Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Self-Formalisation of Higher-Order Logic

Semantics, Soundness, and a Verified Implementation

Ramana Kumar · Rob Arthan ·
Magnus O. Myreen · Scott Owens

Received: date / Accepted: date

Abstract We present a mechanised semantics for higher-order logic (HOL), and a proof of soundness for the inference system, including the rules for making definitions, implemented by the kernel of the HOL Light theorem prover. Our work extends Harrison's verification of the inference system 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. Apart from adding support for definitions and synthesising an implementation, we improve on Harrison's work by making our model of HOL parametric on the universe of sets, and we prove soundness for an improved principle of constant specification in the hope of encouraging its adoption. Our semantics supports defined constants directly via a context, and we find this approach cleaner than our previous work formalising Wiedijk's Stateless HOL.

What is New for the Special Issue? A version of this paper [16] was originally published in the conference proceedings of ITP 2014, and describes a semantics for Stateless HOL [33]. By contrast, the semantics presented in this paper works directly on standard HOL and uses a theory context to handle definitions (in the style of Arthan [3]). We found this new style of semantics to be both easier to understand and simpler to work with. It also enables us

The first author was supported by the Gates Cambridge Trust. The second author was funded in part by the EPSRC (grant number EP/K503769/1). The third author was partially supported by the Royal Society UK and the Swedish Research Council.

R. Kumar
Computer Laboratory, University of Cambridge
E-mail: Ramana.Kumar@cl.cam.ac.uk

R. D. Arthan
Department of Computer Science, University of Oxford
E-mail: rda@lemma-one.com

M. O. Myreen
CSE Department, Chalmers University of Technology
E-mail: Myreen@chalmers.se

S. Owens
School of Computing, University of Kent
E-mail: S.A.Owens@kent.ac.uk

to better characterise the abstraction barrier provided by the generalised rule of constant specification. As well as describing an improved formalisation, we use the space available in this Special Issue to give a more complete description than was possible in the conference paper.

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 (see Myreen et. al. [26] and Kumar et. al. [16]), to produce a verified machine-code implementation of a HOL theorem prover. This paper continues the top half of the project: soundness of the logic, and a verified implementation of the logical kernel in CakeML [17].

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 its correctness. A theorem prover implementation usually sits at the centre of the trusted code base for verification work, so effort spent verifying the theorem 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.

We build on Harrison's proof of the consistency of HOL without definitions [10], which shares our larger goal of verifying concrete HOL theorem prover implementations, and advance this project by verifying an implementation of the HOL Light [11] 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 to the theorem prover's logic.

Definition by conservative extension is one of the hallmarks of using HOL, and makes the logic expressive with a small number of primitives. When considering implementations of the logic, the definitional rules are important because defined constants are not merely abbreviations, but are distinguished variants in the datatypes representing the syntax of the logic. For this reason, we think it is important to formalise the rules of definition and to use a semantic framework that supports that goal.

The rule for defining term constants that we formalise, which is actually a rule for constant specification, generalises the one found in the various HOL systems by adding support for implicit definitions with fewer constraints and no new primitives. A full account of its history and design can be found in Arthan [4] (and the extended version in this issue). As reported there, our proof of its soundness has cleared the way for adoption of the improved rule.

The main result of the work described in this paper is a verified CakeML implementation of the logical kernel of HOL Light. We intend to use this kernel implementation as the foundation for a verified machine-code implementation of a complete LCF-style [22] theorem prover with the kernel as a module. The specific contributions of this paper are:

- a formal semantics for HOL that supports definitions (§4), against a new specification of set theory (§3),

- proofs of soundness and consistency (§5) for the HOL inference system, including type definitions, the new rule for constant specification, and the three axioms used in HOL Light,
- a verified implementation of a theorem-prover kernel (based on HOL Light's) in CakeML (§6) that should be a suitable basis for a verified implementation of a theorem prover in machine code.

All our definitions and proofs have been formalised in the HOL4 theorem prover [29] and are available from `https://cakeml.org`.[1] We briefly discuss the natural question of trusting an unverified theorem prover for this work, and how we might skate along the barriers around true self-verification, in the conclusion.

2 Approach

Higher-order logic, or HOL, as we use the term in this paper, is a logic that was first proposed and implemented as an alternative object logic for LCF by Mike Gordon [9]. It adds polymorphism *à la* Milner [21] to Church's simple type theory ([7], [2, Chapter 5]) resulting in a tool that has proved remarkably powerful over the last 30 years in a wide range of mechanised theorem-proving applications. The logic is well-understood: a rigorous informal account of its simple and natural set-theoretic semantics has been given by Pitts and is included in the HOL4 theorem prover's documentation [27]. Our aim here is to build a formal model of this semantics, against which we can verify a formalisation of the HOL inference system. We will then take the inference system as a specification, against which we can verify an implementation of a theorem-prover kernel based on HOL Light's. This task can be broken down into the following steps:

1. Specify the set-theoretic notions needed. (§3)
2. Define the syntax of HOL types, terms, and sequents. (§4.1)
3. Define semantic functions assigning appropriate sets to HOL types and terms, and use these to specify validity of a sequent. (§4.2)
4. Define the inference system: how to construct sequents-in-context (the rules of inference), and how to extend a context (the rules of definition). (§4.3)
5. Verify the inference system: prove that every derivable sequent is valid. Deduce that the inference system is consistent: it does not derive a contradiction. (§5)
6. Write an implementation of the inference system as recursive functions in HOL, and verify it against the relational specification of the inference rules. (§6.1)
7. Synthesise a verified implementation of the inference rules in CakeML. (§6.2)

We construct the specifications, definitions, and proofs above in HOL itself (using the HOL4 theorem prover), in the style of Harrison's work [10] towards self-verification of HOL Light. Compared to Harrison's work, items 6–7 are new, items 2–5 are extended and reworked to support a context of definitions, and item 1 uses an improved specification. Our implementation language, CakeML [16], is an ML-like language (syntactically a subset of Standard ML) with a formal semantics specified in HOL and for which there are automated techniques (Myreen and Owens [25]) for accomplishing item 7.

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

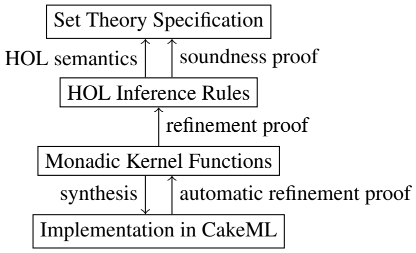

Fig. 1 Producing a Sound Implementation by Refinement

The bulk of our programme is concerned with soundness rather than consistency: an inference rule is sound if whenever its antecedents hold in all models[2] then so does its succedent — a statement which does not depend on the existence of models. However, to validate our formalisation of the semantics it is useful to carry out some proofs that give evidence of consistency. Assuming HOL is consistent, Gödel's second incompleteness theorem [8] prevents us from proving HOL's consistency in HOL. However, we can prove the consistency in HOL of HOL with its axiom of infinity omitted. This gives fairly convincing evidence that we have correctly captured the semantics.

Our approach avoids making any axiomatic extensions to HOL. We also isolate results that are dependent on the set-theoretic axiom of infinity, so that as much as possible is proved without any undischarged assumptions. We are able to use assumptions on our theorems rather than asserting new axioms in the logic because we formalise a specification of set theory rather than defining a particular instance of a set theory as Harrison [10] did.

The results of following the plan above fit together as shown in Figure 1. 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 invariants. In particular, a good value of type *thm* must be a HOL sequent that is valid according to the set-theoretic semantics.

We prove these results by composing the three proof layers in the diagram. The top layer is the result of steps 1–5. The HOL semantics gives meaning to HOL sequents, from which we obtain definitions of validity and consistency. Validity concerns the truth of a sequent within a fixed context of definitions, whereas consistency is about whether the context itself has a model. The soundness proof says that each of the HOL inference rules preserves validity of sequents, and each of the HOL principles of definition preserves consistency of the context.

The middle layer corresponds to step 6. As described in our previous work [26, 16], we define shallowly-embedded HOL functions, using a state-exception monad, for each of the HOL Light kernel functions. These "monadic kernel functions" are a hand-crafted implementation, but are written following the original OCaml code for HOL Light closely, and we prove that they implement the 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 inference system.

[2] Throughout the paper, we are concerned only with standard models of HOL, that is, where the Boolean and function types and the equality constant are interpreted in the standard way, and function spaces are full (unlike in Henkin semantics [12]). See the paragraph on standard interpretations in Section 4.2.

Finally, for step 7 we use the method developed by Myreen and Owens [25] to synthesise CakeML implementations of the monadic kernel functions. The automatic translation from shallowly- to deeply-embedded code is proof-producing. We use the generated certificate theorems to complete the refinement proof that links theorem values produced by the implementation to sequents that are semantically valid.

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.

The focus for most of the rest of the paper is the top layer of Figure 1, wherein we describe how we formalise the syntax, semantics, and soundness of HOL with full support for the definition of new types and constants as well as support for non-definitional context-extension. We return to the verified implementation part of the story, corresponding to the lower layers of the figure, in Section 6. The paper ends with a discussion about self-verification and related work.

Terminology and Notation As our programme involves both refinement proofs (linking implementations to specifications) and soundness proofs (linking an inference system to the semantics of a logic), and for the latter both our meta-logic and object-logic are HOL, we often need to refer to similar but different things and some care must be paid for clarity. By "HOL" we refer, as in the section above, to higher-order logic itself. Particular inference systems for HOL are implemented by interactive theorem-proving systems. The two theorem provers of interest to us are HOL Light, because we formalise its inference system and use its implementation as inspiration for our implementation in CakeML; and HOL4, because we use it to mechanise our proofs. We use "HOL4" and "HOL Light" unqualified to refer to the theorem provers, and clarify explicitly when we mean the inference system instead. HOL4 implements a different inference system from HOL Light's, but the two are inter-translatable.

We include extracts, generated by HOL4, from our formalisation in the paper. These include definitions, for example, here is the standard library function for checking a predicate holds for all elements of a list:

$$\begin{array}{l} \mathsf{every}\ P\ [\,] \iff \mathsf{T} \\ \mathsf{every}\ P\ (h{::}t) \iff P\ h \wedge \mathsf{every}\ P\ t \end{array}$$

Since the result of $\mathsf{every}\ P\ ls$ is Boolean, we use $(\iff)$ in the defining equations; at other types we simply use $(=)$. As well as definitions we have theorems, which are shown with a turnstile, for example:

$$\vdash \neg\mathsf{every}\ P\ ls \iff \mathsf{exists}\ ((\neg) \circ P)\ ls$$

Free variables may appear in theorems; semantically, they behave as if universally quantified. Datatype definitions are shown as in the following example of the polymorphic option type with two constructors:

$$\alpha\ \mathit{option} = \mathsf{None} \mid \mathsf{Some}\ \alpha$$

Terms are sometimes annotated with their types, for example: $(\mathsf{Some} : \mathit{bool} \to \mathit{bool\,option})\ \mathsf{F}$. Quantifiers are printed as binders, as in $\forall x.\ \exists y.\ x \neq \mathsf{Some}\ y$, although in HOL the quantifiers are ordinary constants (that operate on predicates, that is, functions with codomain

bool). The existential quantifier in the previous sentence might more pedantically be printed as an application of $(\exists)$ to $\lambda y.\ x \neq \mathsf{Some}\ y$. Finally, we show the rules of inductive relations using a horizontal line to separate premises from the conclusion. Thus the rule, $\vdash\ R\,x\,y \land R^*\ y\,z \Rightarrow R^*\ x\,z$, about the reflexive transitive closure of a relation can also be written as follows:

$$\frac{\begin{array}{c}R\,x\,y\\ R^*\,y\,z\end{array}}{R^*\,x\,z}$$

3 Set Theory Specification

We now begin the technical part of the paper, starting with a specification of set theory over which the semantics of HOL will be specified in the next section. Axiomatic set theory can be specified in terms of a single binary relation, the membership relation. In HOL, we can give a quite straightforward development of the basic concepts of set theory as may be found in any standard text (e.g., Vaught [30]) thus achieving clarity through familiarity and making it easy to compare our formalisation with Pitts' informal account [27]. Since our specification is in HOL, we can write the membership relation and its axioms within the logic without resorting to the metavariables and schemata required in the first-order setting[3].

The most common set theory in textbook accounts is Zermelo-Fraenkel set theory ZF. However, ZF's axiom of replacement plays no rôle in giving semantics to HOL, so all we need are the axioms of Zermelo's original system: extensionality, separation (a.k.a. comprehension or specification), power set, union, (unordered) pairing, and infinity. It will be convenient to deal with the axiom of infinity separately. So we begin by defining a predicate on membership relations, $\mathsf{is\_set\_theory}\ (\mathit{mem} : \mathscr{U}\ \mathsf{reln})$, that asserts that the membership relation satisfies each of the Zermelo axioms apart from the axiom of infinity. By formalising the set-theoretic universe as a type variable, $\mathscr{U}$, we can specify what it means to be a model of Zermelo set theory, while deferring the problem of whether such a model can be constructed.

The specification of the set-theoretic axioms is as follows:

$$\begin{array}{l}\mathsf{is\_set\_theory}\ \mathit{mem} \iff\\ \quad \mathsf{extensional}\ \mathit{mem} \land (\exists \mathit{sub}.\ \mathsf{is\_separation}\ \mathit{mem}\ \mathit{sub}) \land\\ \quad (\exists \mathit{power}.\ \mathsf{is\_power}\ \mathit{mem}\ \mathit{power}) \land (\exists \mathit{union}.\ \mathsf{is\_union}\ \mathit{mem}\ \mathit{union}) \land\\ \quad \exists \mathit{upair}.\ \mathsf{is\_upair}\ \mathit{mem}\ \mathit{upair}\end{array}$$

$$\begin{array}{l}\mathsf{extensional}\ \mathit{mem} \iff\\ \quad \forall x\,y.\ x = y \iff \forall a.\ \mathit{mem}\ a\ x \iff \mathit{mem}\ a\ y\end{array}$$

$$\begin{array}{l}\mathsf{is\_separation}\ \mathit{mem}\ \mathit{sub} \iff\\ \quad \forall x\,P\,a.\ \mathit{mem}\ a\ (\mathit{sub}\ x\ P) \iff \mathit{mem}\ a\ x \land P\ a\end{array}$$

$$\begin{array}{l}\mathsf{is\_power}\ \mathit{mem}\ \mathit{power} \iff\\ \quad \forall x\,a.\ \mathit{mem}\ a\ (\mathit{power}\ x) \iff \forall b.\ \mathit{mem}\ b\ a \Rightarrow \mathit{mem}\ b\ x\end{array}$$

[3] In our statement of the separation axiom, if the set x is infinite then P ranges over an uncountable set corresponding to all subsets of x. Technically, this is a significant strengthening of the axiom of separation, since it is not restricted to the countably many subsets of x that can be specified in the language of first-order set theory. However, this is irrelevant to our purposes: we are interested in a standard semantics for HOL not a Henkin semantics [12].

$$\begin{array}{l}\mathsf{is\_union}\ \mathit{mem}\ \mathit{union} \iff \\ \quad \forall x\, a.\ \mathit{mem}\ a\ (\mathit{union}\ x) \iff \exists b.\ \mathit{mem}\ a\ b \land \mathit{mem}\ b\ x\end{array}$$

$$\begin{array}{l}\mathsf{is\_upair}\ \mathit{mem}\ \mathit{upair} \iff \\ \quad \forall x\, y\, a.\ \mathit{mem}\ a\ (\mathit{upair}\ x\ y) \iff a = x \lor a = y\end{array}$$

To state the (set-theoretic) axiom of infinity, we define what it means for an element of $\mathscr{U}$ to be infinite: $\mathsf{is\_infinite}\,\mathit{mem}\ s \iff \neg\mathsf{FINITE}\ \{\ a \mid \mathit{mem}\ a\ s\ \}$. Here FINITE is inductively defined (in HOL) for sets-as-predicates, so we are saying a set is infinite if it does not have finitely many members. The (set-theoretic) axiom of infinity asserts that such a set exists.

3.1 Derived Operations

Using the axioms above, it is straightforward to define standard set-theoretic constructions that will support our specification of the semantics of HOL. In this subsection, we introduce some of our notation for such derived operations. All our definitions are parametrised by the membership relation, $(\mathit{mem} : \mathscr{U}\ \mathsf{reln})$; we often elide this argument with a pretty-printing abbreviation, for example writing $\mathsf{Funspace}\,x\,y$ instead of $\mathsf{funspace}\,\mathit{mem}\ x\ y$.[4] When *mem* is used as a binary operator, we will from now on write it infix as $x \lessdot y$ instead of $\mathit{mem}\ x\ y$. Also, most of our theorems are under the assumption $\mathsf{is\_set\_theory}\ \mathit{mem}$; we often elide this assumption.

Using the axiom of separation we define the empty set and prove it has no elements, then using pairing we define sets containing exactly one and exactly two elements. The latter serves as our representation of the set of Booleans. We have the following, shown with and without abbreviations for clarity:

(full notation)

$$\begin{array}{l}\vdash\ \mathsf{is\_set\_theory}\ \mathit{mem} \Rightarrow \\ \quad \forall x.\ \mathit{mem}\ x\ (\mathsf{two}\ \mathit{mem}) \iff x = \mathsf{true}\ \mathit{mem} \lor x = \mathsf{false}\ \mathit{mem}\end{array}$$

(abbreviated notation)

$$\vdash\ x \lessdot \mathsf{Boolset} \iff x = \mathsf{True} \lor x = \mathsf{False}$$

We define Kuratowski pairs (defn. V in [18]) as well as the cross product of two sets so that the following properties hold.

$$\begin{array}{l}\vdash\ (a,b) = (c,d) \iff a = c \land b = d \\ \vdash\ a \lessdot x \times y \iff \exists b\ c.\ a = (b,c) \land b \lessdot x \land c \lessdot y\end{array}$$

From cross products, we can define relations, and then functions (graphs) as functional relations. $\mathsf{Abstract}\ s\ t\ f$ is our notation for the subset of $s \times t$ that is the graph of $(f : \mathscr{U} \to \mathscr{U})$, and $a\ '\ x$ denotes application of such a set-theoretic function a to an argument x. The main theorem about application in set theory is that it acts like application in HOL:

$$\vdash\ x \lessdot s \land f\ x \lessdot t \Rightarrow \mathsf{Abstract}\ s\ t\ f\ '\ x = f\ x$$

[4] We follow a loose convention that capitalised functions have the hidden *mem* argument. Be aware that datatype constructors, which are also capitalised, are amongst the exceptions.

Furthermore, we know functions obey extensional equality:

$$\begin{aligned}&\vdash\ (\forall x.\ x \lessdot s \Rightarrow f_1\ x \lessdot t_1 \wedge f_2\ x \lessdot t_2 \wedge f_1\ x = f_2\ x) \Rightarrow\\ &\quad \mathsf{Abstract}\ s\ t_1\ f_1 = \mathsf{Abstract}\ s\ t_2\ f_2\end{aligned}$$

We define the set of functions between two sets, and prove that its elements are precisely those made using Abstract:

$$\begin{aligned}&\vdash\ (\forall x.\ x \lessdot s \Rightarrow f\ x \lessdot t) \Rightarrow \mathsf{Abstract}\ s\ t\ f \lessdot \mathsf{Funspace}\ s\ t\\ &\vdash\ a \lessdot \mathsf{Funspace}\ s\ t \Rightarrow\\ &\quad \exists f.\ a = \mathsf{Abstract}\ s\ t\ f \wedge \forall x.\ x \lessdot s \Rightarrow f\ x \lessdot t\end{aligned}$$

The derived operations in our formalisation (a selection of which were shown in this subsection) may be considered as an alternative description (compared to the Zermelo axioms) of the interface required for giving semantics to HOL. In other words, any structure supporting such constructions as pairs and functions is suitable.

It is worth noting that a relation $(mem : \mathscr{U}\ \mathit{reln})$ satisfying is_set_theory *mem* will automatically satisfy the set-theoretic axiom of choice (AC), that is, we can prove the following:

$$\vdash\ \forall x.\ (\forall a.\ a \lessdot x \Rightarrow \mathsf{inhabited}\ a) \Rightarrow \exists f.\ \forall a.\ a \lessdot x \Rightarrow f\ {'}\ a \lessdot a$$

We prove this by using the axiom of choice in HOL (i.e., the language we are using to formalise set theory) to provide a HOL function $(g : \mathscr{U} \to \mathscr{U})$ such that for every non-empty $(a : \mathscr{U})$, we have $g\ a \lessdot a$. Then, given a set $(x : \mathscr{U})$ whose members are all non-empty, we use Abstract (ultimately depending on our strong form of the axiom of separation) to define the graph of g restricted to the members of x (as a member of the set-theoretic universe $\mathscr{U}$) and hence conclude that AC holds in our set theory. This is a consequence of our decision to give a standard semantics to HOL.

3.2 Consistency of the Specification

So far we have specified a predicate on a membership relation asserting that it represents a set theory with our desired structure (without the axiom of infinity). As a sanity check to convince us that this part of the specification is consistent, we can construct a membership relation that satisfies the predicate.

The hereditarily finite sets provide a simple model of set theory without the axiom of infinity. This model can be represented concretely by taking $\mathscr{U}$ to be the type *num* of natural numbers and defining *mem n m* to hold whenever the *n*th bit in the binary representation of *m* is not zero. Of course, the axiom of infinity fails in this model since every set in the model is finite. Nevertheless, we can prove that it satisfies the other axioms (V_mem is essentially the membership relation just described):

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

This shows that the notion of a set theory that we have formalised is not vacuous. One might argue that we could just use the monomorphic type *num* in place of the type variable $\mathscr{U}$. Or a little more abstractly, we could introduce a new type witnessed by the above construction on *num*. However, we wish to state some properties that are conditional on the set-theoretic axiom of infinity. Unfortunately, the axiom of infinity is provably false in a model subtyped from the countable set *num* and so results that assumed the axiom of infinity would be trivially true.

Instead, if we identify the universe of the hereditarily finite sets construction with a the right-hand summand of the polymorphic type $\alpha + \mathit{num}$, we can define a subtype $\alpha\ V$ of $\alpha + \mathit{num}$ whose defining property is the existence of a membership relation satisfying the Zermelo axioms other than infinity. Hence we can introduce a constant $\mathsf{V\_mem}$ of type $\alpha\ V\ \mathit{reln}$ with $\vdash \mathsf{is\_set\_theory}\ \mathsf{V\_mem}$ as its defining property. Thus if we work with $\mathsf{V\_mem}$ the axiom of infinity is not provably false and we can meaningfully take it is as an assumption when necessary.

In the remainder of our development, we leave *mem* as a free variable and add one or both of the assumptions, $\mathsf{is\_set\_theory}\ \mathit{mem}$ and $\exists \mathit{inf}.\ \mathsf{is\_infinite}\ \mathit{mem}\ \mathit{inf}$, whenever they are required. We provide $\mathsf{V\_mem}$ in this section as a possible non-contradictory instantiation for the free variable *mem* in our theorems. Any instantiation that satisfies both assumptions would do, but we know we cannot exhibit one within HOL itself, so we prefer to leave the theorems uninstantiated. Our decision to leave *mem* loosely specified (i.e., as a free variable) throughout the development is made easier by HOL4's parsing/printing support for hiding the free variable. In a theorem prover without such syntactic abbreviations, the notational clutter might have lent some encouragement to picking an instantiation up front.

Comparison to Harrison's Approach Rather than formalising a specification of axiomatic set theory (which can then be instantiated), Harrison [10] constructs his model of HOL in HOL at the same time as proving its requisite properties. In fact, his proof scripts allow one to choose (by rearranging comments) between an axiomatic formalisation of the set theory with an axiom of infinity or a conservative definition of the set theory without that axiom. More specifically, with the first option, he declares a new type, intended to be the universe of sets and asserts an axiom about its size, while in the second option he defines the type to be countably infinite. He then (in both options) uses what amounts to a type system (Harrison calls the types "levels") to define a coherent notion of membership in terms of injections into the universe. As a result, his sets are not extensional since there are empty sets at every level; because of this technicality his construction does not satisfy our $\mathsf{is\_set\_theory}$.

Under the conservative option, Harrison still achieves a model without axiomatic extensions for HOL without the axiom of infinity, since he can prove his size axiom on a construction similar to our $\mathsf{V\_mem}$ above. The disadvantage is that the set-theoretic axiom of infinity is provably false when one chooses the conservative option. In our approach, the polymorphism means that the set-theoretic axiom of infinity is unprovable rather than false, and so it is meaningful to prove theorems with that axiom as an assumption.

Harrison's use of levels is motivated by the desire to assert just one axiom: the cardinality property of the universe. With only this property, levels are required to distinguish different embeddings into the universe (e.g., to distinguish powersets from cross products). Our approach with an explicit membership relation gives us a specification where these distinctions are explicit. We do not need to appeal to a theory of cardinalities in the meta-logic, since the assumptions we make ($\mathsf{is\_set\_theory}\ \mathit{mem} \wedge \exists \mathit{inf}.\ \mathsf{is\_infinite}\ \mathit{mem}\ \mathit{inf}$) mirror the standard axioms of Zermelo set theory.

4 HOL Specification

With our specification of set theory in place, we now turn to the task of specifying the syntax and semantics of HOL. At the level of terms and types, our specification of the syntax is almost the same as Harrison's [10]. Our terms are simplified by not needing to bake in any primitive constants since we support a general mechanism for introducing new constants, and our approach to substitution and instantiation of bound variables is improved. Also, our abstraction terms match the implementation better by using a term (rather than just a name and type) for the bound variable.

At the level of sequents, we introduce the notion of a *theory* describing the defined constants, which we implement in the inference system as a *context* of theory-extending updates. Compared to our previous work [16] on Stateless HOL [33], where information about defined constants is carried on the terms and types themselves, this separate context of definitions makes the inference system clearer and allows us to easily quantify over all interpretations of the constants.

We present the HOL specification concisely, but give the important definitions in full so that it might serve as a reference.

4.1 Sequents: the judgements of the logic

Formally, derivations in HOL produce judgements of the following form[5]:

$$(thy,h) \;\texttt{|-}\; c$$

This judgement is known as a *sequent*. It has a conclusion, c, a set of hypotheses (represented by a list of terms), h, and is interpreted in a theory, *thy* consisting of axioms and a signature. The meaning of a sequent is that the conclusion is true whenever all the hypotheses are true, all the axioms are true, and all the terms are well-formed with respect to the signature. We begin our specification at the bottom of this structure, starting with terms and types.

4.1.1 Terms and Types

The syntax of HOL is the syntax of the (polymorphic) simply-typed lambda-calculus. Types are either variables or applied type operators.

$$\mathit{type} \;=\; \mathsf{Tyvar}\ \mathit{string} \mid \mathsf{Tyapp}\ \mathit{string}\ (\mathit{type\ list})$$

Primitive type operators include Booleans and function spaces. We abbreviate Tyapp `"bool"` $[]$ by Bool and Tyapp `"fun"` $[x;\ y]$ by Fun $x\ y$.

A term is either a variable, a constant, a combination (application), or an abstraction.

$$\begin{aligned}&\mathit{term} \;=\\&\quad \mathsf{Var}\ \mathit{string\ type}\\&\mid \mathsf{Const}\ \mathit{string\ type}\\&\mid \mathsf{Comb}\ \mathit{term\ term}\\&\mid \mathsf{Abs}\ \mathit{term\ term}\end{aligned}$$

Variables carry their types: two variables with the same name but different types are distinct. We expect the first argument to Abs to be a variable, but use a *term* so the implementation can avoid destructing and reconstructing variables whenever it manipulates an abstraction.

Constants also carry a type, but are identified by their name: the type is there only to indicate the instantiation of polymorphic constants. (Different constants with the same name are disallowed, as we will see when we describe the signature of a theory and how it is updated.) The sole primitive constant is equality; we abbreviate Const `"="` (Fun ty (Fun ty Bool)) by Equal ty.

[5] We use the symbol (|-) for the sequents defined in our specification of HOL, reserving ($\vdash$) for theorems proved in the meta-logic (that of HOL4).

Well-Typed Terms The datatype above might better be called “pre-terms”, because the only terms of interest are those that are well-typed. Every well-typed term has a unique type, which is specified by the following relation.

$$\frac{}{(\mathsf{Var}\ n\ ty)\ \mathsf{has\_type}\ ty} \qquad\qquad \frac{}{(\mathsf{Const}\ n\ ty)\ \mathsf{has\_type}\ ty}$$

$$\frac{\begin{array}{c} s\ \mathsf{has\_type}\ (\mathsf{Fun}\ dty\ rty) \\ t\ \mathsf{has\_type}\ dty \end{array}}{(\mathsf{Comb}\ s\ t)\ \mathsf{has\_type}\ rty} \qquad \frac{t\ \mathsf{has\_type}\ rty}{(\mathsf{Abs}\ (\mathsf{Var}\ n\ dty)\ t)\ \mathsf{has\_type}\ (\mathsf{Fun}\ dty\ rty)}$$

Well-typed terms differ from pre-terms only in that the first argument of every combination has a function type, where the domain matches the second argument’s type, and the first argument of every abstraction is a variable. We say $\mathsf{welltyped}\ tm \iff \exists ty.\ tm\ \mathsf{has\_type}\ ty$ and it is straightforward to define a function, typeof, to calculate the type of a term if it exists so that $\mathsf{welltyped}\ tm \iff tm\ \mathsf{has\_type}\ (\mathsf{typeof}\ tm)$ holds.

Two operations over terms and types remain for us to describe, namely alpha-equivalence and substitution. Both are complicated by the need to correctly implement the concept of variable binding.

4.1.2 Alpha-Equivalence

Terms are alpha-equivalent when they are equal up to a renaming of bound variables. The key idea of Harrison’s original approach is to formalise when two variables are equivalent in a context of pairs of equivalent bound variables.

$$\begin{array}{l} \mathsf{avars}\ []\ (v_1,v_2) \iff v_1 = v_2 \\ \mathsf{avars}\ ((b_1,b_2)::bvs)\ (v_1,v_2) \iff \\ \quad v_1 = b_1 \land v_2 = b_2 \lor v_1 \neq b_1 \land v_2 \neq b_2 \land \mathsf{avars}\ bvs\ (v_1,v_2) \end{array}$$

The variables must be equal to some pair of bound variables (or to themselves) without either of them being equal to (captured by) any of the bound variables above. We lift this relation up to terms, for example:

$$\frac{\mathsf{avars}\ bvs\ (\mathsf{Var}\ x_1\ ty_1,\mathsf{Var}\ x_2\ ty_2)}{\mathsf{aterms}\ bvs\ (\mathsf{Var}\ x_1\ ty_1,\mathsf{Var}\ x_2\ ty_2)}$$

$$\frac{\begin{array}{c} \mathsf{typeof}\ v_1 = \mathsf{typeof}\ v_2 \\ \mathsf{aterms}\ ((v_1,v_2)::bvs)\ (t_1,t_2) \end{array}}{\mathsf{aterms}\ bvs\ (\mathsf{Abs}\ v_1\ t_1,\mathsf{Abs}\ v_2\ t_2)}$$

Finally, we define $\mathsf{aconv}\ t_1\ t_2 \iff \mathsf{aterms}\ []\ (t_1,t_2)$ It is straightforward to show that this is an equivalence relation.

4.1.3 Substitution and Instantiation

Now on substitution, let us first deal with types. Since there are no type variable binders, type variables can simply be replaced uniformly throughout a type, given a type substitution mapping variable names to types. We define $\mathsf{tysubst}\ i\ ty$ as the type obtained by instantiating ty according to the type substitution i. We say $\mathsf{is\_instance}\ ty_0\ ty$ if $\exists i.\ ty = \mathsf{tysubst}\ i\ ty_0$.

Substitution of terms for variables and of types for type variables in terms are the most complex operations we need to deal with. Naïve substitution for variables in a term may introduce unwanted binding, for example when substituting $\mathsf{Comb}\ v_1\ t_1$ for v_2 in $\mathsf{Abs}\ v_1\ v_2$ the variable v_1 ought to remain free. The algorithm for term substitution (subst) therefore renames bound variables as required to avoid unintended capture.

The algorithm for type instantiation (inst) in terms is also complicated by this kind of problem. Consider, with $x_1 = \mathsf{Var}\ \texttt{"x"}\ (\mathsf{Tyvar}\ \texttt{"A"})$ and $x_2 = \mathsf{Var}\ \texttt{"x"}\ \mathsf{Bool}$, substitution of Bool for Tyvar "A" in $\mathsf{Abs}\ x_1\ (\mathsf{Abs}\ x_2\ x_1)$. The inner x_1 refers to the outer binder, but after a naïve substitution (which makes $x_1 = x_2$) it would incorrectly refer to the inner binder. The solution is for the type instantiation algorithm to keep track of potential shadowing as it traverses the term, and if any occurs to backtrack and rename the shadowing bound variable.

In Harrison's original formulation of HOL in HOL, the main lemma about type instantiation takes 377 lines of proof script and mixes reasoning about name clashes with the semantics of instantiation itself. To clarify our formalisation, we develop a small theory of nameless terms using de Bruijn indices, where substitution and instantiation are relatively straightforward, and shift the required effort to the task of translating to and from de Bruijn terms, which is somewhat easier than tackling capture-avoiding substitution directly. The analogous lemma about type instantiation in our formalisation is only 47 lines: the bulk of the work about name clashes appears in two lemmas totalling 166 lines about how instantiation can just as well be done on de Bruijn terms.[6]

We prove that two terms are alpha-equivalent if and only if their de Bruijn representations are equal. Using this fact, the main theorems we obtain about substitution and instantiation are that they both respect alpha-equivalence:

$$\begin{aligned}
\vdash\ & \mathsf{welltyped}\ t_1 \wedge \mathsf{welltyped}\ t_2 \wedge \mathsf{subst\_ok}\ \mathit{ilist} \wedge \mathsf{aconv}\ t_1\ t_2 \Rightarrow \\
& \mathsf{aconv}\ (\mathsf{subst}\ \mathit{ilist}\ t_1)\ (\mathsf{subst}\ \mathit{ilist}\ t_2) \\
\vdash\ & \mathsf{welltyped}\ t_1 \wedge \mathsf{welltyped}\ t_2 \wedge \mathsf{aconv}\ t_1\ t_2 \Rightarrow \\
& \mathsf{aconv}\ (\mathsf{inst}\ \mathit{tyin}\ t_1)\ (\mathsf{inst}\ \mathit{tyin}\ t_2)
\end{aligned}$$

Here subst_ok *ilist* means *ilist* is a substitution mapping variables to well-typed terms of the same type. Since we also prove that alpha-equivalent terms have the same semantics, these theorems allow us to prove soundness of the inference rules that do substitution and instantiation.

4.1.4 Theories

In our specification of HOL, every sequent carries a *theory*, which embodies information about constants and type operators and thereby allows us to support principles of definition. Formally, a theory (*thy*) consists of a signature (sigof *thy*) together with a set of axioms (axsof *thy*). The signature restricts the constants and type operators that may appear in a sequent, and the axioms provide sequents that may be derived immediately. The principles of definition introduce axioms to characterise the things that are defined.

We specify a *signature* as a pair of maps, (tysof *sig*,tmsof *sig*), assigning the defined type operator names to their arities and the defined term constant names to their types. Well-

[6] Harrison's lemma is called `SEMANTICS_INST_CORE`, and ours are `INST_CORE_dbINST`, `INST_CORE_simple_inst`, and `termsem_simple_inst`.

formed types obey the type signature:

$$\frac{}{\mathsf{type\_ok}\ \mathit{tysig}\ (\mathsf{Tyvar}\ x)} \qquad \frac{\begin{array}{c}\mathsf{lookup}\ \mathit{tysig}\ \mathit{name} = \mathsf{Some}\ (\mathsf{length}\ \mathit{args})\\ \mathsf{every}\ (\mathsf{type\_ok}\ \mathit{tysig})\ \mathit{args}\end{array}}{\mathsf{type\_ok}\ \mathit{tysig}\ (\mathsf{Tyapp}\ \mathit{name}\ \mathit{args})}$$

And well-formed terms obey both signatures:

$$\frac{\mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{sig})\ \mathit{ty}}{\mathsf{term\_ok}\ \mathit{sig}\ (\mathsf{Var}\ x\ \mathit{ty})} \qquad \frac{\begin{array}{c}\mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{sig})\ \mathit{ty}\\ \mathsf{lookup}\ (\mathsf{tmsof}\ \mathit{sig})\ \mathit{name} = \mathsf{Some}\ \mathit{ty}_0\\ \mathsf{is\_instance}\ \mathit{ty}_0\ \mathit{ty}\end{array}}{\mathsf{term\_ok}\ \mathit{sig}\ (\mathsf{Const}\ \mathit{name}\ \mathit{ty})}$$

$$\frac{\begin{array}{c}\mathsf{term\_ok}\ \mathit{sig}\ \mathit{tm}_1\\ \mathsf{term\_ok}\ \mathit{sig}\ \mathit{tm}_2\\ \mathsf{welltyped}\ (\mathsf{Comb}\ \mathit{tm}_1\ \mathit{tm}_2)\end{array}}{\mathsf{term\_ok}\ \mathit{sig}\ (\mathsf{Comb}\ \mathit{tm}_1\ \mathit{tm}_2)} \qquad \frac{\begin{array}{c}\mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{sig})\ \mathit{ty}\\ \mathsf{term\_ok}\ \mathit{sig}\ \mathit{tm}\end{array}}{\mathsf{term\_ok}\ \mathit{sig}\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ \mathit{ty})\ \mathit{tm})}$$

We include the condition that the term be well-typed in the combination case, and only allow abstractions of variables, hence we have $\vdash\ \mathsf{term\_ok}\ \mathit{sig}\ t \Rightarrow \mathsf{welltyped}\ t$.

We say a signature is standard if it maps the primitive type operators—function spaces and Booleans—and the primitive constant—equality—in the way we would expect:

$$\begin{array}{l}\mathsf{is\_std\_sig}\ \mathit{sig} \iff\\ \quad \mathsf{lookup}\ (\mathsf{tysof}\ \mathit{sig})\ \texttt{"fun"} = \mathsf{Some}\ 2\ \wedge\\ \quad \mathsf{lookup}\ (\mathsf{tysof}\ \mathit{sig})\ \texttt{"bool"} = \mathsf{Some}\ 0\ \wedge\\ \quad \mathsf{lookup}\ (\mathsf{tmsof}\ \mathit{sig})\ \texttt{"="} = \mathsf{Some}\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ \mathsf{Bool}))\end{array}$$

We have a straightforward condition for a theory to be well-formed: all its components are well-formed and the axioms are Boolean terms.

$$\begin{array}{l}\mathsf{theory\_ok}\ \mathit{thy} \iff\\ \quad (\forall \mathit{ty}.\ \mathit{ty} \in \mathsf{range}\ (\mathsf{tmsof}\ \mathit{thy}) \Rightarrow \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{thy})\ \mathit{ty})\ \wedge\\ \quad (\forall p.\ p \in \mathsf{axsof}\ \mathit{thy} \Rightarrow \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{thy})\ p \wedge p\ \mathsf{has\_type}\ \mathsf{Bool})\ \wedge\\ \quad \mathsf{is\_std\_sig}\ (\mathsf{sigof}\ \mathit{thy})\end{array}$$

(Here $\mathsf{tmsof}\ \mathit{thy}$ is shorthand for $\mathsf{tmsof}\ (\mathsf{sigof}\ \mathit{thy})$, and similarly for the types.)

4.2 Semantics

The idea behind the standard (e.g., Pitts [27]) semantics for HOL is to interpret types as non-empty sets and terms as their elements. Equality and function application and abstraction are interpreted as in set theory, and a sequent is considered true if its interpretation is the true element of the set of Booleans. Semantics for HOL in this style are a mostly straightforward example of model theory.

The most fiddly parts of the semantics arise when dealing with polymorphic constants and type operators with arguments, followed closely by issues arising from substitution and type instantiation (which we covered in Section 4.1.3). Polymorphism is especially relevant to being able to support defined constants. The approach we have taken here is to keep the treatment of constants and type operators separate from the semantics of the lambda-calculus terms, by parameterising the semantics by an interpretation, so that both pieces remain simple.

Our goal is to show how we give semantics to sequents (and their component parts) in a theory. The ultimate notion we are aiming for is validity, $(thy,h) \models c$, which says that the semantics of c is true whenever the semantics of all the h are true and the axioms of *thy* are satisfied. Validity quantifies over, and hence does not need to mention the semantic parameters that give meaning to constants and variables. But these parameters, called interpretations and valuations, are required for building the definition validity out of semantics for the component parts of a sequent.

The details of our semantic apparatus are new, compared to Harrison's work [10] on HOL semantics in a fixed context without definitions, and are inspired by the second author's specification [3] of ProofPower HOL's logic.

Semantics of Types The meaning of a HOL type is a non-empty set. Thus, we require type valuations (τ) to assign type variables to non-empty sets.

$$\textsf{is\_type\_valuation}\ \tau \iff \forall x.\ \textsf{inhabited}\ (\tau\ x)$$

The type signature (*tysig* below) says what the type operators are and how many arguments they each expect. A type assignment (δ) gives semantics to type operators; we require it to assign correct applications of type operators to non-empty sets.

$$\begin{array}{l}\textsf{is\_type\_assignment}\ tysig\ \delta \iff \\ \quad \textsf{every} \\ \quad\ (\lambda\ (name,arity). \\ \quad\quad \forall ls.\ \textsf{length}\ ls = arity \land \textsf{every inhabited}\ ls \Rightarrow \textsf{inhabited}\ (\delta\ name\ ls)) \\ \quad\ tysig\end{array}$$

The semantics of types simply maps the type valuation and type assignment through the type, as follows:

$$\begin{array}{l}\textsf{typesem}\ \delta\ \tau\ (\textsf{Tyvar}\ s) = \tau\ s \\ \textsf{typesem}\ \delta\ \tau\ (\textsf{Tyapp}\ name\ args) = \delta\ name\ (\textsf{map}\ (\textsf{typesem}\ \delta\ \tau)\ args)\end{array}$$

Observe that since the type assignment (δ) is a function in HOL, there are not necessarily any set-theoretic functions involved in the semantics of type operators.

Semantics of Terms The meaning of a HOL term is an element of the meaning of its type. Thus, a term valuation (σ) must assign each variable to an element of the meaning of its type. To speak of valid types and their meanings requires a type signature and type assignment, so the notion of a term valuation depends on them.

$$\begin{array}{l}\textsf{is\_term\_valuation}\ tysig\ \delta\ \tau\ \sigma \iff \\ \quad \forall v\ ty.\ \textsf{type\_ok}\ tysig\ ty \Rightarrow \sigma\ (v,ty) \lessdot \textsf{typesem}\ \delta\ \tau\ ty\end{array}$$

The constant signature (*tmsig* below) gives the names and types of the constants, and a term assignment (γ) must assign each constant to an element of the meaning of the appropriate type. This picture is complicated by the fact that constants may be polymorphic (that is, their types may contain type variables), so a term assignment takes not only the name of the constant but a list of meanings for the type variables, and the condition it must satisfy quantifies over type valuations. For any type valuation, the term assignment must assign the constant under that type valuation to an element of the meaning of the constant's type.

$$\begin{array}{l}
\mathsf{is\_term\_assignment}\ \mathit{tmsig}\ \delta\ \gamma \iff \\
\quad \mathsf{every} \\
\quad\ \ (\lambda\ (\mathit{name},\mathit{ty}). \\
\quad\quad \forall \tau. \\
\quad\quad\ \ \mathsf{is\_type\_valuation}\ \tau \Rightarrow \\
\quad\quad\quad \gamma\ \mathit{name}\ (\mathsf{map}\ \tau\ (\mathsf{sorted\_tyvars}\ \mathit{ty})) \lessdot \mathsf{typesem}\ \delta\ \tau\ \mathit{ty})\ \mathit{tmsig}
\end{array}$$

The semantics of terms is defined recursively as follows. For variables, we simply apply the valuation.

$$\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Var}\ x\ \mathit{ty}) = \sigma\ (x,\mathit{ty})$$

For constants, we apply the interpretation but need to match the instantiated type of the constant against its generic type, that is, the type given for the constant in the signature. This is done using the instance function, explained in the next paragraph.

$$\begin{array}{l}
\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Const}\ \mathit{name}\ \mathit{ty}) = \\
\quad \mathsf{instance}\ \mathit{tmsig}\ (\delta,\gamma)\ \mathit{name}\ \mathit{ty}\ \tau
\end{array}$$

Assuming[7] the given type is an instance of the generic type under some type substitution i, instance applies the term assignment for the constant passing the meanings of the types to which the type variables are bound under i.

$$\begin{array}{l}
\mathsf{lookup}\ \mathit{tmsig}\ \mathit{name} = \mathsf{Some}\ \mathit{ty}_0 \Rightarrow \\
\quad \mathsf{instance}\ \mathit{tmsig}\ (\delta,\gamma)\ \mathit{name}\ (\mathsf{tysubst}\ i\ \mathit{ty}_0)\ \tau = \\
\quad\quad \gamma\ \mathit{name}\ (\mathsf{map}\ (\mathsf{typesem}\ \delta\ \tau \circ \mathsf{tysubst}\ i \circ \mathsf{Tyvar})\ (\mathsf{sorted\_tyvars}\ \mathit{ty}_0))
\end{array}$$

For applications, we simply use function application at the set-theoretic level.

$$\begin{array}{l}
\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Comb}\ t_1\ t_2) = \\
\quad \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ t_1\ {}'\ (\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ t_2)
\end{array}$$

Similarly, for abstractions we create a set-theoretic function that takes an element, m, of the meaning of the type of the abstracted variable to the meaning of the body under the appropriately extended valuation.

$$\begin{array}{l}
\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ \mathit{ty})\ b) = \\
\quad \mathsf{Abstract}\ (\mathsf{typesem}\ \delta\ \tau\ \mathit{ty})\ (\mathsf{typesem}\ \delta\ \tau\ (\mathsf{typeof}\ b)) \\
\quad\quad (\lambda\, m.\ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,((x,\mathit{ty}) \mapsto m)\ \sigma)\ b)
\end{array}$$

Above, $((x,ty) \mapsto m)\ \sigma$ means the valuation that returns m when applied to (x,ty) but otherwise acts like σ.

[7] We leave unspecified the semantics of constants that are not in the signature or whose types do not match the type in the signature.

Standard Interpretations Our semantics so far makes no special treatment of HOL's primitive types and constants; indeed, we can neatly factor out the special treatment as a condition on interpretations. First, we collect the parameters for terms and types together. A pair of a type valuation and a term valuation is called a *valuation*. Similarly, a pair of a type assignment and a term assignment is called an *interpretation*.

$$\begin{array}{l}
\mathsf{is\_valuation}\ \mathit{tysig}\ \delta\ (\tau,\sigma) \iff \\
\quad \mathsf{is\_type\_valuation}\ \tau \land \mathsf{is\_term\_valuation}\ \mathit{tysig}\ \delta\ \tau\ \sigma \\
\mathsf{is\_interpretation}\ (\mathit{tysig},\mathit{tmsig})\ (\delta,\gamma) \iff \\
\quad \mathsf{is\_type\_assignment}\ \mathit{tysig}\ \delta \land \mathsf{is\_term\_assignment}\ \mathit{tmsig}\ \delta\ \gamma
\end{array}$$

We say an interpretation is *standard* if it interprets the primitive constants in the standard way; namely, functions types as set-theoretic function spaces, Booleans as the set of Booleans, and equality as set-theoretic equality (which is inherited from the meta-logic).

$$\begin{array}{l}
\mathsf{is\_std\_type\_assignment}\ \delta \iff \\
\quad (\forall\, \mathit{dom}\ \mathit{rng}.\ \delta\ \texttt{"fun"}\ [\mathit{dom};\ \mathit{rng}] = \mathsf{Funspace}\ \mathit{dom}\ \mathit{rng}) \land \\
\quad \delta\ \texttt{"bool"}\ [\,] = \mathsf{Boolset}
\end{array}$$

$$\begin{array}{l}
\mathsf{is\_std\_interpretation}\ (\delta,\gamma) \iff \\
\quad \mathsf{is\_std\_type\_assignment}\ \delta \land \\
\quad \gamma\ \mathsf{interprets}\ \texttt{"="}\ \mathsf{on}\ [\texttt{"A"}]\ \mathsf{as} \\
\quad\ (\lambda\, l. \\
\qquad \mathsf{Abstract}\ (\mathsf{head}\ l)\ (\mathsf{Funspace}\ (\mathsf{head}\ l)\ \mathsf{Boolset}) \\
\qquad\ (\lambda\, x.\ \mathsf{Abstract}\ (\mathsf{head}\ l)\ \mathsf{Boolset}\ (\lambda\, y.\ \mathsf{Boolean}\ (x = y))))
\end{array}$$

The notation used above is defined as follows:

$$\begin{array}{l}
\gamma\ \mathsf{interprets}\ \mathit{name}\ \mathsf{on}\ \mathit{args}\ \mathsf{as}\ f \iff \\
\quad \forall\, \tau.\ \mathsf{is\_type\_valuation}\ \tau \Rightarrow \gamma\ \mathit{name}\ (\mathsf{map}\ \tau\ \mathit{args}) = f\ (\mathsf{map}\ \tau\ \mathit{args})
\end{array}$$

We will only be concerned with standard interpretations.

Satisfaction We now turn to packaging the basic semantics of types and terms up and lifting it to the level of sequents. A sequent, containing both hypotheses and a conclusion, represents an implication. We say that an interpretation *satisfies* a sequent if the conclusion of the sequent is true whenever the hypotheses are (for all valuations). Precisely,

$$\begin{array}{l}
(\delta,\gamma)\ \mathsf{satisfies}\ ((\mathit{tysig},\mathit{tmsig}),h,c) \iff \\
\quad \forall\, v. \\
\qquad \mathsf{is\_valuation}\ \mathit{tysig}\ \delta\ v \land \mathsf{every}\ (\lambda\, t.\ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ v\ t = \mathsf{True})\ h \Rightarrow \\
\qquad\ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ v\ c = \mathsf{True}
\end{array}$$

We defer checking syntactic well-formedness of the sequent (for example, that c has type Bool) until the definition of validity below.

Modeling An interpretation *models* a theory if it is standard and satisfies the theory's axioms.

$$\begin{array}{l}
i\ \mathsf{models}\ \mathit{thy} \iff \\
\quad \mathsf{is\_interpretation}\ (\mathsf{sigof}\ \mathit{thy})\ i \land \mathsf{is\_std\_interpretation}\ i \land \\
\quad \forall\, p.\ p \in \mathsf{axsof}\ \mathit{thy} \Rightarrow i\ \mathsf{satisfies}\ (\mathsf{sigof}\ \mathit{thy},[\,],p)
\end{array}$$

Validity Finally, a sequent is *valid* if every model of the sequent's theory also satisfies the sequent itself.

$$\begin{array}{l}(thy,h) \models c \iff \\ \quad \mathsf{theory\_ok}\ thy \wedge \mathsf{every}\ (\mathsf{term\_ok}\ (\mathsf{sigof}\ thy))\ (c{::}h) \wedge \\ \quad \mathsf{every}\ (\lambda\, p.\ p\ \mathsf{has\_type}\ \mathsf{Bool})\ (c{::}h) \wedge \mathsf{hypset\_ok}\ h \wedge \\ \quad \forall\, i.\ i\ \mathsf{models}\ thy \Rightarrow i\ \mathsf{satisfies}\ (\mathsf{sigof}\ thy,h,c)\end{array}$$

4.3 Inference System

We have seen what HOL sequents look like and how they are to be interpreted in set theory. Now we turn to the inference system used to construct derivations of sequents.

Whereas the notion of a derivable sequent in a particular theory depends only on the abstract formulation (signature plus axioms) of theories, when it comes to extending the theory with new definitions (and other extensions) we introduce the more concrete notion of a *context*. A context is a linear sequence of theory-extending[8] updates. This formulation corresponds nicely to the actual behaviour of an implementation of the inference system (that is, a theorem prover).

We first look at the (within-theory) inference rules, then turn to the rules for theory extension (definitions and non-definitional updates). At the end of the subsection, we look at how we use theory extension to provide the initial axioms for the system.

4.3.1 Inference Rules

Recall that a sequent has the form $(thy,h)\ \texttt{|-}\ c$ where *thy* is a theory, *h* is a set of hypotheses (Boolean terms) and *c* is the conclusion (another Boolean term). For simplicity, we represent the hypothesis set as a list but endeavour to ensure its elements are distinct (up to alpha-equivalence); we write the union of two such lists as $h_1 \uplus h_2$, removal of an element *c* from *h* as $h \setminus c$, and the image of *h* under *f* as $\mathsf{map\_set}\, f\ h$.

In the HOL Light kernel, there are ten inference rules. Like Harrison, we define an abbreviation for equations since they appear frequently:

$$s == t = \mathsf{Comb}\ (\mathsf{Comb}\ (\mathsf{Equal}\ (\mathsf{typeof}\ s))\ s)\ t$$

We also use the following helper functions: $\mathsf{vfree\_in}\ v\ tm$ means *v* occurs free in *tm*, and $\mathsf{subst\_ok}\ sig\ ilist$ ensures only well-formed terms are substituted and only for variables of the same type. The rules are as follows:

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ thy \\ \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ t\end{array}}{(thy,[\,])\ \texttt{|-}\ t == t}\ \mathsf{REFL} \qquad \frac{\begin{array}{c}(thy,h_1)\ \texttt{|-}\ l == m_1 \\ (thy,h_2)\ \texttt{|-}\ m_2 == r \\ \mathsf{aconv}\ m_1\ m_2\end{array}}{(thy,h_1 \uplus h_2)\ \texttt{|-}\ l == r}\ \mathsf{TRANS}$$

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ thy \\ p\ \mathsf{has\_type}\ \mathsf{Bool} \\ \mathsf{term\_ok}\ (\mathsf{sigof}\ thy)\ p\end{array}}{(thy,[p])\ \texttt{|-}\ p}\ \mathsf{ASSUME} \qquad \frac{\begin{array}{c}(thy,h_1)\ \texttt{|-}\ p == q \\ (thy,h_2)\ \texttt{|-}\ p' \\ \mathsf{aconv}\ p\ p'\end{array}}{(thy,h_1 \uplus h_2)\ \texttt{|-}\ q}\ \mathsf{EQ\_MP}$$

[8] Our updates have a finer granularity than HOL4 theory segments or Isabelle/HOL theories, which usually include multiple updates in our sense.

$$\frac{\begin{array}{c}(\mathit{thy},h_1)\ \mathsf{|-}\ c_1\\ (\mathit{thy},h_2)\ \mathsf{|-}\ c_2\end{array}}{(\mathit{thy},h_1 \setminus c_2 \uplus h_2 \setminus c_1)\ \mathsf{|-}\ c_1\ \mathsf{==}\ c_2}\ \mathsf{DEDUCT\_ANTISYM}$$

$$\frac{\begin{array}{c}(\mathit{thy},h_1)\ \mathsf{|-}\ l_1\ \mathsf{==}\ r_1\\ (\mathit{thy},h_2)\ \mathsf{|-}\ l_2\ \mathsf{==}\ r_2\\ \mathsf{welltyped}\ (\mathsf{Comb}\ l_1\ l_2)\end{array}}{(\mathit{thy},h_1 \uplus h_2)\ \mathsf{|-}\ \mathsf{Comb}\ l_1\ l_2\ \mathsf{==}\ \mathsf{Comb}\ r_1\ r_2}\ \mathsf{MK\_COMB}$$

$$\frac{\begin{array}{c}\neg\mathsf{exists}\ (\mathsf{vfree\_in}\ (\mathsf{Var}\ x\ \mathit{ty}))\ h\\ \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{thy})\ \mathit{ty}\\ (\mathit{thy},h)\ \mathsf{|-}\ l\ \mathsf{==}\ r\end{array}}{(\mathit{thy},h)\ \mathsf{|-}\ \mathsf{Abs}\ (\mathsf{Var}\ x\ \mathit{ty})\ l\ \mathsf{==}\ \mathsf{Abs}\ (\mathsf{Var}\ x\ \mathit{ty})\ r}\ \mathsf{ABS}$$

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ \mathit{thy}\\ \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{thy})\ \mathit{ty}\\ \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{thy})\ t\end{array}}{(\mathit{thy},[\,])\ \mathsf{|-}\ \mathsf{Comb}\ (\mathsf{Abs}\ (\mathsf{Var}\ x\ \mathit{ty})\ t)\ (\mathsf{Var}\ x\ \mathit{ty})\ \mathsf{==}\ t}\ \mathsf{BETA}$$

$$\frac{\begin{array}{c}\mathsf{subst\_ok}\ (\mathsf{sigof}\ \mathit{thy})\ \mathit{ilist}\\ (\mathit{thy},h)\ \mathsf{|-}\ c\end{array}}{(\mathit{thy},\mathsf{map\_set}\ (\mathsf{subst}\ \mathit{ilist})\ h)\ \mathsf{|-}\ \mathsf{subst}\ \mathit{ilist}\ c}\ \mathsf{INST}$$

$$\frac{\begin{array}{c}\mathsf{every}\ (\mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{thy}))\ (\mathsf{map}\ \mathsf{fst}\ \mathit{tyin})\\ (\mathit{thy},h)\ \mathsf{|-}\ c\end{array}}{(\mathit{thy},\mathsf{map\_set}\ (\mathsf{inst}\ \mathit{tyin})\ h)\ \mathsf{|-}\ \mathsf{inst}\ \mathit{tyin}\ c}\ \mathsf{INST\_TYPE}$$

There is one additional way for a sequent to be provable, namely, if it is an axiom of the theory:

$$\frac{\begin{array}{c}\mathsf{theory\_ok}\ \mathit{thy}\\ c \in \mathsf{axsof}\ \mathit{thy}\end{array}}{(\mathit{thy},[\,])\ \mathsf{|-}\ c}$$

Thus the new piece of the sequent syntax, the theory, interacts with the inference system (which remains essentially as formalised by Harrison) only via the axioms of the theory and the checks that all types and terms respect the signature of the theory.

4.3.2 Theory Extension

In the previous section, we defined provability within a fixed theory. To complete the inference system, we also need mechanisms for changing the theory. At this point, we take a more concrete view of theories, which we call contexts, by considering the specific changes that can be made. For simplicity, we restrict ourselves to a linear sequence of extensions and do not allow redefinition or branching or merging of theories. This linear view is sufficient for HOL Light; a more complicated model might be necessary for theorem provers like HOL4, which supports redefinition, or Isabelle/HOL [32] which supports both redefinition and context merging.

In our linear view, each change is an update and updates come in two kinds: definitional extensions (the first two) and postulates (a.k.a. axiomatic extensions, the last three).

$$\begin{aligned}
&\mathit{update} = \\
&\quad \mathsf{ConstSpec}\ ((\mathit{string} \times \mathit{term})\ \mathit{list})\ \mathit{term} \\
&\quad \mid \mathsf{TypeDefn}\ \mathit{string}\ \mathit{term}\ \mathit{string}\ \mathit{string} \\
&\quad \mid \mathsf{NewType}\ \mathit{string}\ \mathit{num} \\
&\quad \mid \mathsf{NewConst}\ \mathit{string}\ \mathit{type} \\
&\quad \mid \mathsf{NewAxiom}\ \mathit{term}
\end{aligned}$$

We call a list of such updates a *context*. From a context (*ctxt*) we can recover a theory (thyof *ctxt*) by calculating the constants and axioms introduced by each kind of update. Postulates simply add new constants or axioms to the theory. We will specify exactly how the definitional updates extend a theory shortly.

Some basic well-formedness conditions are required. The relation *upd* updates *ctxt* specifies when *upd* is a valid extension of *ctxt*. It can be considered as specifying the conditions under which an update is allowed to be made. For example, the conditions for the postulates, which simply ensure names remain distinct and the each piece of the postulate is well-formed, are shown below.

$$\frac{\mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt})}{\mathsf{NewType}\ \mathit{name}\ \mathit{arity}\ \mathsf{updates}\ \mathit{ctxt}} \qquad \frac{\begin{array}{c}\mathit{name} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \\ \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{ctxt})\ \mathit{ty}\end{array}}{\mathsf{NewConst}\ \mathit{name}\ \mathit{ty}\ \mathsf{updates}\ \mathit{ctxt}}$$

$$\frac{\begin{array}{c}\mathit{prop}\ \mathsf{has\_type}\ \mathsf{Bool} \\ \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{ctxt})\ \mathit{prop}\end{array}}{\mathsf{NewAxiom}\ \mathit{prop}\ \mathsf{updates}\ \mathit{ctxt}}$$

A context *extends* another if it is a series of valid updates applied to the other. The initial context, supporting only equality, can be specified as an extension of the empty context:

$$\begin{aligned}
&\mathsf{init\_ctxt} = \\
&\quad [\mathsf{NewConst}\ \texttt{"="}\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ \mathsf{Bool})); \\
&\quad\ \mathsf{NewType}\ \texttt{"bool"}\ 0;\ \mathsf{NewType}\ \texttt{"fun"}\ 2]
\end{aligned}$$

We turn now to the changes introduced by definitional extensions and the conditions on making them. Let us start with the definition of new types, represented by the update TypeDefn *name pred abs rep*. Here *name* is the name of the new type and *pred* is a predicate on an existing type called the *representing type*. The intuition behind the principle of type definition is to make the new type isomorphic to the subset of the representing type carved out by *pred*, which is required to be inhabited. Type definition introduces the new type and also two constants between the new type and the representing type, asserting a bijection via the following two axioms.

```
axioms_of_upd (TypeDefn name pred abs_name rep_name) =
 (let abs_type = Tyapp name (sorted_tyvars pred) in
  let rep_type = domain (typeof pred) in
  let abs = Const abs_name (Fun rep_type abs_type) in
  let rep = Const rep_name (Fun abs_type rep_type) in
  let a = Var "a" abs_type in
  let r = Var "r" rep_type
  in
   [Comb abs (Comb rep a) == a;
    Comb pred r == (Comb rep (Comb abs r) == r)])
```

As can be seen in the construction of *abs_type* above, the new type has a type argument for each of the type variables appearing in *pred*. The type variables are sorted (according to their name) to ensure a canonical order for the new type's arguments. The two introduced axioms assert that the introduced constants, *abs* and *rep*, are inverses (when restricted to elements of the representing type that satisfy *pred*).

The main condition on making a type definition is that the new type is non-empty. This is ensured by requiring a sequent asserting that the predicate holds of some witness. Additionally the predicate itself must not contain free variables, and the new names must not already appear in the context.

$$\frac{\begin{array}{c}(\mathsf{thyof}\ \mathit{ctxt},[\,])\ \texttt{|-}\ \mathsf{Comb}\ \mathit{pred}\ \mathit{witness}\\ \mathsf{closed}\ \mathit{pred}\\ \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt})\\ \mathit{abs} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})\\ \mathit{rep} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})\\ \mathit{abs} \neq \mathit{rep}\end{array}}{\mathsf{TypeDefn}\ \mathit{name}\ \mathit{pred}\ \mathit{abs}\ \mathit{rep}\ \mathsf{updates}\ \mathit{ctxt}}$$

We will prove that context extension by type definition is sound, that is, the axioms it introduces are not contradictory, in Section 5.1.2.

Finally, let us look at the definition of new term constants via our new generalised rule for constant specification. The update ConstSpec *eqs prop*, where *eqs* are equations with variables (varsof *eqs*) on the left, signifies introduction of a new constant for each of the variables which together share the defining specification *prop*. Thus *prop* (after substituting the new constants for the variables) is the sole new axiom:

```
axioms_of_upd (ConstSpec eqs prop) =
 (let ilist = consts_for_vars eqs in [subst ilist prop])
```

The purpose of the equations is to provide witnesses that *prop* is satisfiable, so the rule takes as input a theorem concluding *prop* assuming the equations. The complete list of conditions can be seen below:

$$\frac{\begin{array}{c}(\mathsf{thyof}\ \mathit{ctxt}, \mathsf{map}\ (\lambda\ (s,t).\ \mathsf{Var}\ s\ (\mathsf{typeof}\ t) == t)\ \mathit{eqs})\ \mathsf{|\!-}\ \mathit{prop} \\ \mathsf{every}\ (\lambda\ t.\ \mathsf{closed}\ t \land \mathsf{closed\_tyvars}\ t)\ (\mathsf{map\ snd}\ \mathit{eqs}) \\ \forall x\ ty.\ \mathsf{vfree\_in}\ (\mathsf{Var}\ x\ ty)\ \mathit{prop} \Rightarrow \mathsf{member}\ (x,ty)\ (\mathsf{varsof}\ \mathit{eqs}) \\ \forall s.\ \mathsf{member}\ s\ (\mathsf{map\ fst}\ \mathit{eqs}) \Rightarrow s \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \\ \mathsf{all\_distinct}\ (\mathsf{map\ fst}\ \mathit{eqs})\end{array}}{\mathsf{ConstSpec}\ \mathit{eqs}\ \mathit{prop}\ \mathsf{updates}\ \mathit{ctxt}}$$

Here closed_tyvars t means that type variables appearing in t also appear in typeof t. The design of this principle of constant specification is explained in detail by Arthan [4]. We prove its soundness in Section 5.1.2. The rule of constant definition, which defines a new constant x to be equal to a term t, can be recovered as an instance of constant specification:

$$\mathsf{ConstDef}\ x\ t = \mathsf{ConstSpec}\ [(x,t)]\ (\mathsf{Var}\ x\ (\mathsf{typeof}\ t) == t).$$

4.4 Axioms

We need some logical connectives and quantifiers to state two of the axioms asserted in HOL Light. Since they are generally useful, it is convenient to define them first before asserting the axioms.

4.4.1 Embedding Logical Operators

The connectives of propositional logic and universal and existential quantifiers (ranging over HOL types) can be defined as constants[9] in HOL. We define a list of updates each of which defines a connective or quantifier as it is defined in HOL Light, and show, by calculating out the semantics, that they all behave as intended.

Each of the connectives and quantifiers can be defined by an equation, so we use the simple ConstDef *name term* version of the rule for constant specification. The following function extends a context with definitions of the Boolean constants[10].

```
mk_bool_ctxt ctxt =
 ConstDef "~" NotDef::ConstDef "F" FalseDef::
   ConstDef "\\/" OrDef::ConstDef "?" ExistsDef::
   ConstDef "!" ForallDef::ConstDef "==>" ImpliesDef::
   ConstDef "/\\" AndDef::ConstDef "T" TrueDef::ctxt
```

Here the definition terms are as in HOL Light, for example,

```
ForallDef =
 Abs (Var "P" (Fun (Tyvar "A") Bool))
  (Var "P" (Fun (Tyvar "A") Bool) ==
   Abs (Var "x" (Tyvar "A")) (Const "T" Bool))

FalseDef =
 Comb (Const "!" (Fun (Fun Bool Bool) Bool))
  (Abs (Var "p" Bool) (Var "p" Bool))
```

[9] In HOL theorem prover parlance these are sometimes collectively known as the theory of Booleans.

[10] The names, "\\/" and "/\\", associated with OrDef and AndDef may appear to include extra backslashes, because backslashes must be escaped in strings in HOL4. The names are intended to be textual representations of $\lor$ and $\land$.

We also specify the expected signature for constants with these names, for example,

$$\begin{array}{l}
\mathsf{is\_forall\_sig}\ \mathit{tmsig} \iff \\
\ \ \mathsf{lookup}\ \mathit{tmsig}\ \texttt{"!"} = \mathsf{Some}\ (\mathsf{Fun}\ (\mathsf{Fun}\ (\mathsf{Tyvar}\ \texttt{"A"})\ \mathsf{Bool})\ \mathsf{Bool})
\end{array}$$

and we show, by simple calculation, that $\mathsf{sigof}\ (\mathsf{mk\_bool\_ctxt}\ \mathit{ctxt})$ has the right signatures.

For the desired semantics of the Boolean constants, we refer to the connectives and quantifiers in the meta-logic (that is, for us, the logic of HOL4). For example, for implication and universal quantification, we have

$$\begin{array}{l}
\mathsf{is\_implies\_interpretation}\ \gamma \iff \gamma\ \mathsf{interprets}\ \texttt{"==>"}\ \mathsf{on}\ []\ \mathsf{as}\ (\lambda\, y.\ \mathsf{Boolrel}\ (\Rightarrow)) \\
\mathsf{is\_forall\_interpretation}\ \gamma \iff \\
\ \ \gamma\ \mathsf{interprets}\ \texttt{"!"}\ \mathsf{on}\ [\texttt{"A"}]\ \mathsf{as} \\
\ \ \ \ (\lambda\, l. \\
\ \ \ \ \ \ \mathsf{Abstract}\ (\mathsf{Funspace}\ (\mathsf{head}\ l)\ \mathsf{Boolset})\ \mathsf{Boolset} \\
\ \ \ \ \ \ \ \ (\lambda\, P.\ \mathsf{Boolean}\ (\forall x.\ x \lessdot \mathsf{head}\ l \Rightarrow P \mathbin{'} x = \mathsf{True})))
\end{array}$$

where the following helper functions interpret meta-level Booleans and relations on Booleans in our set theory:

$$\begin{array}{l}
\mathsf{Boolean}\ b = \texttt{if}\ b\ \texttt{then}\ \mathsf{True}\ \texttt{else}\ \mathsf{False} \\
\mathsf{Boolrel}\ R = \\
\ \ \mathsf{Abstract}\ \mathsf{Boolset}\ (\mathsf{Funspace}\ \mathsf{Boolset}\ \mathsf{Boolset}) \\
\ \ \ \ (\lambda\, p.\ \mathsf{Abstract}\ \mathsf{Boolset}\ \mathsf{Boolset}\ (\lambda\, q.\ \mathsf{Boolean}\ (R\ (p = \mathsf{True})\ (q = \mathsf{True}))))
\end{array}$$

The desired interpretations for all the Boolean constants are collected together as follows.

$$\begin{array}{l}
\mathsf{is\_bool\_interpretation}\ (\delta,\gamma) \iff \\
\ \ \mathsf{is\_std\_interpretation}\ (\delta,\gamma) \wedge \mathsf{is\_true\_interpretation}\ \gamma \wedge \\
\ \ \mathsf{is\_and\_interpretation}\ \gamma \wedge \mathsf{is\_implies\_interpretation}\ \gamma \wedge \\
\ \ \mathsf{is\_forall\_interpretation}\ \gamma \wedge \mathsf{is\_exists\_interpretation}\ \gamma \wedge \\
\ \ \mathsf{is\_or\_interpretation}\ \gamma \wedge \mathsf{is\_false\_interpretation}\ \gamma \wedge \\
\ \ \mathsf{is\_not\_interpretation}\ \gamma
\end{array}$$

The theorem we prove about the definitions of the Boolean constants says they have the desired semantics, that is, any interpretation that models a theory containing the definitions interprets the constants as specified by $\mathsf{is\_bool\_interpretation}$.

$$\begin{array}{l}
\vdash\ \mathsf{theory\_ok}\ (\mathsf{thyof}\ (\mathsf{mk\_bool\_ctxt}\ \mathit{ctxt})) \wedge \\
\ \ \ i\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_bool\_ctxt}\ \mathit{ctxt})) \Rightarrow \\
\ \ \ \ \ \mathsf{is\_bool\_interpretation}\ i
\end{array}$$

The semantics of a constant defined by an equation is uniquely specified, since that equation must be satisfied by any model of the definition. So, proving the theorem above is simply a matter of calculating out the semantics of the definitions of each of the constants and observing that they match the specification.

4.4.2 Statement of the Axioms

The standard library of HOL Light appeals to NewAxiom exactly three times, to assert the basic axioms of HOL that make it a classical logic and allow it to define the natural numbers. The axioms are: functional extensionality, choice, and infinity. Since the deeply-embedded syntax for the statements of the axioms is somewhat verbose, let us first look at their statements at the meta level:

- extensionality: $(\lambda x. f\, x) = f$
- choice: $P\, x \Rightarrow P\; ((\varepsilon)\; P)$
- infinity: $\exists (f : \mathit{ind} \to \mathit{ind}).\ \mathsf{ONE\_ONE}\, f \wedge \mathsf{ONTO}\, f$

While extensionality can be asserted in the initial context, the other two need additional constants to be added to the signature. For choice, we need to define implication, and to introduce the choice operator (ε above, but named "@" in the deep embedding.) For infinity, we need to introduce the type *ind* of individuals, and to define the existential quantifier[11], conjunction, and the ONE_ONE and ONTO functions.

We define context-updating functions for each of the axioms, asserting the axiom with NewAxiom after introducing new constants if necessary. These are defined below, with some of the deeply-embedded syntax abbreviated (SelectAx, InfinityAx, OntoDef, and OneOneDef).

```
mk_eta_ctxt ctxt =
 NewAxiom
  (Abs (Var "x" (Tyvar "A"))
    (Comb (Var "f" (Fun (Tyvar "A") (Tyvar "B"))) (Var "x" (Tyvar "A"))) ==
   Var "f" (Fun (Tyvar "A") (Tyvar "B")))::ctxt

mk_select_ctxt ctxt =
 NewAxiom SelectAx::NewConst "@" (Fun (Fun (Tyvar "A") Bool) (Tyvar "A"))::ctxt

mk_infinity_ctxt ctxt =
 NewAxiom InfinityAx::NewType "ind" 0::ConstDef "ONTO" OntoDef::
   ConstDef "ONE_ONE" OneOneDef::ctxt
```

4.5 Comparison to Stateless HOL

The semantics (and inference system) we have just described cleanly separates the semantics of types from the semantics of terms. It also uses an explicit theory, with an interpretation, to track which constants are defined, what their semantics are, and the axioms. By contrast, Stateless HOL [33] puts types and terms in mutual recursion and has no separate theory parameter. Stateless HOL constants carry their definitions as syntactic tags (rather than in a separate signature), and the semantics interprets those tags directly (instead of using a separate interpretation parameter). We described the semantics for Stateless HOL in the ITP paper [16] on which this paper is based. By keeping the theory and its interpretation separate in the present work, we gain the following advantages:

- The semantics of types and terms are no longer in mutual recursion, and are simpler to understand individually.

[11] Axioms do not need to universally quantify their variables: free variables act as if universally quantified because of the INST rule of inference.

- We can more naturally use functions (typesem and termsem) for the semantics, instead of mutually recursive relations.
- Specific parameters to support the axioms of choice and infinity are no longer required within the semantics. Instead, they are handled generically by the type and term interpretations, applied to `"ind"` and `"@"`.
- We can support new axioms, beyond the initial three axioms asserted in HOL Light, in the same manner as the initial ones; the initial axioms are not baked into the semantics.
- The semantics of constants defined by new specification now properly captures the abstraction intended to be provided by that rule. The semantics are not tied to the specific witnesses given when the definition is made.

  In the Stateless HOL semantics, the semantics of a defined constant needs to be given in terms of the tag on the constant which provides the witnesses. By contrast in the current setup, the witnesses are only used in proving that the rule is sound (see Section 5.1.2). Since we now have an explicit interpretation of the constants, it can vary over many possible interpretations, constrained only by the axiom produced by the definition.

The primary motivation for Stateless HOL is the ability to “undo” definitions (this is achieved by soundly allowing simultaneous distinct definitions of constants with the same name). We did not take advantage of this ability in our verified implementation built under a Stateless HOL semantics [16], since we first translated to a stateful implementation. If we wanted to support undoing definitions, Stateless HOL is still an option worth consideration, but based on our experience we would first consider adding undo support to the context-based approach.

Additional advantages of Stateless HOL are that its kernel is purely functional, and therefore, we thought, would be easier to understand theoretically. We now claim that the difficulty of verifying a stateful implementation (as we do in Section 6.2) is smaller than the difficulty of giving semantics to the mutually recursive datatypes of Stateless HOL especially when the rules of definition are included.

As an alternative approach to purely functional kernels, the OpenTheory [13] kernel achieves purity by a careful redesign of the interface to the kernel while maintaining the traditional idea of a context-extending mechanism for making definitions. The semantics of an OpenTheory article is specified via a stateful virtual machine, but the higher-level operations on the resulting OpenTheory packages are pure, and names are carefully managed, so definitions never accidentally collide or go out of date. We expect to be able to verify an OpenTheory proof checker against our HOL semantics.

5 Soundness and Consistency

We have now seen HOL’s inference system, which provides rules for proving sequents within a theory and updating that theory, and we have seen a specification of the meaning of such sequents: in particular, when they are considered valid. The main results of this section are that every sequent proved by the inference system is valid (soundness), and its corollary that some sequents cannot be proved (consistency).

Soundness holds for both the inference rules and the rules for theory extension, with the exception of NewAxiom. For an extension rule to be sound, it must put the inference system in a state whereby it continues to produce only valid sequents. We ground this idea by proving the continued existence of a model of the theory. Since we cannot prove NewAxiom sound in general, we also need to prove the three axioms used in HOL Light to set up the initial HOL context sound on a case-by-case basis.

We prove consistency for any non-axiomatic extensions of the following contexts:

$$\begin{aligned}&\mathsf{fhol\_ctxt} = \mathsf{mk\_select\_ctxt}\ (\mathsf{mk\_eta\_ctxt}\ (\mathsf{mk\_bool\_ctxt}\ \mathsf{init\_ctxt}))\\&\mathsf{hol\_ctxt} = \mathsf{mk\_infinity\_ctxt}\ \mathsf{fhol\_ctxt}\end{aligned}$$

The name of the first context above stands for “finitary HOL” since it omits the axiom of infinity. We name it separately because the consistency theorem we can prove of it has no assumptions apart from is_set_theory *mem*, which we saw in Section 3.2 can be discharged. The consistency theorem for the full hol_ctxt requires the set-theoretic axiom of infinity as an additional assumption.

5.1 Soundness

5.1.1 Inference Rules

The main soundness result for a fixed theory context is that every provable sequent is valid:

$$\vdash\ (thy,h)\ \mathsf{|-}\ c \Rightarrow (thy,h) \models c$$

Our proof of this does not differ substantially form Harrison’s, apart from our indirect treatment of substitution and instantiation via de Bruijn terms. Recall that by convention we elide on the theorem above the assumption is_set_theory *mem* and the *mem* argument passed to the validity relation ($\models$).

The result above is proved by induction on the provability relation ($\mathsf{|-}$). Thus we have a case for each of HOL’s inference rules, for example for EQ_MP:

$$\vdash\ (thy,h_1) \models p \mathrel{\mathsf{==}} q \wedge (thy,h_2) \models p' \wedge \mathsf{aconv}\ p\ p' \Rightarrow (thy,h_1 \uplus h_2) \models q$$

The proof for each case typically expands out the semantics of the sequents involved then invokes properties of the set theory. The case for the rule allowing an axiom to be proved is trivial by the definition of validity which assumes the theory is modeled.

The main work in proving soundness of the inference rules is establishing properties of the semantics of the operations used by the inference rules in constructing their conclusions. For example, for instantiation of type variables in terms, we show that instead of instantiating the term we can instantiate the valuations:

$$\begin{aligned}\vdash\ &\mathsf{term\_ok}\ (tysig,tmsig)\ tm \Rightarrow\\&\ \mathsf{termsem}\ tmsig\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{inst}\ tyin\ tm) =\\&\ \ \mathsf{termsem}\ tmsig\ (\delta,\gamma)\\&\ \ \ \ ((\lambda\, x.\ \mathsf{typesem}\ \delta\ \tau\ (\mathsf{tysubst}\ tyin\ (\mathsf{Tyvar}\ x))),\\&\ \ \ \ \ (\lambda\ (x,ty).\ \sigma\ (x,\mathsf{tysubst}\ tyin\ ty)))\ tm\end{aligned}$$

This lemma is the main support for the INST_TYPE case of the soundness theorem

$$\begin{aligned}\vdash\ &\mathsf{every}\ (\mathsf{type\_ok}\ (\mathsf{tysof}\ thy))\ (\mathsf{map}\ \mathsf{fst}\ tyin) \wedge (thy,h) \models c \Rightarrow\\&(thy,\mathsf{map\_set}\ (\mathsf{inst}\ tyin)\ h) \models \mathsf{inst}\ tyin\ c\end{aligned}$$

since we need to establish conclusions about instantiations of terms from hypotheses about the terms themselves.

5.1.2 Theory Extension

The definition of new types and constants extends the context in which sequents may be proved, in particular it changes the signature of the theory and introduces new axioms depending on the kind of definition. Intuitively, we do not want such extensions to invalidate previously proved sequents, nor do we want the definitions to introduce an inconsistency.

The first property, preserving existing sequents, is easy to prove because the only dependence of a term's semantics on the theory is via the signature of constants and type operators that appear in the term. Thus, as shown below, satisfaction is preserved as long as the context grows monotonically, that is, without changing the signature of existing constants and type operators (the syntax $f \sqsubseteq f'$ means that f' agrees with f on everything in $\mathsf{domain}\, f$).

$$\begin{array}{l} \vdash\ \mathit{tmsig} \sqsubseteq \mathit{tmsig}' \land \mathit{tysig} \sqsubseteq \mathit{tysig}' \land \mathsf{every}\ (\mathsf{term\_ok}\ (\mathit{tysig},\mathit{tmsig}))\ (c{::}h) \land \\ \quad i\ \mathsf{satisfies}\ ((\mathit{tysig},\mathit{tmsig}),h,c) \Rightarrow \\ \quad\ \ i\ \mathsf{satisfies}\ ((\mathit{tysig}',\mathit{tmsig}'),h,c) \end{array}$$

All of our context-updating rules are monotonic, since we do not allow redefinition.

The second desired property of an update, not introducing an inconsistency, is what we shall designate as making the update *sound*. To be precise, we call an update sound if any model of a theory before the update can be extended to a model of the theory with the update:

$$\begin{array}{l} \mathsf{sound\_update}\ \mathit{ctxt}\ \mathit{upd} \iff \\ \ \forall i. \\ \quad i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \Rightarrow \\ \quad\ \ \exists i'.\ \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt})\ i\ i' \land i'\ \mathsf{models}\ (\mathsf{thyof}\ (\mathit{upd}{::}\mathit{ctxt})) \end{array}$$

The constant equal_on helps formalise what we mean by one interpretation being an extension of another: they must be equal on terms and types in the previous context.

$$\begin{array}{l} \mathsf{equal\_on}\ \mathit{sig}\ i\ i' \iff \\ \ (\forall \mathit{name}.\ \mathit{name} \in \mathsf{domain}\ (\mathsf{tysof}\ \mathit{sig}) \Rightarrow \mathsf{tyaof}\ i'\ \mathit{name} = \mathsf{tyaof}\ i\ \mathit{name}) \land \\ \ \forall \mathit{name}.\ \mathit{name} \in \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{sig}) \Rightarrow \mathsf{tmaof}\ i'\ \mathit{name} = \mathsf{tmaof}\ i\ \mathit{name} \end{array}$$

It is now simply a matter of showing that each of our rules for updating the context are sound when their side conditions are met.

It is straightforward to show that NewType and NewConst are sound, because they do not introduce any new axioms. We simply need to extend the interpretation with some plausible interpretation of the data. The extended interpretation cannot be completely arbitrary, because to be a model of a theory an interpretation must be well-formed (that is, must satisfy is_interpretation). But a well-formed extension is always possible: for example mapping each new type to the set of Booleans and each new constant to an arbitrary member of the interpretation of its type (which is non-empty since the original theory is modelled). We thereby prove the following theorems.

$$\begin{array}{l} \vdash\ \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \land \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt}) \Rightarrow \\ \quad \mathsf{sound\_update}\ \mathit{ctxt}\ (\mathsf{NewType}\ \mathit{name}\ \mathit{arity}) \\ \vdash\ \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \land \mathit{name} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{ctxt})\ \mathit{ty} \Rightarrow \\ \quad \mathsf{sound\_update}\ \mathit{ctxt}\ (\mathsf{NewConst}\ \mathit{name}\ \mathit{ty}) \end{array}$$

Soundness of Type Definition A type definition, TypeDefn *name pred abs rep*, is sound if the two axioms it introduces (asserting the *abs* and *rep* constants form a bijection between the new type and the range of *pred*) can be made true by extending the original model with well-formed interpretations for the new type and two new constants. Such an extension is always possible, thus we can prove the following:

$$\begin{array}{l}
\vdash\ (\mathsf{thyof}\ \mathit{ctxt},[\,])\ \mathsf{|\!-}\ \mathsf{Comb}\ \mathit{pred}\ \mathit{witness} \land \mathsf{closed}\ \mathit{pred} \land \\
\quad \mathit{name} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt}) \land \mathit{abs} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \\
\quad \mathit{rep} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \mathit{abs} \neq \mathit{rep} \Rightarrow \\
\qquad \mathsf{sound\_update}\ \mathit{ctxt}\ (\mathsf{TypeDefn}\ \mathit{name}\ \mathit{pred}\ \mathit{abs}\ \mathit{rep})
\end{array}$$

The idea behind the proof is to interpret the new type as the subset of the representing type delineated by the semantics of *pred*, and to interpret the new constants as inclusion maps. When the *abs* constant is applied to a member of the representing type that is not in the new type, it simply picks an arbitrary element of the new type. The new type is guaranteed not to be empty by the theorem saying *pred* holds for some witness, which is required to make the type definition.

The proof of soundness of type definitions is the longest of the proofs about the rules for extension, taking around 400 lines of proof script compared to around 200 for constant specifications below and 40 for each of the other (non-definitional) updates. The reason is not that the soundness argument is significantly more complicated; rather, it is because the rule introduces many things (two axioms, two constants, and a type operator), where by contrast constant specification only introduces one axiom and introduces its constants uniformly; some work is required to calculate out the semantics of the equations in the axioms introduced by a type definition, and to ensure that each piece of the extension is well-formed.

Soundness of Constant Specification Specification of new constants, via ConstSpec *eqs prop*, introduces a single axiom, namely *prop* with its term variables replaced by the new constants, and is sound if the new constants are interpreted so as to make this axiom true. Such an interpretation is always possible when the side-conditions of the rule are met, thus we have the following:

$$\begin{array}{l}
\vdash\ \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \land \\
\quad (\mathsf{thyof}\ \mathit{ctxt},\mathsf{map}\ (\lambda\ (s,t).\ \mathsf{Var}\ s\ (\mathsf{typeof}\ t) == t)\ \mathit{eqs})\ \mathsf{|\!-}\ \mathit{prop} \land \\
\quad \mathsf{every}\ (\lambda\ t.\ \mathsf{closed}\ t \land \mathsf{closed\_tyvars}\ t)\ (\mathsf{map}\ \mathsf{snd}\ \mathit{eqs}) \land \\
\quad (\forall x\ \mathit{ty}.\ \mathsf{vfree\_in}\ (\mathsf{Var}\ x\ \mathit{ty})\ \mathit{prop} \Rightarrow \mathsf{member}\ (x,\mathit{ty})\ (\mathsf{varsof}\ \mathit{eqs})) \land \\
\quad (\forall s.\ \mathsf{member}\ s\ (\mathsf{map}\ \mathsf{fst}\ \mathit{eqs}) \Rightarrow s \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt})) \land \\
\quad \mathsf{all\_distinct}\ (\mathsf{map}\ \mathsf{fst}\ \mathit{eqs}) \Rightarrow \\
\qquad \mathsf{sound\_update}\ \mathit{ctxt}\ (\mathsf{ConstSpec}\ \mathit{eqs}\ \mathit{prop})
\end{array}$$

The idea behind the proof is to interpret the new constants as the semantics of the witness terms (that is, map snd *eqs*) given in the input theorem that concludes *prop*. This works because then substitution of the new constants for the variables in *prop* has the same effect, semantically, as discharging the hypotheses of the input theorem.

The key lemmas required are about how the term semantics interacts with the interpretation and valuation. In particular, term substitution can be moved into the valuation; and, we can ignore extensions made to the interpretation when considering the semantics of a term that does not mention the new constants, since the semantics only cares about the interpretation of things in the signature.

$$\begin{array}{l}
\vdash \mathsf{welltyped}\ \mathit{tm} \land \mathsf{subst\_ok}\ \mathit{ilist} \Rightarrow \\
\quad \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma)\ (\mathsf{subst}\ \mathit{ilist}\ \mathit{tm}) = \\
\quad\ \ \mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma \uplus \mathsf{map\_subst}\ (\mathsf{termsem}\ \mathit{tmsig}\ (\delta,\gamma)\ (\tau,\sigma))\ \mathit{ilist})\ \mathit{tm}
\end{array}$$

$$\begin{array}{l}
\vdash \mathsf{is\_std\_sig}\ (\mathit{tysig},\mathit{tmsig}) \land \mathsf{term\_ok}\ (\mathit{tysig},\mathit{tmsig})\ \mathit{tm} \land \mathsf{equal\_on}\ (\mathit{tysig},\mathit{tmsig})\ i\ i' \Rightarrow \\
\quad \mathsf{termsem}\ \mathit{tmsig}\ i'\ v\ \mathit{tm} = \mathsf{termsem}\ \mathit{tmsig}\ i\ v\ \mathit{tm}
\end{array}$$

Above, $f \uplus \mathit{ls}$ means the function that maps according to a binding in *ls* if it exists else defaults to applying f; and map_subst g *ilist* modifies the substitution *ilist*, which binds variables to terms, by applying g to all the terms.

Using these lemmas, we can reduce showing that the new axiom is satisfied to showing that *prop* is true under a valuation assigning the variables to the interpretations of the new constants. Since we interpreted the new constants as the witness terms corresponding to each variable, this then follows directly from the input theorem.

Sequences of Definitions Combining the results in this subsection, which cover soundness of all the update types except for NewAxiom, we prove that well-formed updates are sound.

$$\begin{array}{l}
\vdash \mathit{upd}\ \mathsf{updates}\ \mathit{ctxt} \land \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \land (\forall p.\ \mathit{upd} \neq \mathsf{NewAxiom}\ p) \Rightarrow \\
\quad \mathsf{sound\_update}\ \mathit{ctxt}\ \mathit{upd}
\end{array}$$

It is then a straightforward induction to show that a sequence of updates that do not introduce any axioms except via definitions preserve the existence of a model.

$$\begin{array}{l}
\vdash \mathit{ctxt}_2\ \mathsf{extends}\ \mathit{ctxt}_1 \land \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}_1) \land i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}_1) \land \\
\quad (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathit{ctxt}_2 \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathit{ctxt}_1) \Rightarrow \\
\quad\ \ \exists i'.\ \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt}_1)\ i\ i' \land i'\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}_2)
\end{array}$$

5.1.3 Axioms

We show that each of the axioms is consistent by proving: if the axiom is asserted in a theory that has a model, there is an extended interpretation that models the resulting theory. (This is the same idea as was formalised for sound_update, which we do not reuse since it only applies to a single update).

At this point, we drop our convention of eliding the is_set_theory *mem* assumption from our theorems, to make clear which of the axioms depend on which facts about the set theory.

The semantics of the axiom of extensionality is true because set-theoretic functions are extensional, and HOL functions are interpreted as set-theoretic functions. No constants are introduced, so the interpretation does not need extending.

$$\begin{array}{l}
\vdash \mathsf{is\_set\_theory}\ \mathit{mem} \Rightarrow \\
\quad \mathsf{is\_std\_sig}\ (\mathsf{sigof}\ \mathit{ctxt}) \Rightarrow \\
\quad\ \ \forall i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \Rightarrow i\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_eta\_ctxt}\ \mathit{ctxt}))
\end{array}$$

For the axiom of choice, the soundness theorem asserts existence of a model of the context extension produced by mk_select_ctxt, presuming the original context has a model, does not already define "@", and correctly interprets implication. The theorem is as follows

$$\begin{array}{l}
\vdash\ \mathsf{is\_set\_theory}\ \mathit{mem} \Rightarrow \\
\quad \texttt{"@"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \mathsf{is\_implies\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \\
\quad \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \Rightarrow \\
\quad\ \forall i. \\
\quad\quad i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \land \mathsf{is\_implies\_interpretation}\ (\mathsf{tmaof}\ i) \Rightarrow \\
\quad\quad\ \exists i'. \\
\quad\quad\quad \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt})\ i\ i' \land \\
\quad\quad\quad i'\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_select\_ctxt}\ \mathit{ctxt}))
\end{array}$$

To prove this theorem, we need to provide an interpretation of the Hilbert choice constant, "@", that satisfies the axiom: given a predicate on some type it should return an element of the type satisfying the predicate if one exists, or else an arbitrary element of the type. A suitable interpretation can be constructed using the choice operator in the meta-logic, that is, the logic of HOL4 (whose properties imply the set-theoretic axiom of choice, as shown at the end of Section 3.1).

For the axiom of infinity, the statement of the soundness theorem follows essentially the same structure as for the axiom of choice, except it uses mk_infinity_ctxt instead of mk_select_ctxt and assumes the set-theoretic axiom of infinity. Additionally, there are more assumptions about the context—that it contains certain constants, and does not already contain others—so we can define ONE_ONE and ONTO correctly. The theorem is as follows:

$$\begin{array}{l}
\vdash\ \mathsf{is\_set\_theory}\ \mathit{mem} \land (\exists \mathit{inf}.\ \mathsf{is\_infinite}\ \mathit{mem}\ \mathit{inf}) \Rightarrow \\
\quad \mathsf{theory\_ok}\ (\mathsf{thyof}\ \mathit{ctxt}) \land \texttt{"ONTO"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \\
\quad \texttt{"ONE\_ONE"} \notin \mathsf{domain}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \texttt{"ind"} \notin \mathsf{domain}\ (\mathsf{tysof}\ \mathit{ctxt}) \land \\
\quad \mathsf{is\_implies\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \mathsf{is\_and\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \\
\quad \mathsf{is\_forall\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \mathsf{is\_exists\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \land \\
\quad \mathsf{is\_not\_sig}\ (\mathsf{tmsof}\ \mathit{ctxt}) \Rightarrow \\
\quad\ \forall i. \\
\quad\quad i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \land i\ \mathsf{models}\ (\mathsf{thyof}\ \mathit{ctxt}) \land \\
\quad\quad \mathsf{is\_implies\_interpretation}\ (\mathsf{tmaof}\ i) \land \mathsf{is\_and\_interpretation}\ (\mathsf{tmaof}\ i) \land \\
\quad\quad \mathsf{is\_forall\_interpretation}\ (\mathsf{tmaof}\ i) \land \\
\quad\quad \mathsf{is\_exists\_interpretation}\ (\mathsf{tmaof}\ i) \land \mathsf{is\_not\_interpretation}\ (\mathsf{tmaof}\ i) \Rightarrow \\
\quad\quad\ \exists i'. \\
\quad\quad\quad \mathsf{equal\_on}\ (\mathsf{sigof}\ \mathit{ctxt})\ i\ i' \land \\
\quad\quad\quad i'\ \mathsf{models}\ (\mathsf{thyof}\ (\mathsf{mk\_infinity\_ctxt}\ \mathit{ctxt}))
\end{array}$$

To prove this theorem, we need to provide an interpretation of the type of individuals in such a way that the axiom of infinity is satisfied. We pick the infinite set *inf* whose existence is assumed. Then proving the theorem is simply a matter of calculating out the semantics and observing that the axiom holds because the set is infinite.

Having proved the soundness of each axiom separately, we can put them together within a single context and prove soundness for it and all its extensions (as long as they do not introduce further axioms). Recall the definitions of the contexts that assert the axioms:

$$\begin{array}{l}
\mathsf{fhol\_ctxt} = \mathsf{mk\_select\_ctxt}\ (\mathsf{mk\_eta\_ctxt}\ (\mathsf{mk\_bool\_ctxt}\ \mathsf{init\_ctxt})) \\
\mathsf{hol\_ctxt} = \mathsf{mk\_infinity\_ctxt}\ \mathsf{fhol\_ctxt}
\end{array}$$

We obtain the following results by combining the soundness theorems for the three axioms presented in this section with the result from Section 5.1.2 about theory extensions that do not add any further new axioms.

$$\begin{array}{l}\vdash \mathsf{is\_set\_theory}\ mem \Rightarrow\\ \quad \forall ctxt.\\ \quad\ \ ctxt\ \mathsf{extends}\ \mathsf{fhol\_ctxt} \wedge\\ \quad\ \ (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{fhol\_ctxt}) \Rightarrow\\ \quad\ \ \ \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \wedge \exists i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt)\end{array}$$

$$\begin{array}{l}\vdash \mathsf{is\_set\_theory}\ mem \wedge (\exists inf.\ \mathsf{is\_infinite}\ mem\ inf) \Rightarrow\\ \quad \forall ctxt.\\ \quad\ \ ctxt\ \mathsf{extends}\ \mathsf{hol\_ctxt} \wedge\\ \quad\ \ (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ ctxt \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{hol\_ctxt}) \Rightarrow\\ \quad\ \ \ \mathsf{theory\_ok}\ (\mathsf{thyof}\ ctxt) \wedge \exists i.\ i\ \mathsf{models}\ (\mathsf{thyof}\ ctxt)\end{array}$$

The order in which the extensions are made ensure that the signature and interpretation assumptions of each of the soundness theorems for the axioms is satisfied.

5.2 Consistency

We have seen that the inference system for HOL (as implemented by HOL Light) is sound in that every sequent it derives is semantically valid. As a corollary, we can show that there are some sequents which cannot be derived (since some sequents are not valid). Our strategy for proving this syntactic notion of consistency is to use the fact, sometimes called semantic consistency, that every theory produced by the inference system has a model (as proved in the previous section).

We define a consistent theory as one for which there are sequents one of which can be derived and the other which cannot. In fact, we choose particular sequents for this purpose, an equation of equal variables and an equation of potentially different variables:

$$\begin{array}{l}\mathsf{consistent\_theory}\ thy \iff\\ \quad (thy,[\,])\ \mathsf{|-}\ \mathsf{Var}\ \texttt{"x"}\ \mathsf{Bool}\ \mathsf{==}\ \mathsf{Var}\ \texttt{"x"}\ \mathsf{Bool} \wedge\\ \quad \neg((thy,[\,])\ \mathsf{|-}\ \mathsf{Var}\ \texttt{"x"}\ \mathsf{Bool}\ \mathsf{==}\ \mathsf{Var}\ \texttt{"y"}\ \mathsf{Bool})\end{array}$$

Any theory with a model is consistent, as the following lemma demonstrates.

$$\begin{array}{l}\vdash \mathsf{is\_set\_theory}\ mem \Rightarrow\\ \quad \forall thy.\ \mathsf{theory\_ok}\ thy \wedge (\exists i.\ i\ \mathsf{models}\ thy) \Rightarrow \mathsf{consistent\_theory}\ thy\end{array}$$

We prove the lemma by appeal to soundness: if the sequent equating two different variables were derivable, it would be valid (by soundness), and since the theory has a model it would be true in that model under every valuation. But it is not true under the valuation that sends Var "x" Bool to True and Var "y" Bool to False, so it cannot be derivable. As for the sequent equating equal variables, it is derivable as an instance of the REFL rule.

Combining the lemma above with the results in the previous section, the following consistency theorems follow immediately.

$$\begin{array}{l} \vdash \mathsf{is\_set\_theory}\ \mathit{mem} \Rightarrow \\ \quad \forall \mathit{ctxt}. \\ \quad\ \ \mathit{ctxt}\ \mathsf{extends}\ \mathsf{fhol\_ctxt} \wedge \\ \quad\ \ (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathit{ctxt} \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{fhol\_ctxt}) \Rightarrow \\ \quad\ \ \ \mathsf{consistent\_theory}\ (\mathsf{thyof}\ \mathit{ctxt}) \end{array}$$

$$\begin{array}{l} \vdash \mathsf{is\_set\_theory}\ \mathit{mem} \wedge (\exists \mathit{inf}.\ \mathsf{is\_infinite}\ \mathit{mem}\ \mathit{inf}) \Rightarrow \\ \quad \forall \mathit{ctxt}. \\ \quad\ \ \mathit{ctxt}\ \mathsf{extends}\ \mathsf{hol\_ctxt} \wedge \\ \quad\ \ (\forall p.\ \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathit{ctxt} \Rightarrow \mathsf{member}\ (\mathsf{NewAxiom}\ p)\ \mathsf{hol\_ctxt}) \Rightarrow \\ \quad\ \ \ \mathsf{consistent\_theory}\ (\mathsf{thyof}\ \mathit{ctxt}) \end{array}$$

The free variable *mem* in these theorems only appears in the assumptions, but those assumptions are of course necessary since we appealed to soundness, which depends on *mem* via the *i* models *thy* relation (and ultimately the semantics of terms and types).

6 Verifying the Kernel in CakeML

We have now seen that the HOL inference system, as specified by the provability relation (|-) and the rules for updating the context, is sound and consistent. Next, we turn our attention to producing a verified theorem prover implementing this sound inference system. Recall that our strategy is to produce the implementation in two steps: first, we define a theorem-prover kernel as recursive functions in a state-exception monad within the logic of HOL4, then we use an automated proof-producing technique to translate these recursive functions into code in the CakeML programming language. A preliminary description of this strategy can be found in our short paper [26] at ITP 2013.

6.1 The Monadic Functions

In implementations of HOL theorem provers, including the original OCaml implementation of HOL Light, the kernel module defines a datatype of theorems whose values correspond to the provable sequents of the HOL inference system. Our theorem datatype is defined with a single constructor as follows.

$$\mathit{thm}\ =\ \mathsf{Sequent}\ (\mathit{term\ list})\ \mathit{term}$$

In the implementation, the theory part of a sequent is not included on the theorem values, being instead embodied by the state of the theorem prover and the history of computations that led it into that state. The state of the theorem prover consists of the following four values, which will be implemented as references in CakeML.

$$\begin{array}{l} \mathit{state} = \\ \ \langle\ \mathsf{the\_type\_constants} : ((\mathit{string} \times \mathit{num})\ \mathit{list}); \\ \quad \mathsf{the\_term\_constants} : ((\mathit{string} \times \mathit{type})\ \mathit{list}); \\ \quad \mathsf{the\_axioms} : (\mathit{thm\ list}); \\ \quad \mathsf{the\_context} : (\mathit{update\ list})\ \rangle \end{array}$$

The first three fields of the state correspond to references found in the original OCaml implementation of HOL Light. The fourth field represents the current context. As we saw when describing the inference system, the type constants, term constants, and axioms can all be calculated from the context, so it is redundant to include them all in the state. For efficiency, and faithfulness to the original, we do not discard the other three references in favour of the context; rather, we think of the context as a "ghost" variable, which we will prove is always consistent with the rest of the state but which is not actually required for the implementation. For clarity, we leave it in the implementation rather than as an existentially quantified variable on our correctness theorems.

The monadic functions only raise two kinds of exceptions: failure with an error message, and, in the implementation of instantiation of type variables within a term, a "clash" exception for backtracking when unintended variable capture is detected.

$$\mathit{exn} \;=\; \mathsf{Fail}\ \mathit{string} \mid \mathsf{Clash}\ \mathit{term}$$

With our models of state and exceptions in place, we define our state-exception monad ($\alpha\ M$) as follows.

$$\begin{aligned}\alpha\ \mathit{result} \;&=\; \mathsf{HolRes}\ \alpha \mid \mathsf{HolErr}\ \mathit{exn}\\ \alpha\ M = \mathit{state} \;&\to\; \alpha\ \mathit{result} \times \mathit{state}\end{aligned}$$

We define monadic bind as would be expected (that is, we either compute with the result or propagate the exception, and propagate the state in both cases), and make use of HOL4's support for `do` notation (as found also in Haskell) for composition of monadic binds.

Let us look now at how the monadic functions are defined. For example, here is the function implementing the ASSUME rule of inference.

```
ASSUME tm =
 do
  ty ← type_of tm;
  bty ← mk_type ("bool",[]);
  if ty = bty then return (Sequent [tm] tm)
  else failwith "ASSUME: not a proposition"
 od
```

Here type_of *tm* computes the type of *tm* (failing on ill-typed terms), mk_type (*name*,*args*) constructs a type operator (failing if the number of arguments does not match the current signature in the state's type constants reference), and failwith *msg* raises the Fail *msg* exception. We define a function like the above for each of the rules of inference and of definition, as well as all the requisite helper functions (like type_of), following the original OCaml implementation closely.

The monadic functions operate over the *thm* datatype, and re-use the underlying terms and types from the inference system. What we prove about them is that every computation preserves invariants on the values being computed. Importantly, the invariant on theorem values states that they are provable within the HOL inference system. The full list of invariants we use, each of which is parametrised by the current context, is given below.

$$\begin{array}{l}
\mathsf{TYPE}\ \mathit{ctxt}\ \mathit{ty} \iff \mathsf{type\_ok}\ (\mathsf{tysof}\ \mathit{ctxt})\ \mathit{ty} \\
\mathsf{TERM}\ \mathit{ctxt}\ \mathit{tm} \iff \mathsf{term\_ok}\ (\mathsf{sigof}\ \mathit{ctxt})\ \mathit{tm} \\
\\
\mathsf{THM}\ \mathit{ctxt}\ (\mathsf{Sequent}\ h\ c) \iff (\mathsf{thyof}\ \mathit{ctxt},h)\ \texttt{|-}\ c \\
\\
\mathsf{STATE}\ \mathit{ctxt}\ \mathit{state} \iff \\
\quad \mathit{ctxt} = \mathit{state}.\mathsf{the\_context} \wedge \mathit{ctxt}\ \mathsf{extends}\ \mathsf{init\_ctxt} \wedge \\
\quad \mathit{state}.\mathsf{the\_type\_constants} = \mathsf{type\_list}\ \mathit{ctxt} \wedge \\
\quad \mathit{state}.\mathsf{the\_term\_constants} = \mathsf{const\_list}\ \mathit{ctxt}
\end{array}$$

The STATE invariant requires the current context to be a valid extension (of init_ctxt). Thus preserving the STATE invariant entails only making valid updates to the context.

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 successfully, the return value is a good theorem:

$$\begin{array}{l}
\vdash\ \mathsf{TERM}\ \mathit{ctxt}\ \mathit{tm} \wedge \mathsf{STATE}\ \mathit{ctxt}\ s \wedge \mathsf{ASSUME}\ \mathit{tm}\ s = (\mathit{res},s') \Rightarrow \\
\quad s' = s \wedge \forall \mathit{th}.\ \mathit{res} = \mathsf{HolRes}\ \mathit{th} \Rightarrow \mathsf{THM}\ \mathit{ctxt}\ \mathit{th}
\end{array}$$

This theorem is proved by stepping through the definition of ASSUME, and, at the crucial point where a Sequent value is created, observing that the assumptions for the ASSUME clause of the provability (|-) relation are satisfied, so the THM invariant holds.

We prove a similar theorem for each function in the kernel, showing that they implement the HOL inference system correctly. As another example, consider the rule for constant specification, which may update the state. We prove that the new state still satisfies our invariants, as does the returned theorem.

$$\begin{array}{l}
\vdash\ \mathsf{THM}\ \mathit{ctxt}\ \mathit{th} \wedge \mathsf{STATE}\ \mathit{ctxt}\ s \Rightarrow \\
\quad \texttt{case}\ \mathsf{new\_specification}\ \mathit{th}\ s\ \texttt{of} \\
\quad\ (\mathsf{HolRes}\ \mathit{th},s') \Rightarrow \exists \mathit{upd}.\ \mathsf{THM}\ (\mathit{upd}::\mathit{ctxt})\ \mathit{th} \wedge \mathsf{STATE}\ (\mathit{upd}::\mathit{ctxt})\ s' \\
\quad \texttt{|}\ (\mathsf{HolErr}\ \mathit{exn},s') \Rightarrow s' = s
\end{array}$$

6.2 Producing CakeML

The monadic functions constitute a *shallow embedding* of a theorem-prover-kernel implementation, because they are functions whose semantics is given implicitly by HOL (as implemented by HOL4): consider the fact that the ASSUME function has type $\mathit{term} \rightarrow \mathit{thm}\ M$. In this section, we turn to production of a *deep embedding* of the same functions, with semantics given explicitly as the operational semantics of the CakeML programming language. In the deep embedding, the ASSUME function is a piece of syntax; its type is *dec*, that is, a CakeML declaration. Furthermore, since CakeML supports references and exceptions directly, the functions no longer need to be monadic.

We produce the deep embeddings from our shallow embeddings automatically, using the proof-producing translation technique described in Myreen and Owens [25]. The result of translation is syntax and a certificate theorem. For example, for the monadic ASSUME function, we obtain the following syntax (shown as abbreviated CakeML abstract syntax):

```
⊢ nth_element 99 ml_hol_kernel_decls =
    Dlet (Pvar "assume")
     (Fun "v3"
       (Let (Some "v2") (App Opapp [Var (Short "type_of"); Var (Short "v3")])
         (Let (Some "v1")
           (App Opapp [Var (Short "mk_type"); Con None [... ... ; ... ]])
           (If (App Equality [Var (... ... ); ... ... ])
             (Con (Some (Short "Sequent"))
               [Con (... ... ) [... ... ; ... ]; Var (... ... )])
             (Raise
               (Con (Some (Short "Fail"))
                 [Lit (StrLit "ASSUME: not a proposition")])))))))
```

The same code pretty-printed in CakeML concrete syntax:

```
fun assume v3 =
  let val v2 = type_of v3
      val v1 = mk_type ("bool",[])
  in
    if (v2 = v1)
    then (Sequent([v3],v3))
    else (raise Fail("ASSUME: not a proposition"))
  end
```

The meaning of the declaration above is specified by the operational semantics of CakeML. The certificate theorem produced by translation connects evaluation of the declaration to the monadic function ASSUME:

```
⊢ DeclAssum (Some "Kernel") ml_hol_kernel_decls env tys ⇒
    EvalM env (Var (Short "assume"))
     ((PURE TERM_TYPE ─M→ HOL_MONAD THM_TYPE) ASSUME)
```

Here, DeclAssum *mn decls env tys* means that (*env*,*tys*) is the environment (of declared values and types) obtained by evaluating the list *decls* of declarations within a module *mn*; and, EvalM *env exp P* means that evaluation of the expression *exp* in environment *env* terminates and produces a result satisfying the refinement invariant P. In the theorem above, the refinement invariant takes the form $(A \xrightarrow{M} B)\,f$, which specifies a closure that, when applied to an input value satisfying A, terminates and produces an output value, which will satisfy B, according to the monadic function f.

To understand the guarantee provided by the certificate theorem, let us unpack the refinement invariant a little further. The thing to remember is that the refinement invariants specify the relationship between certain HOL terms (values in the shallow embedding) and deeply-embedded CakeML values. For example, the following fact demonstrates how the THM_TYPE invariant relates values of type *thm* to CakeML values (Conv denotes a

CakeML value made from application of a CakeML constructor):

$$\begin{array}{l}\vdash \mathsf{LIST\_TYPE}\ \mathsf{TERM\_TYPE}\ []\ v_1 \wedge \mathsf{TERM\_TYPE}\ tm\ v_2 \Rightarrow \\ \quad \mathsf{THM\_TYPE}\ (\mathsf{Sequent}\ []\ tm) \\ \quad\ (\mathsf{Conv}\ (\mathsf{Some}\ (\texttt{"Sequent"}, \mathsf{TypeId}\ (\mathsf{Long}\ \texttt{"Kernel"}\ \texttt{"thm"})))\ [v_1;\ v_2])\end{array}$$

Here, we have CakeML values, v_1 and v_2, that are related by the refinement invariants for terms (and lists of terms) to the empty list and a term *tm*, and they are used to put together a CakeML value that is related to the theorem Sequent [] *tm*. The operators PURE and HOL_MONAD extend these refinement invariants to also relate the CakeML store (that is, the contents of references) and result (normal termination or raised exception) to the corresponding parts of the state-exception monad. (PURE lifts non-monadic values into the monad while HOL_MONAD works directly on a monadic value.)

Finally, $(A \xrightarrow{M} B)\,f$ is the refinement invariant for monadic functions as explained earlier. Thus using the certificate theorem for ASSUME we can prove in CakeML's operational semantics that the return value of any successful application of the deeply-embedded `assume` function will be related by the THM_TYPE invariant to the corresponding application of the monadic ASSUME function. And, as we saw in the previous section, the result of applying the monadic ASSUME function is related by the THM invariant to a sequent in the sound inference system (|-).

We have certificate theorems like this for every function in the CakeML implementation of the HOL Light kernel. It is on that basis that we make the claim that our kernel only produces theorem values that correspond to true sequents according to the semantics of HOL.

7 Related Work

The themes of the work described in this paper are formalising (and mechanising) the syntax and semantics of logic, and verifying (or producing verified) theorem-prover implementations. We factor our review of prior work in these areas by the particular logic under consideration.

Higher-Order Logic There has been prior work on producing a formal (mechanised) specification of the semantics of HOL. The documentation for HOL4 includes a description, originally due to Pitts [27], of the semantics of HOL. However, this description is given in the traditional semiformal style of the mathematical logic literature. In the early 1990s, the development of the ProofPower logical kernel was informed by a formal specification in ProofPower-HOL of the proof development system, including a formalisation of the HOL language, logic and semantics. However, no formal proofs were carried out. The present work found several errors in the ProofPower formalisation of the semantics (all now corrected [3]). As discussed in Section 1, Harrison's work [10] on a proof in HOL Light of the consistency of the HOL logical kernel without definitions formed the starting point for the present work (initially, [26, 16]).

Krauss and Schropp [15] have formalised a translation from HOL to set theory, automatically producing proofs in Isabelle/ZF [32]. Their motivation was to revive Isabelle/ZF by importing Isabelle/HOL proofs into it, but this task necessitates formalising an interpretation of HOL in set theory for which they use the standard approach (as we did) sending types to non-empty sets and terms to elements of their types. Although the Isabelle/HOL logic is slightly more complicated than the HOL we described, due to type classes and overloading, they remove the extra features in a preprocessing phase. They handle type definitions and (equational) constant definitions by making equivalent definitions in Isabelle/ZF, which supports Isabelle's general facility for definitions.

Dependent Type Theory Barras [5] has formalised a reduced version of the calculus of inductive constructions, the logic used by the Coq proof assistant [6], giving it a semantics in set theory and formalising a soundness proof in Coq itself. The approach is modular, and Wang and Barras [31] have extended the framework and applied it to the calculus of constructions plus an abstract equational theory.

Anand and Rahli [1] have formalised the semantics of NuPRL's type theory and proved soundness for its sequent calculus. The mechanisation is carried out within Coq. The semantics of NuPRL is rather more complex than of HOL, so its formalisation is impressive; on the other hand, they do not yet go so far as producing a verified implementation, but allude to the interesting possibility of producing it directly from the proof term for the soundness of the inference system.

First-Order Logic Myreen and Davis [24] 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 of the delicate foundational territory necessitating our $\mathsf{is\_set\_theory}\ \mathit{mem}$ and $\exists \mathit{inf}.\ \mathsf{is\_infinite}\ \mathit{mem}\ \mathit{inf}$ assumptions.

Other noteworthy prover verifications include a simple first-order tableau prover by Ridge and Margetson [28] and a SAT solver algorithm with many modern optimizations by Marić [19].

8 Conclusion

A theorem prover is a computer program whose correctness can be understood at many levels. At the highest level, we focus solely on the logic, which should be consistent, and the particular inference system, which should be sound. At the next level down, we consider whether the inference system is implemented correctly, that is, whether the (abstract) computations performed by the theorem prover correspond to construction of derivations in the inference system. The remaining levels all concern correct implementation of those computations at more concrete levels of abstraction, from a high-level programming language down to hardware. In this paper we have dealt with correctness of a theorem prover for higher-order logic (HOL) spanning all the levels between consistency of the logic itself and implementation in a high-level programming language (CakeML), within a single mechanically-checked formalisation.

We have gone further than the previous work in this vein in two directions: the coverage of the logic formalised and the concreteness of the theorem-prover implementation verified. Our formalisation, with full support for making extensions to the context, now covers all of HOL as it is implemented by real theorem provers. Our implementation is a deeply-embedded program verified against the operational semantics of a realistic programming language. On both fronts, however, the end of the line has not been reached: one might like to verify a more sophisticated approach to contexts (such as the one implemented by

Isabelle [32]), and a more concrete implementation (for example, in machine code). Additionally, we have so far only verified the kernel of a theorem prover, and would like to extend the result to a complete theorem prover, which means formally validating the LCF design [22] by reasoning about the guarantees provided by a protected (abstract) type in CakeML.

In constructing a formal specification of the semantics of HOL that is suitable both for proofs about the logic and inference system, and for proofs about implementing that inference system, we faced several design decisions. The main theme of the lessons learned is to value explicitness and separation of concerns. Using an explicit theory context gives a simpler semantics than that of Stateless HOL, as was discussed in Section 4.5. Similarly, specifying the axioms of set theory with an explicit membership relation yields a development that is easier to work with than the theoretically equivalent approach based solely on a cardinality assumption. And by specifying the set theory separately from defining an instance of it, we obtain a conservative approach using isolated assumptions about free variables rather than global axiomatic extensions. On a smaller scale, our choice to factor our reasoning about substitution and instantiation, which is complicated with name-carrying terms, through a separate small theory about de Bruijn terms led to simplifications.

Continuing the self-verification project initiated by Harrison [10] for HOL Light, our formalisation of HOL is conducted within HOL itself. It is common to cite Gödel's incompleteness theorems as making it meaningless or impossible for a logical system to be used to prove its own properties. However, this objection applies only to proofs of consistency. In the present work, our primary concern is with soundness, and what we have done is analogous to proving the soundness of first-order logic within a first-order formalisation of ZF set theory, which as standard logic textbooks (e.g., Mendelson [20]) show is well-known and uncontroversial.

The theorem prover we use for our mechanisation (HOL4) is distinct from the verified implementation we produce (in CakeML of a kernel based on HOL Light's). The implementation of HOL4 is not itself verified; one might wonder whether we gain anything by trusting one theorem prover to verify another of a similar (in fact lesser) complexity. While we acknowledge this objection, our reply is that HOL4 can be seen merely as a tool to help us organise our development if we consider the fact that our proofs can be exported from HOL4 (for example, via OpenTheory [13]) for independent checking. Thus although something ultimately needs to be trusted, we do not require it to be HOL4. A second reply is that the exercise of developing the formalisation leads us to clarify our thinking about the systems under consideration, and, on the implementation side, uncovers the kinds of bugs that are likely to occur in theorem provers in practice.

In future work, one might like to import our verified kernel into HOL Light for independent checking, or, more interestingly, to replay the proof in the verified CakeML implementation itself. Checking a correctness proof about its own concrete implementation would be closer to true self-verification than any theorem prover has yet achieved. Of course, such a check does not rule out the possibility that the theorem prover is not sound, because it might be broken in such a way that it fails to detect an incorrect correctness proof. But we would have high confidence in a theorem prover with such an ability (alongside other evidence for soundness, like a readable, concise implementation) and would expect the practical facility required for self-verification to be useful for tackling more ambitious software verification challenges.

Acknowledgements We thank Mike Gordon for helpful comments on drafts of this paper. We are grateful to John Harrison and Freek Wiedijk for providing inspiration for this project.

References

1. Abhishek Anand and Vincent Rahli. Towards a formally verified proof assistant. In Klein and Gamboa [14], pages 27–44.
2. Peter B Andrews. *An introduction to mathematical logic and type theory*, volume 27. Springer, 2002.
3. Rob Arthan. HOL formalised: Semantics. `http://www.lemma-one.com/ProofPower/specs/spc002.pdf`.
4. Rob Arthan. HOL constant definition done right. In Klein and Gamboa [14], pages 531–536.
5. Bruno Barras. Sets in Coq, Coq in sets. *J. Formalized Reasoning*, 3(1):29–48, 2010.
6. Yves Bertot. A short presentation of Coq. In Mohamed et al. [23], pages 12–16.
7. Alonzo Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.
8. Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte fr Mathematik und Physik*, 38(1):173–198, 1931.
9. Mike Gordon. From LCF to HOL: a short history. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 169–186. The MIT Press, 2000.
10. John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2006.
11. John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009.
12. Leon Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15:81–91, 1950.
13. Joe Hurd. The opentheory standard theory library. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.
14. Gerwin Klein and Ruben Gamboa, editors. *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*. Springer, 2014.
15. Alexander Krauss and Andreas Schropp. A mechanized translation from higher-order logic to set theory. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 323–338. Springer, 2010.
16. Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In Klein and Gamboa [14], pages 308–324.
17. Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: A verified implementation of ML. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 179–192. ACM, 2014.
18. Casimir Kuratowski. Sur la notion de l'ordre dans la théorie des ensembles. *Fundamenta Mathematicae*, 2(1):161–171, 1921.
19. Filip Marić. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *Theor. Comput. Sci.*, 411(50):4333–4356, 2010.
20. Elliott Mendelson. *Introduction to mathematical logic. 5th ed.* Boca Raton, FL: CRC Press, 5th ed. edition, 2009.
21. Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.
22. Robin Milner. LCF: A way of doing proofs with a machine. In Jirí Becvár, editor, *Mathematical Foundations of Computer Science 1979, Proceedings, 8th Symposium, Olomouc, Czechoslovakia, September 3-7, 1979*, volume 74 of *Lecture Notes in Computer Science*, pages 146–159. Springer, 1979.
23. Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors. *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*. Springer, 2008.
24. Magnus O. Myreen and Jared Davis. The reflective Milawa theorem prover is sound - (down to the machine code that runs it). In Klein and Gamboa [14], pages 421–436.
25. Magnus O. Myreen and Scott Owens. Proof-producing translation of higher-order logic into pure and stateful ML. *J. Funct. Program.*, 24(2-3):284–315, 2014.

26. Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of HOL Light. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 490–495. Springer, 2013.
27. Andrew M. Pitts. *The HOL System: Logic*, 3rd edition. `http://hol.sourceforge.net/documentation.html`.
28. Tom Ridge and James Margetson. A mechanically verified, sound and complete theorem prover for first-order logic. In Joe Hurd and Thomas F. Melham, editors, *Theorem Proving in Higher Order Logics, 18th International Conference, TPHOLs 2005, Oxford, UK, August 22-25, 2005, Proceedings*, volume 3603 of *Lecture Notes in Computer Science*, pages 294–309. Springer, 2005.
29. Konrad Slind and Michael Norrish. A brief overview of HOL4. In Mohamed et al. [23], pages 28–32.
30. Robert L. Vaught. *Set theory: An introduction*. Basel: Birkhäuser, 2nd edition, 1994.
31. Qian Wang and Bruno Barras. Semantics of intensional type theory extended with decidable equational theories. In Simona Ronchi Della Rocca, editor, *Computer Science Logic 2013 (CSL 2013), CSL 2013, September 2-5, 2013, Torino, Italy*, volume 23 of *LIPIcs*, pages 653–667. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013.
32. Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle framework. In Mohamed et al. [23], pages 33–38.
33. Freek Wiedijk. Stateless HOL. In Tom Hirschowitz, editor, *Proceedings Types for Proofs and Programs, Revised Selected Papers, TYPES 2009, Aussois, France, 12-15th May 2009.*, volume 53 of *EPTCS*, pages 47–61, 2009.