An Interpretation of Isabelle/HOL in HOL Light

Sean McLaughlin

Carnegie Mellon University

Abstract. We define an interpretation of the Isabelle/HOL logic in HOL Light and its metalanguage, OCaml. Some aspects of the Isabelle logic are not representable directly in the HOL Light object logic. The interpretation thus takes the form of a set of elaboration rules, where features of the Isabelle logic that cannot be represented directly are elaborated to functors in OCaml. We demonstrate the effectiveness of the interpretation via an implementation, translating a significant part of the Isabelle standard library into HOL Light.

1 Introduction

The vast advances in computer technology of the last century facilitated the construction of computer programs that could check logical proofs in full detail. These programs, called *proof assistants* or *interactive theorem provers*, were an extension of, and improvement upon, formal logical reasoning in the spirit of Russell and Whitehead [38] and Landau [19]. Such proof assistants, from the pioneer DeBruijn's Automath [5] to its modern counterparts (*e.g.*, Coq [4], HOL4 [11], HOL Light [17], Isabelle [28], Nuprl [7], PVS [27]), seek fully foundational proofs of deep mathematical and scientific problems. While the technical challenges of such developments can be significant, many important theorems have been fully checked in these systems. Some recent examples are the Four Color Theorem [10], the Prime Number Theorem [2], and the Jordan Curve Theorem [13].

Unfortunately, each system has its own library of theorems. The extensive effort involved in constructing a proof in one system must be duplicated to prove the theorem in another. For instance, the three examples cited above are all constructed in different proof assistants, and as of this writing, none have been ported to another system. Indeed, little infrastructure exists to support the sharing of proofs between proof assistants. This dissonance is a serious concern for large verification efforts. For example, the Flyspeck Project [12] seeks to formally prove the Kepler Conjecture [14] in HOL Light. Recently, Nipkow verified an important algorithm in the proof using Isabelle/HOL [25]. Researchers elsewhere are working on other parts of the project using the Coq proof assistant. For

the Kepler Conjecture to exist as a single HOL Light theorem, there must be a way to import the Isabelle and Coq developments.

This paper describes a mechanism and provides an implementation for interpreting formulas and proofs of Isabelle/HOL in HOL Light. The interpretation is interesting because Isabelle/HOL supports features not found in ordinary higher order logic. These include *axiomatic type classes* and *constant overloading*. We therefore do not attempt a direct translation into HOL Light logic. Instead, we elaborate Isabelle's types, terms, and proofs to functors in the HOL Light metalanguage, Objective Caml (OCaml) [36]. We demonstrate the effectiveness of this interpretation via an implementation, translating a significant portion of the Isabelle/HOL standard library into HOL Light, including many proofs which rely on overloading and axiomatic type classes.

A note on fonts: Isabelle text appears in sans serif font. OCaml keywords appear in **bold**. OCaml identifiers, which are also HOL Light inference rules and types, appear in SMALL CAPITAL LETTERS. Metafunctions, such as `tv` which returns the free type variables of a term, are in `typewriter face`.

2 Related work

There are two different approaches to the sharing of formal theories. In one view, which we will call the *trusting* view, we interpret the logic of one proof assistant in another, prove (on paper) some semantic properties of the translation, verify that the axioms of the source system hold in the target interpretation, and finally accept the interpreted formulas that correspond to theorems in the source logic as theorems in the target logic. No translation of proof objects is attempted or, indeed, is necessary. The user of such a translation supposes the soundness of the source theorem prover.

In the other view, which we call the *skeptical* view, a given proof assistant is the final arbiter of correct reasoning. Relying on other systems, which are possibly unsound, is undesirable. Indeed, the very *raison d'être* of the given assistant is to distill the essential axioms and to build rich mathematical structures from these axioms alone. Trusting a large body of computer code would be anathema. To import theorems we check their proofs. There is no need to rely upon a model theory because the proof theory of the target system will guarantee the correctness of the translation.

Examples of the trusting view include [8, 23], which import formulas of HOL and Isabelle/HOL, respectively, into Nuprl. Felty and Howe [9] show how the connection described in [8] can be used in a larger example. The skeptical outlook can be seen in [24, 35, 20]. Obua and Skalberg [26] describe an analogue to our work in the opposite direction, translating HOL Light proofs into Isabelle/HOL. There is also some related work involving general translation infrastructure, which we discuss in section 7.

3 Isabelle/HOL in HOL Light

HOL Light is an interactive prover in the LCF style [33] based on Church's simple theory of types [6]. Isabelle is a logical framework for defining logics [29]. The most well-developed instantiation is the interpretation of higher-order logic, Isabelle/HOL. In addition, Isabelle is extended with axiomatic type classes and constant overloading [37][1]. Though the logics are very similar, these additional features make the Isabelle logic more expressive, in the sense that a single theorem in Isabelle/HOL corresponds to a set of theorems in HOL Light. We thus appeal to the metalanguage to support type classes and overloading.

Note that in the following exposition we give a syntax for Isabelle that is convenient for our purposes. We do not present the actual concrete syntax of Isabelle. We take similar liberties with the OCaml syntax.

3.1 Type class example

Reasoning with type classes can be seen as a generalization of polymorphism. In a logic with polymorphism we avoid constructing similar theorems at different types and instead simply instantiate a polymorphic theorem at any type. In a logic with type classes we do the same, except that we also assert axiomatic properties of the type. An example is the

[1] Isabelle also includes a *locale* mechanism that extends the genericity of its reasoning capabilities [3]. Locales are eliminated in the construction of proof terms in Isabelle, however, and thus we needn't account for them in our interpretation, where we work directly with the proof terms.

class of partial orders.

$$\begin{aligned}&\mathsf{axclass}\ \mathsf{order}\ [] = \\ &\qquad [\leq\!: \alpha \to \alpha \to \mathsf{bool}] \\ &\qquad [\mathsf{refl\ is}\ x : \alpha :: \mathsf{order} \leq x, \\ &\qquad\ \mathsf{antisym\ is}\ x \leq y \wedge y \leq x \supset x = y, \\ &\qquad\ \mathsf{trans\ is}\ x \leq y \wedge y \leq z \supset x \leq z]\end{aligned}$$

An axclass declaration consists of an Isabelle name (order), a list of ancestor classes, a list of constants that should be defined at that type, and a list of named axioms that hold on the universe of α and the constants. The syntax $x : \alpha :: \mathsf{order}$ means x is a variable of type α where α is an *instance* of the class order. A type is an instance of a class if the constants of the class are defined on the type and satisfy the class axioms. More generally, $x : \alpha :: [c_1, \ldots, c_n]$ means that x is a variable of type α where α is an instance of all of $c_1, \ldots, c_n$. The collection $[c_1, \ldots, c_n]$ is called a *sort*. In this case the class has no ancestors.

We can now prove theorems with free type variables $\alpha :: \mathsf{order}$. For instance, we can prove the theorem called order_eq_refl :

$$\forall x : \alpha :: \mathsf{order}.\ x = y \supset x \leq y.$$

The proof term makes use of the class axioms.

To use theorems involving type classes, we must prove that concrete types are instances of the class. We prove that such a concrete type satisfies the class axioms, and then we instantiate the free type variables.

$$\begin{aligned}&\mathsf{instance\ real} :: [\mathsf{order}] \ldots \\ &\mathsf{instance\ nat} :: [\mathsf{order}] \ldots \\ &\forall x : \mathsf{real}.\ x = y \supset x \leq y \\ &\forall x : \mathsf{nat}.\ x = y \supset x \leq y\end{aligned}$$

where the ... stand for an Isabelle proof that the real, nat types satisfy the axioms. Then we may instantiate order_eq_refl twice to get the specific theorems Since HOL Light does not have such capabilities, we use the OCaml module system to emulate this behavior. The class order corresponds to an OCaml signature, while the Isabelle types real and nat

correspond to modules *containing* the HOL Light types REAL and NAT.

```
signature ORDER =
sig
  val α : TYPE
  val ≤: TERM
  val REFL : THM
  val ANTISYM : THM
  val TRANS : THM
end
```

```
module REAL =
struct
  let α = REAL
  let ≤: REAL_LE
  let REFL = REAL_LE_REFL
  let ANTISYM = REAL_LE_ANTISYM
  let TRANS = REAL_LE_TRANS
end
```

It is understood that REAL_LE is a predefined HOL Light constant, and that REAL_LE_REFL, *etc.* are predefined theorems. We can assume a similar definiton of a NAT module (though the HOL Light name of the type of natural numbers is NUM).

The Isabelle proof of order_eq_refl becomes a functor, encapsulating the reasoning involved.

```
functor ORDER_EQ_REFL(A : ORDER) =
struct
  let THM = (proof involving A. ≤, A.REFL, etc.)
end
```

To instantiate the proof, we apply the functor to a module containing a type and the necessary constants and axioms on that type. Functor application "replays" the proof on the new type. The applications followed by projections evaluate to the HOL Light theorems

$$\text{ORDER\_EQ\_REFL}(\text{REAL}).\text{THM} = \forall x : \text{REAL}.\ x = y \supset x \leq y,$$
$$\text{ORDER\_EQ\_REFL}(\text{NAT}).\text{THM} = \forall x : \text{NUM}.\ x = y \supset x \leq y.$$

4 Elaboration

The translation from Isabelle/HOL to HOL Light is a set of syntax-directed elaboration rules. Many of the cases are routine. We give some illustrative cases here. A complete list, along with a complete abstract syntax for Isabelle/HOL and HOL Light, can be found in the the appendix.

Our judgments have the form $Ctx \vdash X \rightsquigarrow Y$, understood to mean "$X$ elaborates to Y in context Ctx," where it is understood that Ctx and X are input arguments and Y is an output argument. We define such a judgment for each syntactic class of Isabelle/HOL. In the following sections we explain the various contexts of the judgments and their elaboration rules.

Note that we introduce judgments in their order of importance, and thus some judgments of lesser interest will be used before they are defined. The curious reader may consult the appendix for the full definitions of the judgments.

4.1 The module system

While in fact Isabelle/HOL theorems are elaborated to OCaml functors, for clarity of presentation we are taking some liberties with the notation. In particular, we allow projections from functor applications. Such functors are called *applicative* in the literature. This is in contrast to the *generative* functors of OCaml [16]. Because our modules save no state, such projections are unproblematic and have the same semantics in both views. We can easily convert these functors to a generative form accepted by OCaml by inventing a new module identifier M (which does not bind anything seen so far in the environment), binding the functor application to that name, and projecting directly from M. *E.g.* ORDER_EQ_REFL(A).THM becomes

module X = ORDER_EQ_REFL(A)
X.THM

For clarity, we also use the keyword **signature** instead of OCaml's **module type**.

We assume that before elaboration begins, the following signatures are defined. These represent HOL Light types, terms and theorems.

```
signature TYPE =
sig
  val TYPE : TYPE
end
```

```
signature TERM =
sig
  val TERM : TERM
end
```

```
signature THM =
sig
  val THM : THM
end
```

Name mapping There is some amount of bookkeeping involved in mapping Isabelle identifiers to their OCaml counterparts. The details are

not interesting. We assume the existence of a function $\ulcorner x \urcorner$ mapping the Isabelle identifier x to its counterpart. For example, $\ulcorner$order_eq_refl$\urcorner$ = ORDER_EQ_REFL. In some cases $\ulcorner x \urcorner$ requires additional arguments. We note such places explicitly. An example is type constructors that are indexed in HOL Light by the sorts of their arguments.

An Isabelle theory is a sequence of declarations that introduce new names into a global environment. To ease the notational burden of frequently inventing new names, we extend the definition of $\ulcorner x \urcorner$ to generate a fresh name for the HOL Light counterpart of a declaration x, that will thereafter be returned by $\ulcorner x \urcorner$. For instance, when we elaborate order_eq_refl, $\ulcorner$order_eq_refl$\urcorner$ generates the name ORDER_EQ_REFL and from the point of that declaration on, $\ulcorner$order_eq_refl$\urcorner$ = ORDER_EQ_REFL.

4.2 Contexts

The elaborator manages a number of distinct contexts during elaboration.

$\boldsymbol{\Delta}$ is a map from Isabelle type variables to OCaml functor arguments. In the example above, Δ would consist of the single pair $\langle \alpha, \mathrm{A} \rangle$, where α is the type variable from the Isabelle theorem order_eq_refl and A is the argument of the OCaml functor ORDER_EQ_REFL. The type variable α is elaborated to A.TYPE in the ORDER_EQ_REFL functor. We often look up a block of type variables in Δ. Thus, $\Delta(\alpha_1, \ldots, \alpha_k) = (T_1, \ldots, T_k)$ means $\Delta(\alpha_1) = T_1, \ldots, \Delta(\alpha_k) = T_k$.

$\boldsymbol{\Gamma}$ maps Isabelle term variables to types, and Isabelle proof variables to Isabelle terms.

$\boldsymbol{\Sigma}$ As we wish to make no reference to a global data structure, Σ simply maintains the state of the elaboration process, mapping Isabelle declarations to their previously elaborated OCaml functors.

4.3 Functions

We assume the existence of a function `tv` which returns the free type variables in a term with their sorts, and a predicate $(A_1, \ldots, A_n)$ `fresh` indicating that the names $A_1, \ldots, A_n$ are new.

4.4 Classes

The elaboration of type classes are one of the the most interesting parts of the translator. The Isabelle abstract syntax for a class is

$$\begin{aligned}\mathsf{axclass}\ c < [c_1, \ldots, c_k]\ &= [con_1, \ldots, con_l], \\ &\quad [name_1\ \mathsf{is}\ axm_1, \ldots, name_m\ \mathsf{is}\ axm_m]\end{aligned}$$

where c is a new Isabelle class identifier, the c_i are previously defined type classes, the con_i are constants, and the axm_i are formulas representing type class axioms referred to by $name_i$. The evidence for a type τ being an instance of the class c is a proof that τ has constants of all classes $c_1, \ldots, c_k$ in addition to $con_1, \ldots, con_l$ and that those constants satisfy the axioms of $c_1, \ldots, c_k$ in addition to $axm_1, \ldots, axm_m$.

$\Sigma \vdash \mathsf{axclass}\ c < [c_1, \ldots, c_k] = [con_1, \ldots, con_l],$
$\quad [name_1\ \mathsf{is}\ axm_1, \ldots, name_m\ \mathsf{is}\ axm_m] \rightsquigarrow$
signature $\ulcorner c \urcorner =$
sig
include $\ulcorner c_1 \urcorner$ ... **include** $\ulcorner c_k \urcorner$
val $\ulcorner con_1 \urcorner$: TERM ... **val** $\ulcorner con_l \urcorner$: TERM
val $\ulcorner name_1 \urcorner$: THM ... **val** $\ulcorner name_m \urcorner$: THM
end, Σ

The **include** statements textually replace the $\ulcorner c_i \urcorner$ with their definitions, thus capturing the semantics of the Isabelle class hierarchy. Note how the formulas axm_i are totally ignored. Here we make note of the phase distinction between *elaborating* the Isabelle theories and *using* the elaborated theorems. During the elaboration stage, the inability to specify the form of the axioms of a class is unproblematic. Both the signatures and the concrete types are created directly from Isabelle declarations, and, barring a bug in Isabelle, the concrete theorems match the declared class axioms. After the elaboration, however, when attempting to use these theorems with new types not defined by Isabelle, it is the HOL Light user's responsibility to ensure the well-formedness of the theorems she supplies. If the theorem supplied to a user-created module is not well-formed, a run-time error occurs during the functor application.

4.5 Instance

In Isabelle, instance declarations allow theorems with type variables of a class to be instantiated with concrete classes. The abstract syntax is

$$\mathsf{instance}\ \tau :: (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_n, \sigma_n \rangle)\ c = \langle [con_1, \ldots, con_k], p \rangle$$

which means that type constructor τ, when given arguments of sort σ_i is an instance of class c, where con_i are the constants required by c, and p is a proof that the axioms of c are satisfied by the type $\tau(\alpha_1, \ldots, \alpha_n)$, where α_i has sort σ_i.

$$\frac{\begin{array}{l}\texttt{fresh}(A_1, \ldots, A_k) \quad \Delta = (\langle \alpha_1, A_1 \rangle, \ldots, \langle \alpha_k, A_k \rangle) \\ \Delta, \cdot \vdash con_1 \rightsquigarrow c_1 \quad \ldots \quad \Delta, \cdot \vdash con_n \rightsquigarrow c_n \\ \Delta, \cdot \vdash p \rightsquigarrow thm \quad thm = (thm_1 \wedge \ldots \wedge thm_m) \\ \left.\begin{array}{l}\textbf{signature}\ \ulcorner c \urcorner = \\ \textbf{sig} \\ \quad \textbf{val}\ \ulcorner con_1 \urcorner\ :\ \textsc{term}\ \ldots\ \textbf{val}\ \ulcorner con_l \urcorner\ :\ \textsc{term} \\ \quad \textbf{val}\ axm_1\ :\ \textsc{thm}\ \ldots\ \textbf{val}\ axm_m\ :\ \textsc{thm} \\ \textbf{end}\end{array}\right\} \in \Sigma \\ A = \ulcorner \tau(\sigma_i, \ldots, \sigma_n) \urcorner\end{array}}{\begin{array}{l}\Sigma \vdash \mathsf{instance}\ \tau :: (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_n, \sigma_n \rangle)\ c = \langle [con_1, \ldots, con_k], p \rangle \rightsquigarrow \\ \quad \textbf{functor}\ A(A_1 : \ulcorner \sigma_1 \urcorner) \ldots (A_k : \ulcorner \sigma_n \urcorner) = \\ \quad \textbf{struct} \\ \quad\ \textbf{let}\ \ulcorner con_1 \urcorner = c_1\ \ldots\ \textbf{let}\ \ulcorner con_k \urcorner = c_k \\ \quad\ \textbf{let}\ axm_1 = thm_1\ \ldots\ \textbf{let}\ axm_m = thm_m \\ \quad \textbf{end}, \Sigma\end{array}}$$

To elaborate an instance, we begin by creating Δ from the free sorted type variables α. Then we translate the required constants and the proof of the axioms. Finally, we look up the definition of the class to get the signature identifiers for constants and axioms.

Note that the name of the generated functor depends on the sorts of the instance declaration. This is inevitable. Consider the Isabelle product type $\alpha \times \beta$. The generated functor for the type definition (see the appendix for details) would be

$$\textbf{functor}\ \ulcorner \times \urcorner(A_1 : \textsc{Type})(A_2 : \textsc{Type}) : \textsc{Type} = \ldots$$

In Isabelle we can declare

$$\mathsf{instance} \times :: (\langle \alpha_1, \mathsf{order} \rangle, \langle \alpha_2, \mathsf{order} \rangle)\ \mathsf{order}\ = \ldots$$

where we use the lexicographic ordering from α_1 and α_2. The elaboration of this instance declaration becomes

$$\textbf{functor}\ ^\ulcorner\times^\urcorner(A_1 : \textsc{Order})(A_2 : \textsc{Order}) : \textsc{Order} = \ldots.$$

If $^\ulcorner\times^\urcorner$ were not indexed by sorts, the first functor would be shadowed by the second, and thus inaccessible. Since not all types are instances of ORDER, in such a situation it would be impossible to create product types of unordered types.

4.6 Theorems

Theorems in Isabelle are a name together with a formula and a proof. The abstract syntax is $\mathsf{Thm}(id, t, p)$. Because in general the free type variables have nontrivial sorts, we abstract the type variables into functor arguments of the appropriate signature.

$$\frac{\begin{array}{ll}\mathtt{tv}(t) = (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_k, \sigma_k \rangle) & \mathtt{fresh}(A_1, \ldots, A_k) \\ \Delta = (\langle \alpha_1, A_1 \rangle, \ldots, \langle \alpha_k, A_k \rangle) & \Delta, \cdot \vdash p \rightsquigarrow thm\end{array}}{\begin{array}{l}\Sigma \vdash \mathsf{Thm}(id, t, p) \rightsquigarrow \\ \quad \textbf{functor}\ ^\ulcorner id^\urcorner(A_1 : {}^\ulcorner\sigma_1{}^\urcorner) \ldots (A_k : {}^\ulcorner\sigma_k{}^\urcorner) : \textsc{Thm} = \\ \quad \textbf{struct} \\ \quad\ \ \textbf{val}\ \textsc{Thm} = thm \\ \quad \textbf{end}, \Sigma\end{array}}$$

4.7 Types

As both Isabelle/HOL and HOL Light have their basis in classical higher order logic, translating terms and proofs is straightforward. We include the rules in the appendix for completeness. Translating types has one complication, which is that a type variable corresponds to a functor argument instead of a specific HOL Light type. In order to make type translation syntax directed (in the sense that to translate a type constructor, it is enough to translate its arguments) we elaborate types to module variables. When the types themselves are needed, we simply project the TYPE component.

$$\frac{\Delta(\alpha) = A}{\Delta \vdash \alpha :: \sigma \rightsquigarrow A}$$

$$\frac{\Delta \vdash \tau_1 \rightsquigarrow A_1 \quad \ldots \quad \Delta \vdash \tau_k \rightsquigarrow A_k}{\Delta \vdash con(\tau_1, \ldots, \tau_k) \rightsquigarrow \ulcorner con \urcorner (A_1) \ldots (A_k)}$$

This completes our overview of the elaboration rules. A complete list can be found in the appendix.

5 Name mapping

The name map $\ulcorner x \urcorner$ from Isabelle identifiers to HOL Light identifiers plays an important role in many of the elaboration judgments. Some declarations, *e.g.*, axioms, refer to HOL Light identifiers that are assumed already to be mapped before the translation begins. In order for the user to extend the translator without modifying the source code, we include a simple specification language that allows a user to include these identifier maps in $\ulcorner x \urcorner$. In addition, the systems have a number of types and constants in common. The language allows a user to specify mappings between them. This avoids creating a second copy of the type or constant in HOL Light. For instance, both Isabelle/HOL and HOL Light have a type of natural numbers nat and NUM respectively. They are both similarly constructed from an axiom of infinity. Instead of having two separate developments of the natural numbers in HOL Light, we can map one to the other with the *typemap* declaration, followed by a number of *thmmap* declarations mapping the peano axioms.

typemap : nat $\rightsquigarrow$ NUM
thmmap : Suc_not_zero $\rightsquigarrow$ NOT_SUC ...

The complete language definition and description can be found at [1].

6 Implementation

While we feel the elaboration makes novel use of the OCaml module system, the real contribution of this work is not theoretical, but practical. We have a working implementation of the elaboration rules written in Standard ML [22]. We have used the implementation to translate approximately 2000 theorems of the Isabelle/HOL standard library. While this is only about a third of more than 6000 theorems in the library, we foresee no difficulties in translating the rest. Already included in the first 2000 are all the difficulties of type classes, type definitions, and instances. Most of the effort of translation goes into carefully defining the theory in the given specification language and in proving the necessary HOL Light

theorems corresponding to an Isabelle theory. We expect the rest of the library to be completed in the near future. The translated libraries and the SML source code of the elaborator are available on the web at [1]. Users can download our libraries for experimentation.

7 Future Work

7.1 More libraries, more logics

The most natural direction for this work is to translate more libraries. Indeed, we would like to import the rest of the standard library[2] and continue on to Avigad's prime number theorem. We also intend to use the implementation to translate the Isabelle portions of the Flyspeck project. Nipkow's algorithm verification relies on a reflection mechanism, whereby an algorithm is verified formally, code is extracted, and the code is run directly. There is no analogue to this mechanism in HOL Light, so this too presents a challenge for future work.

We would also like to perform similar translations for more diverse deductive systems. An interpretation of Coq will be essential for Flyspeck, though the logics are so different that this will be a significant challenge.

7.2 Formalizing the translation in LF

As effective as it is in practice, the interpretation given is unsatisfying in a number of ways. To begin, the elaboration of classes includes no information about what formula the declared axioms should prove. This is no oversight, as it would require OCaml to allow dependently typed terms. We therefore do not discover an error when using the functors until run-time. Given the length of time required to load a library into OCaml, this is a significant disadvantage. The problem occurs both in the elaboration phase when the HOL Light programmer must supply translations of the Isabelle axioms and in the usage phase when instantiating functors at concrete types. (*cf.* Section 4.4). HOL Light inference rule calls fail for many reasons, for instance, when the supplied theorem does not have exactly the right form. It would be much better to catch such errors at compile-time.

Moreover, the interpretation given has no obvious metatheoretic properties. For one, there is not an obvious relationship between the formula of an imported proof to the translation of the initial Isabelle/HOL formula.

[2] What I call the standard library is the contents of the theories included in Main

We would hope, for example, that if a proof p of t elaborates to p', then t elaborates to `concl`(p'). Another such property is completeness. We believe that the translation is total in the sense that every Isabelle/HOL theorem could be translated to an OCaml functor that, when run on any "correctly" implemented type modules, would yield the desired theorem instance. A formal proof of these facts, though, would involve reasoning about the operational semantics of the OCaml module system in addition to the logics involved. While we may convince ourselves on paper that our reasoning is correct, the full details of the proof would be overwhelming.

These concerns can be addressed by modeling the translation in LF [15], via the Twelf [30] implementation. Using the Twelf methodology, and that generally espoused by the Logosphere Project [31], we could formalize the Isabelle/HOL and HOL Light logics and give an operational semantics to a subset of the OCaml module language. We could then hope to prove theorems about the interpretation. An example of this kind of formalization, from HOL to Nuprl, can be found in [32]. We intend to follow a similar path with our Isabelle/HOL HOL Light translation and to extend the work of [32] by generating the OCaml code directly from Twelf.

8 Conclusion

The usefulness and importance of sharing libraries between proof assistants is abundantly clear. As a step in this direction, we presented an interpretation of the Isabelle/HOL logic in HOL Light and demonstrated its effectiveness through an implementation that produces executable OCaml functors. These functors construct HOL Light proofs. A significant part of the Isabelle/HOL standard library was translated in this way. In addition we provide a specification language that allows the translator to be extended easily to new theories. We hope that our work will be useful to the formal mathematics community.

9 Acknowledgments

We would like to thank Frank Pfenning for his advice and support throughout this work. John Reynolds, Tom Murphy, and William Lovas gave helpful suggestions. Thanks also to the members of the Isabelle mailing list who patiently answered many questions on the minutiae of Isabelle, and to John Harrison for editing a draft of the paper. This work was supported by NSF grant CCR-ITR-0325808.

References

1. http://www.cs.cmu.edu/~seanmcl/projects/logosphere/isabelle-holl.
2. J. Avigad, K. Donnelly, D. Gray, and P. Raff. A formally verified proof of the prime number theorem. To appear in the ACM Transactions on Computational Logic.
3. C. Ballarin. Locales and locale expressions in Isabelle/Isar. In S. B. *et al*, editor, *Types for Proofs and Programs: International Workshop*, 2003.
4. Y. Bertot and P. Castéran. *CoqÁrt: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.
5. N. G. d. Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays in Combinatory Logic, Lambda Calculus, and Formalism*, pages 589–606. Academic Press, 1980.
6. A. Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.
7. R. Constable. *Implementing Mathematics with The Nuprl Proof Development System*. Prentice-Hall, 1986.
8. D. J. Howe. Importing mathematics from HOL into Nuprl. In J. Von Wright, J. Grundy, and J. Harrison, editors, *Ninth International Conference on Theorem Proving in Higher Order Logics TPHOL*, volume LNCS 1125, pages 267–282, Turku, Finland, 1996. Springer Verlag.
9. A. P. Felty and D. J. Howe. Hybrid interactive theorem proving using Nuprl and HOL. In *Fourteenth International Conference on Automated Deduction*, pages 351–365. Springer-Verlag Lecture Notes in Computer Science, 1997.
10. G. Gonthier. A computer-checked proof of the four colour theorem. Available on the Web via `http://research.microsoft.com/~gonthier/`, 2005.
11. M. J. C. Gordon and T. F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
12. T. Hales. The Flyspeck Project fact sheet. Project description available at `http://www.math.pitt.edu/~thales/flyspeck/`, 2005.
13. T. Hales. The Jordan Curve Theorem in HOL Light. Source code available at `http://www.math.pitt.edu/~thales/`, 2005.
14. T. C. Hales. A proof of the the Kepler conjecture. *Annals of Mathematics*, 162:1065–1185, 2005.
15. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proceedings of the Second Annual Symposium on Logic in Computer Science*, pages 194–204, Ithaca, NY, 1987. IEEE Computer Society Press.
16. R. Harper and B. C. Pierce. Design issues in advanced module systems. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*. MIT Press, 2005.
17. J. Harrison. HOL Light: A tutorial introduction. In Srivas and Camilleri [34], pages 265–269.
18. W. A. Howard. The formulae-as-types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1998.
19. E. Landau. *Grundlagen der Analysis*. Leipzig, 1930. English translation by F. Steinhardt: 'Foundations of analysis: the arithmetic of whole, rational, irrational, and complex numbers. A supplement to textbooks on the differential and integral calculus', published by Chelsea; 3rd edition 1966.

20. S. McLaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining CVC Lite and HOL Light. In A. Armando and A. Cimatti, editors, *Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, volume 144, pages 43–51, 2005.
21. T. F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.
22. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.
23. P. Naumov. Importing Isabelle formal mathematics into Nuprl. Technical Report TR99-1734, Cornell University, 26, 1999.
24. P. Naumov, M.-O. Stehr, and J. Meseguer. The HOL/NuPRL proof translator - a practical approach to formal interoperability. In *Theorem Proving in Higher Order Logics, 14th International Conference*, volume 2152 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
25. T. Nipkow, G. Bauer, and P. Schultz. Flyspeck I: Tame Graphs. Technical report, Institut für Informatik, TU München, Jan. 2006.
26. S. Obua and S. Skalberg. Importing HOL into Isabelle/HOL. submitted, 2006.
27. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752, Saratoga, NY, 1992. Springer-Verlag.
28. L. C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. With contributions by Tobias Nipkow.
29. F. Pfenning. Logical frameworks. In *Handbook of Automated Reasoning*, pages 1063–1147. MIT Press, 2001.
30. F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framweork for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction*, pages 202–206, 1999.
31. F. Pfenning, C. Schürmann, M. Kohlhase, N. Shankar, and S. Owre. The Logosphere Project. Project description available at `http://www.logosphere.org`, 2005.
32. C. Schürmann and M.-O. Stehr. An Executable Formalization of the HOL/NuPRL Connection in Twelf. In *11th International Conference on Logic for Programming Artificial Intelligence and Reasoning*, 2005.
33. D. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Annotated version of a 1969 manuscript.
34. M. Srivas and A. Camilleri, editors. *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
35. M.-O. Stehr, P. Naumov, and J. Meseguer. A proof-theoretic approach to the HOL-NuPRL connection with applications to proof-translation. In *WADT/CoFI'01*, 2001.
36. P. Weis and X. Leroy. *Le langage Caml*. InterEditions, 1993. See also the CAML Web page: `http://pauillac.inria.fr/caml/`.
37. M. Wenzel. Type Classes and Overloading in Higher-Order Logic. In E. Gunter and A. Felty, editors, *TPHOLs '97*, pages 307–322, Murray Hill, New Jersey, 1997.
38. A. N. Whitehead and B. Russell. *Principia Mathematica (3 vols)*. Cambridge University Press, 1910.

A HOL Light

A.1 Abstract syntax

The object logic of HOL Light consists of types, terms, and proofs, the last of which is represented, *á la* LCF, as inference rules defined by Ocaml functions. Types and terms are those of the simply typed lambda calculus. We also introduce new constant and type definitions.

$$
\begin{array}{rll}
\tau & ::= \alpha \mid con(\tau_1, \ldots, \tau_k) & \text{(type variables and constructors)} \\
\theta & ::= \cdot \mid (\tau, \tau'), \theta & \text{(type instantiation environments)} \\
t & ::= x \mid c \mid t\ t' \mid \lambda x : \tau.\ t & \text{(terms)} \\
thm & ::= & \text{(theorems)} \\
& \mid \textsc{assume}\ t & \text{(initial sequent)} \\
& \mid \textsc{spec}\ t\ thm & \text{(universal instantiation)} \\
& \mid \textsc{mp}\ thm\ thm' & \text{(modus ponens)} \\
& \mid \textsc{gen}\ t\ thm & \text{(universal generalization)} \\
& \mid \textsc{disch}\ t\ thm & \text{(deduction rule)} \\
& \mid \textsc{inst\_type}\ \theta\ thm & \text{(type instantiation)} \\
& \mid \textsc{new\_definition}\ t & \text{(new constants def)} \\
& \mid \textsc{new\_type\_definition}\ (name, name)\ thm & \text{(new type def)}
\end{array}
$$

Examples of type constructors are BOOL with no arguments, and FUN with two.

In the clause for new constant definitions, t is a formula of the form $c = t'$ where c is a fresh name and t' is the defining term.

In the clause for new type definitions, the supplied names denote new constants which will be defined as mapping to and from the new type universe. There are analogous names in the Isabelle type definitions. Indeed, the mechanics are the same. For the details of new type definitions in HOL Light and Isabelle, see [21].

A.2 Operational semantics

For completeness, we give a brief operational semantics of the inference rules. We use the symbol $\vdash$ to indicate theorem-hood in HOL Light. So $A_1, \ldots, A_k \vdash A$ means that A is provable from the A_i using the inference rules. We use the symbol $x \rhd y$ to indicate that x evaluates (in OCaml)

to y. Thus, if x is an OCaml identifier which evaluates to the HOL Light theorem $ctx \vdash p$, we write $x \rhd ctx \vdash p$.

$$\frac{x \rhd t}{\text{ASSUME } x \rhd t \vdash t}$$

x is an OCaml identifier whose value is the HOL Light term t. This rule states that under the assumption t we can prove t. It is assumed that the (HOL Light) type of t is BOOL.

$$\frac{x_1 \rhd t \quad x_2 \rhd ctx \vdash \forall x.p}{\text{SPEC } x_1\ x_2 \rhd ctx \vdash [t/x]p}$$

Here the notation $[t/x]p$ means a renaming meta-substitution of the free occurrences of x with t in the theorem p.

$$\frac{x_1 \rhd ctx_1 \vdash a \supset b \quad x_2 \rhd ctx_2 \vdash a}{\text{MP } x_1\ x_2 \rhd ctx_1 \cup ctx_2 \vdash b}$$

This is the rule of implication elimination, or *modus ponens*.

$$\frac{x_1 \rhd v \quad x_2 \rhd ctx \vdash p \quad v \notin \mathtt{fv}(ctx)}{\text{GEN } x_1\ x_2 \rhd ctx \vdash \forall v.\ p}$$

This is the universal generalization rule. If v is a variable not free in ctx then we can generalize it to be universally quantified.

$$\frac{x_1 \rhd a \quad x_2 \rhd ctx \vdash b}{\text{DISCH } x_1\ x_2 \rhd ctx - a \vdash a \supset b}$$

This is the deduction rule. If we can prove b, we can prove $a \supset b$ without the assumption a.

$$\frac{x_1 \rhd \theta \quad x_2 \rhd ctx \vdash p}{\text{INST\_TYPE } x_1\ x_2 \rhd \theta(ctx) \vdash \theta(p)}$$

This rule instantiates the type variables in the theorem. θ is a list of pairs, mapping type variables to new types. The definition of the substitution is the obvious one.

$$\frac{x \rhd c = t}{\text{NEW\_DEFINITION } x\ \rhd \cdot \vdash c = t}$$

This rule defines a new constant, c with definition t. c must be undefined, and after successful evaluation makes $c = t$ a new HOL Light axiom.

$$\frac{x_1 \rhd con \quad x_2 \rhd rep \quad x_3 \rhd abs \quad x_4 \rhd \cdot \vdash \exists v.\ p(v)}{\begin{array}{l}\textsc{new\_type\_definition}\ x_1\ (x_2, x_3)\ x_4 \rhd \\ \quad \cdot \vdash (\forall a.\ abs(rep\ a) = a) \wedge (\forall r.\ pr \iff rep(abs\ r) = r)\end{array}}$$

This rule defines a new type constructor, *con*. *abs*, *rep* are names of the functions giving a bijection with a subset p of an existing type. The theorem proves that p is nonempty. Again, see [21] for details.

B Isabelle/HOL

B.1 Abstract syntax

The syntax of Isabelle/HOL[3] is similar, though there are also *classes* and *sorts*. Class identifiers are represented by the metavariable γ. Type variables α are classified by sorts σ. A sort is a collections of classes, and represents the intersection of its elements. Instead of the LCF approach, Isabelle stores proof terms. Thus the syntactic class of proofs look somewhat different. An Isabelle theory is a sequence of declarations (*dec*).

$$\begin{array}{rll}
\tau & ::= \alpha :: \sigma \mid c(\tau_1, \ldots, \tau_k) & \text{(sorted type variables and constructors)} \\
t & ::= x \mid c : \tau \mid t\ t' \mid \lambda x : \tau.\ t & \text{(terms)} \\
p & ::= & \text{(proof terms)} \\
& \mid x & \text{(proof variable)} \\
& \mid p\ t & \text{(universal instantiation)} \\
& \mid p\ p' & \text{(modus ponens)} \\
& \mid \Lambda t : \tau.\ p & \text{(universal generalization)} \\
& \mid \Lambda p : t.\ p' & \text{(deduction rule)} \\
& \mid \mathsf{axm}(id, t) & \text{(axiom with name } id \text{ and formula } t) \\
& \mid \mathsf{thm}(id, t) & \text{(theorem with name } id \text{ and formula } t) \\
dec & ::= \mathsf{Axm}(id, t) \mid \mathsf{Thm}(id, t, pf) \mid \mathsf{def}(id, c, \tau, t) \mid \mathsf{typedef}(id, rep, abs, pf) & \\
& \quad \mid \mathsf{axclass}(id, id\ list, id\ list, t\ list) \mid \mathsf{instance}(id, \tau, \gamma, pf) &
\end{array}$$

[3] Because Isabelle is a logical framework and HOL is a single instance, there is a distinction in Isabelle between types and terms of the meta and object level. We identify the levels for the translation, thus the absence of type prop, *etc.*

B.2 Semantics

Proof terms Proof terms can be interpreted as in the Curry-Howard correspondence [18], where abstraction over a term variable represents universal quantification and abstraction over a proof variable represents implication introduction. Here, instead of Γ being a list of assumed theorems as in HOL Light, it is a map from proof variables to the formulas they prove. Here Φ is the Isabelle store of axiom and previously proved theorems. The judgment $\Phi, \Gamma \vdash p : t$ should be read, "Given Isabelle store Φ and context Γ, the proof term p is a proof of formula t.

$$\frac{\Gamma(x) = t}{\Phi, \Gamma \vdash x : t}$$

$$\frac{\Phi, \Gamma \vdash p : \forall x.\ q}{\Phi, \Gamma \vdash p\ t : [t/x]q}$$

$$\frac{\Phi, \Gamma \vdash p : a \supset b \quad \Phi, \Gamma \vdash p' : a}{\Phi, \Gamma \vdash p\ p' : b}$$

$$\frac{\Phi, (\Gamma, x : \tau) \vdash p : a}{\Phi, \Gamma \vdash \Lambda x : \tau.\ p : \forall x.\ a}$$

$$\frac{\Phi, (\Gamma, x : a) \vdash p' : b}{\Phi, \Gamma \vdash \Lambda x : a.\ p' : a \supset b}$$

$$\frac{\Phi(id) = t}{\Phi, \Gamma \vdash \mathsf{axm}(id, t) : t}$$

$$\frac{\Phi(id) = t}{\Phi, \Gamma \vdash \mathsf{thm}(id, t) : t}$$

C Elaboration rules

Elaboration is syntax directed, based on the abstract syntax for Isabelle given above.

C.1 Types

$$\boxed{\Delta \vdash \tau \rightsquigarrow T}$$

$$\frac{\Delta(\alpha) = T}{\Delta \vdash \alpha :: \sigma \rightsquigarrow T}$$

$$\frac{\Delta \vdash \tau_1 \rightsquigarrow A_1 \quad \ldots \quad \Delta \vdash \tau_k \rightsquigarrow A_k}{\Delta \vdash con(\tau_1, \ldots, \tau_k) \rightsquigarrow \ulcorner con \urcorner (A_1) \ldots (A_k)}$$

C.2 Terms

$$\boxed{\Delta, \Gamma \vdash t \rightsquigarrow t}$$

$$\frac{\Gamma(x) = \tau \quad \Delta \vdash \tau \rightsquigarrow A}{\Delta, \Gamma \vdash x \rightsquigarrow x : A.\textsc{typ}}$$

$$\frac{\mathtt{tv}(\tau) = (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_k, \sigma_k \rangle) \quad \Delta(\alpha_1, \ldots, \alpha_n) = (A_1, \ldots, A_n)}{\Delta, \Gamma \vdash c : \tau \rightsquigarrow \ulcorner c \urcorner (A_1) \ldots (A_n).\textsc{term}}$$

$$\frac{\Delta, \Gamma \vdash t_1 \rightsquigarrow t_1' \quad \Delta, \Gamma \vdash t_2 \rightsquigarrow t_2'}{\Delta, \Gamma \vdash t_1\ t_2 \rightsquigarrow t_1'\ t_2'}$$

$$\frac{\Delta \vdash \tau \rightsquigarrow A \quad \Delta, (x : A.\textsc{typ}, \Gamma) \vdash t \rightsquigarrow t'}{\Delta, \Gamma \vdash \lambda x : \tau.\ t \rightsquigarrow \lambda x : A.\textsc{typ}.\ t'}$$

C.3 Proofs

$$\boxed{\Gamma \vdash pf \rightsquigarrow thm}$$

$$\frac{\Gamma(x) = t \quad \Gamma \vdash t \rightsquigarrow t'}{\Gamma \vdash x \rightsquigarrow \textsc{assume}\ t'}$$

$$\frac{\Delta, \Gamma \vdash p \rightsquigarrow thm \quad \Gamma \vdash t \rightsquigarrow t'}{\Gamma \vdash p\ t \rightsquigarrow \textsc{spec}\ t'\ thm}$$

$$\frac{\Gamma \vdash p_1 \rightsquigarrow thm_1 \quad \Gamma \vdash p_2 \rightsquigarrow thm_2}{\Gamma \vdash p_1\ p_2 \rightsquigarrow \textsc{mp}\ thm_1\ thm_2}$$

$$\frac{\Delta \vdash \tau \rightsquigarrow T \quad \Delta, (x : T.\text{TYP}, \Gamma) \vdash p \rightsquigarrow thm}{\Gamma \vdash \Lambda x : \tau.\ p \rightsquigarrow \text{GEN}\ (x : T.\text{TYP})\ thm}$$

$$\frac{\Delta, \Gamma \vdash t \rightsquigarrow t' \quad \Delta, (v : t', \Gamma) \vdash p \rightsquigarrow p'}{\Gamma \vdash \Lambda x : t.\ p \rightsquigarrow \text{DISCH}\ t'\ p'}$$

$$\frac{\mathtt{tv}(t) = (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_k, \sigma_k \rangle \quad \Delta(\alpha_1, \ldots, \alpha_k) = (A_1, \ldots, A_k)}{\Gamma \vdash \mathsf{axm}(id, t) \rightsquigarrow \ulcorner id \urcorner (A_1) \ldots (A_n).\text{THM}}$$

$$\frac{\mathtt{tv}(t) = (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_k, \sigma_k \rangle) \quad \Delta(\alpha_1, \ldots, \alpha_k) = (A_1, \ldots, A_k)}{\Gamma \vdash \mathsf{thm}(id, t) \rightsquigarrow \ulcorner id \urcorner (A_1) \ldots (A_n).\text{THM}}$$

C.4 Declarations

$\boxed{\Sigma \vdash dec \rightsquigarrow \Sigma'}$

$$\frac{\mathtt{tv}(t) = (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_k, \sigma_k \rangle) \quad \mathtt{fresh}(A_1, \ldots, A_k)}{\begin{array}{l}\Sigma \vdash \mathsf{Axm}(id, t) \rightsquigarrow \\ \quad \textbf{module}\ \ulcorner id \urcorner (A_1 : S_1) \ldots (A_k : S_k) : \text{THM} = \\ \quad \textbf{struct} \\ \quad\quad \textbf{val}\ \text{THM} = \text{INST\_TYPE}\ [A_1.\text{TYP}/\alpha_1, \ldots, A_k.\text{TYP}/\alpha_k]\ thm \\ \quad \textbf{end}, \Sigma\end{array}}$$

$$\frac{\begin{array}{c}\mathtt{tv}(t) = (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_k, \sigma_k \rangle) \quad \mathtt{fresh}(A_1, \ldots, A_k) \\ \Delta = (\langle \alpha_1, A_1 \rangle, \ldots, \langle \alpha_k, A_k \rangle) \quad \Delta, \cdot \vdash p \rightsquigarrow thm\end{array}}{\begin{array}{l}\Sigma \vdash \mathsf{Thm}(id, t, p) \rightsquigarrow \\ \quad \textbf{functor}\ \ulcorner id \urcorner (A_1 : \ulcorner \sigma_1 \urcorner) \ldots (A_k : \ulcorner \sigma_k \urcorner) : \text{THM} = \\ \quad \textbf{struct} \\ \quad\quad \textbf{val}\ \text{THM} = thm \\ \quad \textbf{end}, \Sigma\end{array}}$$

$$\frac{\begin{array}{c}\texttt{fresh}(A_1,\ldots,A_k)\\ \texttt{tv}(\tau) = (\langle\alpha_1,\sigma_1\rangle,\ldots,\langle\alpha_k,\sigma_k\rangle) \quad \Delta,\cdot\vdash t\rightsquigarrow t'\end{array}}{\begin{array}{l}\Sigma\vdash\mathsf{def}(id,c,\tau,t)\rightsquigarrow\\ \quad\textbf{functor}\ \ulcorner id\urcorner(A_1:\ulcorner\sigma_1\urcorner)\ldots(A_k:\ulcorner\sigma_k\urcorner):\textsc{Thm} =\\ \quad\textbf{struct}\\ \quad\ \textbf{let}\ \textsc{thm} = \textsc{new\_definition}\ t'\\ \quad\textbf{end},\\ \quad\textbf{functor}\ \ulcorner c\urcorner(A_1:\ulcorner\sigma_1\urcorner)\ldots(A_k:\ulcorner\sigma_k\urcorner):\textsc{Term} =\\ \quad\textbf{struct}\\ \quad\ \textbf{let}\ \textsc{term} = \textsc{mk\_term}(\overline{c},[A_1.typ,\ldots,A_k.typ])\\ \quad\textbf{end},\Sigma\end{array}}$$

Note that in this rule we make use of a new function $\overline{x}$. This function is analogous to $\ulcorner x\urcorner$ and for any Isabelle constant returns the corresponding HOL Light constant name, rather than the corresponding module name. Again, its details are uninteresting. The OCaml function MK_TERM constructs HOL Light constants.

$$\frac{\begin{array}{c}\Delta = (\langle\alpha_1,A_1\rangle,\ldots,\langle\alpha_k,A_k\rangle) \quad \Delta,\cdot\vdash p\rightsquigarrow thm\\ \texttt{fresh}(A_1,\ldots,A_k)\end{array}}{\begin{array}{l}\Sigma\vdash\mathsf{typedef}(id,con,(\alpha_1,\ldots,\alpha_k),rep,abs,p)\rightsquigarrow\\ \quad\textbf{functor}\ \ulcorner id\urcorner(A_1:\ulcorner\sigma_1\urcorner)\ldots(A_k:\ulcorner\sigma_k\urcorner):\textsc{Thm} =\\ \quad\textbf{struct}\\ \quad\ \textbf{let}\ \textsc{thm} = \textsc{new\_type\_definition}\ (\ulcorner abs\urcorner,\ulcorner rep\urcorner)\ thm\\ \quad\textbf{end},\\ \quad\textbf{module}\ \ulcorner con\urcorner(A_1:\ulcorner\sigma_1\urcorner)\ldots(A_k:\ulcorner\sigma_k\urcorner):\textsc{Type} =\\ \quad\textbf{struct}\\ \quad\ \textbf{let}\ \textsc{term} = \textsc{mk\_type}(\overline{c},[A_1.typ,\ldots,A_k.typ])\\ \quad\textbf{end},\Sigma\end{array}}$$

$$
\frac{
\begin{array}{l}
\fresh(A_1, \ldots, A_k) \quad \Delta = (\langle \alpha_1, A_1 \rangle, \ldots, \langle \alpha_k, A_k \rangle) \\
\Delta, \cdot \vdash con_1 \rightsquigarrow c_1 \quad \ldots \quad \Delta, \cdot \vdash con_n \rightsquigarrow c_n \\
\Delta, \cdot \vdash p \rightsquigarrow thm \quad thm = (thm_1 \wedge \ldots \wedge thm_m) \\
\left.
\begin{array}{l}
\textbf{signature} \ulcorner c \urcorner = \\
\textbf{sig} \\
\quad \textbf{val} \ulcorner con_1 \urcorner \ : \ \textsc{term} \ \ldots \ \textbf{val} \ulcorner con_l \urcorner \ : \ \textsc{term} \\
\quad \textbf{val} \ axm_1 \ : \ \textsc{thm} \ \ldots \ \textbf{val} \ axm_m \ : \ \textsc{thm} \\
\textbf{end}
\end{array}
\right\} \in \Sigma \\
A = \ulcorner \tau(\sigma_i, \ldots, \sigma_n) \urcorner
\end{array}
}{
\begin{array}{l}
\Sigma \vdash \mathsf{instance}\ \tau :: (\langle \alpha_1, \sigma_1 \rangle, \ldots, \langle \alpha_n, \sigma_n \rangle)\ c = \langle [con_1, \ldots, con_k], p \rangle \rightsquigarrow \\
\quad \textbf{functor}\ A(A_1 : \ulcorner \sigma_1 \urcorner) \ldots (A_k : \ulcorner \sigma_n \urcorner) = \\
\quad \textbf{struct} \\
\quad\quad \textbf{let} \ulcorner con_1 \urcorner = c_1 \ \ldots \ \textbf{let} \ulcorner con_k \urcorner = c_k \\
\quad\quad \textbf{let}\ axm_1 = thm_1 \ \ldots \ \textbf{let}\ axm_m = thm_m \\
\quad \textbf{end}, \Sigma
\end{array}
}
$$

$$
\frac{
\begin{array}{l}
\Sigma \vdash \mathsf{axclass}\ c < [c_1, \ldots, c_k] = [con_1, \ldots, con_l], \\
\qquad\qquad [name_1\ \mathsf{is}\ axm_1, \ldots, name_m\ \mathsf{is}\ axm_m] \rightsquigarrow \\
\quad \textbf{signature} \ulcorner c \urcorner = \\
\quad \textbf{sig} \\
\quad\quad \textbf{include} \ \ulcorner c_1 \urcorner \ \ldots \ \textbf{include} \ \ulcorner c_k \urcorner \\
\quad\quad \textbf{val} \ \ulcorner con_1 \urcorner \ : \ \textsc{term} \ \ldots \ \textbf{val} \ \ulcorner con_l \urcorner \ : \ \textsc{term} \\
\quad\quad \textbf{val} \ \ulcorner name_1 \urcorner \ : \ \textsc{thm} \ \ldots \ \textbf{val} \ \ulcorner name_m \urcorner \ : \ \textsc{thm} \\
\quad \textbf{end}, \Sigma
\end{array}
}{}
$$