs in establishing soundness of the stateless HOL inference system, again with systematic modifications. The main change, apart from our extensions, is that our semantics uses inductive relations rather than functions.

The purpose of the semantics is to interpret sequents. Subsidiary concepts include valuations interpreting variables and semantic relations interpreting types and terms. We differ from Harrison stylistically in making type and term valuations finite maps with explicit domains (he uses total functions). We briefly describe the proposition expressed by each piece of the semantics as follows:

`type_valuation` τ	τ maps type variables to non-empty sets
`typeset` τ *ty mty*	(*ty* `:type`) is interpreted by (*mty* $:\mathcal{U}$) in τ
`term_valuation` τ σ	σ maps each variable to an element of the interpretation of its type
`semantics` σ τ *tm mtm*	(*tm* `:term`) is interpreted by (*mtm* $:\mathcal{U}$) in τ and σ
`type_has_meaning` *ty*	*ty* has semantics in all closing valuations
`has_meaning` *tm*	*tm* has semantics in all closing valuations, and a pair of closing valuations exists
hs `|=` *c*	*c*::*hs* are meaningful terms of type `Bool`, and, in all closing valuations where the semantics of each of the *hs* is `true`, so is the semantics of *c*

We prove that `semantics` σ τ and `typeset` τ are functional relations. Supporting definitions led us to prefer relations for two reasons. First, the semantics of defined constants in general requires type instantiation, and it is easier to state the condition it should satisfy than to calculate it explicitly. Second, defined types and constants are given semantics in terms of entities supplied by side-conditions on the definitional rules, so it is convenient to assume they hold. It made sense for Harrison to use total functions because without definitions, all terms (including ill-typed ones) can be handled uniformly.

Our inductive relations are mutually recursive: Harrison's had one-way dependency because the meaning of equality, for example, depends on the type. For definitions, we need the other way too because the meaning of a defined type depends on the term used to define it.

Another difference stems from our semantics being parametric on the choice of set theory model, (*mem*,*indset*,*ch*). We always use the free variables *mem*, *indset*, and *ch* for the model, and we often leave these arguments implicit in our notation. So, for example, `typeset` τ *ty mty* above is actually an abbreviation for `typeset0` (*mem*,*indset*,*ch*) τ *ty mty*.

A final addition we found helpful, especially for defined constants, is a treatment of type instantiation and variable substitution that is not complicated by the possibility of variable shadowing.

Now let us look at the new parts of the semantics in detail.

Semantics of Defined Types A type operator is defined by a predicate on an existing type called the representing type. Its semantics is the subset of the representing type where the predicate holds, which must be non-empty.

We define a relation `inhab` τ *p rty mty* to express that the subset of the type *rty* carved out by the predicate *p* is non-empty and equal to *mty*. The semantics of *rty* and *p* are with respect to τ (and the empty term valuation). Then we formally define the semantics of an applied type operator as follows:

`closed` *p* $\wedge$ *p* `has_type (Fun` *rty* `Bool)` $\wedge$ `length (tvars` *p*`) = length` *args* $\wedge$
`pairwise (typeset` τ`)` *args ams* $\wedge$
($\forall \tau$.

```
   type_valuation τ ∧ set (tvars p) ⊆ dom τ ⇒
     ∃ mty. inhab τ p rty mty) ∧
inhab (sort (tvars p) ⇉ ams) p rty mty ⇒
  typeset τ (Tyapp (Tydefn op p) args) mty
```

The purpose of the type arguments is to provide interpretations for type variables appearing in the predicate, hence in the first argument to `inhab` we bind[4] `sort (tvars p)` to the interpretations of the arguments. The penultimate premise, requiring that p carve a non-empty subset of *rty* for any closing τ, is necessary to ensure that a badly defined type does not accidentally get semantics when applied to arguments that happen to produce a non-empty set.

Type definition also introduces two new constants, and they are given semantics as injections between the representing type and the subset carved out of it. We only show the rule for the function to the new type, which makes an arbitrary choice in case its argument is not already in the subset. (The other is the inclusion function.)

```
typeset τ (Tyapp (Tydefn op p) args) maty ∧ p has_type (Fun rty Bool) ∧
pairwise (typeset τ) args ams ∧ τi = sort (tvars p) ⇉ ams ∧
typeset τi rty mrty ∧ semantics ⊥ τi p mp ∧
tyin = sort (tvars p) ⇉ args ⇒
  semantics σ τ
    (Const s (Fun (tyinst tyin rty) (Tyapp (Tydefn op p) args))
       (Tyabs op p))
    (abstract mrty maty (λx. if Holds mp x then x else ch maty))
```

The type definition rule returns two theorems about the new constants, asserting that they form a bijection between the new type and the subset of the representing type defined by the predicate. It is straightforward to prove this rule sound since the semantics simply interprets the new type as the subset to which it must be in bijection.

Substitution and Instantiation In Harrison's work, proving soundness for the two inference rules (INST_TYPE and INST) that use type instantiation and term substitution takes about 60% of the `semantics.ml` proof script by line count. These operations are complicated because they protect against unintended variable capture, *e.g.* instantiating α with `bool` in λ`(`x `:bool). (`x `:`α`)` triggers renaming of the bound variable. Since the semantics of defined constants uses type instantiation, we sought a simpler implementation.

The key observation is that there is always an α-equivalent term—with distinct variable names—for which instantiation is simple, and the semantics should be up to α-equivalence anyway. For any term *tm* and finite set of names s, we define `fresh_term` s *tm* as an arbitrary α-equivalent term with bound variable names that are all distinct and not in s.

We define unsafe but simple algorithms, `simple_inst` and `simple_subst`, which uniformly replace (type) variables in a term, ignoring capture. Then, under

[4] *ks* ⇉ *vs* is the finite map binding *ks* pairwise to *vs*

conditions that can be provided by `fresh_term`, namely, that bound variable names are distinct and not in the set of names appearing in the substitution, it is straightforward to show that `simple_subst` and `simple_inst` behave the same as the capture-avoiding algorithms, `VSUBST` and `INST`.

The inference rules use the capture-avoiding algorithms since they must cope with terms constructed by the user, but when we prove their soundness we first switch to a fresh term then use the simple algorithms. The theorems enabling this switch say that substitution (not shown) and instantiation respect α-equivalence:

$\vdash$ `welltyped` t_1 $\wedge$ `ACONV` t_1 t_2 $\Rightarrow$ `ACONV (INST` *tyin* t_1`) (INST` *tyin* t_2`)`

To prove these theorems, we appeal to a variable-free encoding of terms using de Bruijn indices. We define versions of `VSUBST` and `INST` that operate on de Bruijn terms, and prove that converting to de Bruijn then instantiating is the same as instantiating first then converting. The results then follow because α-equivalence amounts to equality of de Bruijn terms.

Semantics of Defined Constants The semantics of the ith constant defined by application of our principle for new specification on $\{x_1 = t_1, \ldots, x_n = t_n\}$ `|-` p can be specified as the semantics of the term t_i. This choice might constrain the constant more than the predicate p does, but the inference system guarantees that all knowledge about the constant must be derivable from p. When t_i is polymorphic, we need to instantiate its type variables to match the type of the constant. The relevant clause of the semantics is as follows.

$i <$ `length` *eqs* $\wedge$ `EL` i *eqs* `= (`s`,`t_i`)` $\wedge$ t `= fresh_term` $\varnothing$ t_i $\wedge$ `welltyped` t $\wedge$
`closed` t $\wedge$ `set (tvars` t`)` $\subseteq$ `set (tyvars (typeof` t`))` $\wedge$
`tyinst` *tyin* `(typeof` t`) =` *ty* $\wedge$ `semantics` $\bot$ τ `(simple_inst` *tyin* t`)` *mt* $\Rightarrow$
  `semantics` σ τ `(Const` s *ty* `(Defn` i *eqs* p`))` *mt*

To prove our new constant specification principle sound, we may assume $\{x_1 = t_1, \ldots, x_n = t_n\}$ `|-` p and need to show `|-` $p[c_1/x_1, \ldots, c_n/x_n]$. Given the semantics above and the interpretation of sequents, this reduces to proving the correctness of substitution, which we need to prove anyway for the INST rule.

Axioms, Soundness and Consistency The axioms do not introduce new kinds of term or type, so do not affect the semantics. We just have to characterise the constants in the axiom of infinity using the semantics for defined constants. Since our interpretation of functions is natural, mapping functions to their graphs in the set theory, the soundness proofs for the axioms are straightforward.

We have described how we prove soundness for each of our additional inference rules (that is, for definitions and axioms). We prove soundness for all the other inference rules by adapting Harrison's proofs, with improvements where possible (*e.g.* for substitution and instantation). Using the proofs for each rule, we obtain the main soundness theorem by induction on the inference system.

Theorem 2 (Soundness of Stateless HOL).

```
⊢ is_model (mem,indset,ch) ⇒
    (∀ty. type_ok ty ⇒ type_has_meaning ty) ∧
    (∀tm. term_ok tm ⇒ has_meaning tm) ∧
    ∀hs c. hs |- c ⇒ hs |= c
```

It is then straightforward to prove that there exist both provable and unprovable sequents (`VARIANT` here creates a distinct name by priming):

Theorem 3 (Consistency of Stateless HOL).

```
⊢ is_model (mem,indset,ch) ⇒
    [] |- Var x Bool === Var x Bool ∧
    ¬([] |- Var x Bool === Var (VARIANT (Var x Bool) x Bool) Bool)
```

5 From Stateful Back to Stateless

The previous sections have explained the semantics and soundness proof for stateless HOL. Our overall goal is to prove the soundness of a conventional stateful implementation, so we formalise a stateful version of HOL (our rough diamond contains a brief overview [11]) and give it semantics by translation into the stateless version.

The only significant difference between the stateful and stateless versions is that the stateless carries definitions of constants as tags on the terms and types. The translation from the stateful version, which has an explicit global context, simply inlines all the appropriate definitions into the terms and types.

We define this translation from stateful to stateless HOL using inductively defined relations for translation of types and terms. The translation of stateful sequents into stateless sequents is defined as the following relation.

```
seq_trans ((defs,hs′),c′) (hs,c) ⟺
  pairwise (term defs) hs′ hs ∧ term defs c′ c
```

Here *defs* is the global context in which the stateful theorem sequent has been proved, and `term` is the translation relation for terms. The definition of `term` (and `type` similarly) is straightforward and omitted due to space constraints.

We prove, by induction on the stateful inference system, that any sequent that can be derived in the stateful version can be translated into a provable stateless sequent.

Theorem 4 (Stateful HOL contained in stateless HOL).

```
⊢ (type_ok defs ty′ ⇒ ∃ty. type defs ty′ ty ∧ type_ok ty) ∧
(term_ok defs tm′ ⇒ ∃tm. term defs tm′ tm ∧ term_ok tm) ∧
((defs,hs′) |- c′ ⇒ ∃hs c. seq_trans ((defs,hs′),c′) (hs,c) ∧ hs |- c)
```

6 Verifying the Kernel in CakeML

To construct a verified CakeML implementation of the stateful HOL inference rules, we take the implementation of the HOL Light kernel (extended with

our constant specification principle) and define each of its functions in HOL4 using a state-and-exception monad. Using previously developed proof automation [10], these monadic functions are automatically turned into deep embeddings (CakeML abstract syntax) that are proved to implement the original monadic functions.

It only remains to show the following connection between the computation performed by the monadic functions, and the inference system for stateful HOL from §5: any computation on good types, terms and theorems will produce good types, terms and theorems according to stateful HOL. A type, term or theorem sequent is "good" if it is good according to `type_ok`, `term_ok`, or `(|-)` from stateful HOL. Here `hol_tm`, `hol_ty` and `hol_defs` translate into the implementation's representations.

```
⊢ TYPE defs ty ⟺ type_ok (hol_defs defs) (hol_ty ty)
⊢ TERM defs tm ⟺ term_ok (hol_defs defs) (hol_tm tm)
⊢ THM defs (Sequent asl c) ⟺
    (hol_defs defs,map hol_tm asl) |- hol_tm c
```

The prover's state s implements logical context *defs*, if `STATE` s *defs* holds. We omit the definition of `STATE`.

For each monadic function, we prove that good inputs produce good output. For example, for the `ASSUME` function, we prove that, if the input is a good term and the state is good, then the state will be unchanged on exit and if the function returned something (via `HolRes`) then the return value is a good theorem:

```
⊢ TERM defs tm ∧ STATE s defs ∧ ASSUME tm s = (res,s') ⇒
    s' = s ∧ ∀th. res = HolRes th ⇒ THM defs th
```

We prove a similar theorem for each function in the kernel, showing that they implement the stateful inference rules correctly. As another example, take the new rule for constant specification: we prove that if the state is updated then the state is still good and the returned theorem is good.

```
⊢ THM defs th ∧ STATE s defs ⇒
  case new_specification th s of
    (HolRes th,s') ⇒ ∃d. THM (d::defs) th ∧ STATE s' (d::defs)
  | (HolErr msg,s') ⇒ s' = s
```

By expanding the definition of `THM` in these theorems, then applying Theorems 4 and 2, we see that each monadic function implements a valid deduction according to the semantics of HOL. We then compose with the automatically synthesised certificate theorem for the CakeML implementation, to finish the proof about the CakeML implementation of the kernel. The automatically proved certificate theorem for the monadic `new_specification` function is shown below. These certificate theorems are explained in Myreen and Owens [10].

```
⊢ DeclAssum ml_hol_kernel_decls env ⇒
  EvalM env (Var (Short "new_specification"))
    ((PURE HOL_KERNEL_THM_TYPE -M-> HOL_MONAD HOL_KERNEL_THM_TYPE)
       new_specification)
```

7 Related Work

For classical higher-order logic, apart from Harrison's mechanisation [4] of the semantics that we extend here, Krauss and Schropp [6] have also formalised a translation to set theory automatically producing proofs in Isabelle/ZF.

Considering other logics, Barras [3] has formalised a reduced version of the calculus of inductive constructions, the logic used by the Coq proof assistant, giving it a semantics in set theory and formalising a soundness proof in Coq itself. The approach is modular, and Wang and Barras [15] have extended the framework and applied it to the calculus of constructions plus an abstract equational theory.

Myreen and Davis [9] formalised Milawa's ACL2-like first-order logic and proved it sound using HOL4. This soundness proof for Milawa produced a top-level theorem which states that the machine-code which runs the prover will only print theorems that are true according to the semantics of the Milawa logic. Since Milawa's logic is weaker than HOL, it fits naturally inside HOL without encountering any delicate foundational territory such as the assumption on Theorem 1.

Other noteworthy prover verifications include a simple first-order tableau prover by Ridge and Margetson [13] and a SAT solver algorithm with many modern optimizations by Marić [8].

8 Conclusion

CakeML In the context of the CakeML project overall (`https://cakeml.org`), our verified implementation of the HOL Light kernel is an important milestone: the first verified application other than the CakeML compiler itself. This validates both the methodology of working in HOL4 and using automated synthesis to produce verified programs, and the usefulness of the CakeML language for a substantial application. At this point we have a verified compiler, and a verified application to run on it. What remains to be done is the creation of reasoning tools for CakeML programs that do not fit nicely into the HOL4 logic. In particular, we want to establish that arbitrary – possibly malicious – client code that constructs proofs using the verified HOL Light kernel cannot subvert the module-system enforced abstraction that protects the kernel and create false theorems.

Reflections on Stateless HOL Our choice to use stateless HOL was motivated by a desire to keep the soundness proof simple and close to Harrison's by avoiding introduction of a context for definitions. We avoided the context, but stateless HOL did introduce some significant complications: the inference system and semantics both become mutually recursive, and care must be taken to avoid terms with no semantics. The dependence of `typeset` on `semantics` is necessary for type definitions, but it could perhaps be factored through a context. Similarly, the move to relations instead of functions seems reasonable given the side-conditions on the definitional rules, but one could instead use a total lookup function to get definitions from a context.

Overall it is not clear that stateless HOL saved us any work and it is clear that it led to some loss of abstraction in our formalisation. Separating the context from the representation of types and terms is closer to the standard approaches adopted in the mathematical logic literature and would help to separate concerns about the conservative extension mechanisms (which we expect to be loosely specified) from concerns about the semantics of types and terms (which we expect to be deterministic functions of the context). After submitting this paper, we experimented with a formalisation of the semantics using a separate context, and would now recommend the context-based approach as simpler and more expressive.

Acknowledgements We thank Mike Gordon, John Harrison, Roger Jones, Michael Norrish, Konrad Slind, and Freek Wiedijk for useful discussions and feedback. The first author acknowledges support from Gates Cambridge. The third author was funded by the Royal Society, UK.

References

1. Arthan, R.: HOL formalised: Semantics, `http://www.lemma-one.com/ProofPower/specs/spc002.pdf`
2. Arthan, R.: HOL constant definition done right. In: Interactive Theorem Proving. These proceedings, Springer (2014)
3. Barras, B.: Sets in Coq, Coq in sets. J. Formalized Reasoning 3(1) (2010)
4. Harrison, J.: Towards self-verification of HOL Light. In: International Joint Conference on Automated Reasoning (IJCAR). LNCS, vol. 4130. Springer (2006)
5. Harrison, J.: HOL Light: An overview. In: TPHOLs. LNCS, vol. 5674. Springer (2009), `http://www.cl.cam.ac.uk/~jrh13/hol-light/`
6. Krauss, A., Schropp, A.: A mechanized translation from higher-order logic to set theory. In: ITP. LNCS, vol. 6172. Springer (2010)
7. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: a verified implementation of ML. In: Principles of Prog. Lang. (POPL). ACM Press (2014)
8. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. 411(50), 4333–4356 (2010)
9. Myreen, M.O., Davis, J.: The reflective Milawa theorem prover is sound (down to the machine code that runs it). In: ITP. LNCS, Springer (2014)
10. Myreen, M.O., Owens, S.: Proof-producing translation of higher-order logic into pure and stateful ML. Journal of Functional Programming FirstView (1 2014)
11. Myreen, M.O., Owens, S., Kumar, R.: Steps towards verified implementations of HOL Light. In: ITP. LNCS, vol. 7998. Springer (2013), "Rough Diamond" section
12. Norrish, M., Slind, K., et al.: The HOL System: Logic, 3rd edn., `http://hol.sourceforge.net/documentation.html`
13. Ridge, T., Margetson, J.: A mechanically verified, sound and complete theorem prover for first order logic. In: TPHOLs. LNCS, vol. 3603. Springer (2005)
14. Slind, K., Norrish, M.: A brief overview of HOL4. In: Theorem Proving in Higher Order Logics (TPHOLs). LNCS, vol. 5170. Springer (2008)
15. Wang, Q., Barras, B.: Semantics of intensional type theory extended with decidable equational theories. In: CSL. LIPIcs, vol. 23. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
16. Wiedijk, F.: Stateless HOL. In: Types for Proofs and Programs (TYPES). EPTCS, vol. 53 (2009)