Encoding the HOL Light logic in Coq

Freek Wiedijk

freek@cs.kun.nl
University of Nijmegen

Abstract. We show how to encode the HOL Light logic in Coq. This makes an automatic translation of HOL proofs to Coq possible. The translated HOL proofs refer to translated HOL data types but those data types can be related to the standard Coq data types, making the HOL results useful for Coq. The translated proofs have a size linear in the time HOL takes to process the original proofs. However the constant of linearity is large. The approach described in this paper is similar to the method of Pavel Naumov, Mark-Oliver Stehr and José Mesequer for translating HOL98 proofs to Nuprl [10].

1 Introduction

1.1 Problem

There are many systems for proof formalization. Currently popular systems are HOL [5], Coq [2], Nuprl [3], Isabelle [12], ACL2 [9] and PVS [11]. There are two views on this multitude of systems. The one view is to expect the best system to take the lead over the other ones. The other view is to expect the systems to coexists and to be able to exchange mathematical results between them. Often this second view is utopian and presents a future in which 'mathematical services' will be available through the world wide web. These services will then solve almost all mathematical problems at the click of a mouse. Of course for this to become a reality the various system need to be able to understand each other's mathematics.

There are two reasons why exchanging mathematics between systems is difficult: a theoretical and a practical reason. The theoretical reason is that the notions of the various systems are generally not compatible. For instance Randy Pollack writes [14]:

> *An advantage [...] is that proofs in the same logic can be shared by different proof checkers for that logic if a standard syntax can be found (or mechanical translations believed), because the official proofs don't depend on the tactics that are particular to individual proof tools. In current practice this idea is a can of worms, and the phrase 'in the same logic' causes experts in the field to roll on the floor with laughter.*

The practical reason why exchanging mathematics between systems is difficult is that translation makes things bigger and slower. The translated mathematics is an order of magnitude worse than the mathematics done within the system itself (like for instance 100M instead of 1M). Also mathematics imported from a foreign system will be less readable and often will not follow the conventions of the system. So people will not be inclined to use translation.

This paper looks at a specific case of the translation problem. We investigate a method to move mathematics from the HOL system to Coq.

The incentive to look at this was that in Nijmegen we were interested in formalizing a proof of the fundamental theorem of calculus in Coq. The fundamental theorem of calculus is the theorem that states that integration and differentiation are inverse to each other. But then it turned out that this theorem had already been proved by John Harrison in HOL, called theorem `FTC1`. So we wanted to know whether it would be possible to import this result from HOL in Coq.

1.2 Approach

This paper will model the HOL logic in Coq. So for every HOL notion we will have a Coq definition. This means that we build a *model* of HOL inside the Coq universe. Therefore when we translate a HOL statement it will become a statement about this model. However we have chosen a *natural* representation for the various HOL constructions. This makes it possible to relate the HOL objects to their Coq counterparts. For instance the HOL type of natural numbers is called `num`. To this corresponds a Coq type `(hol_elts hol_'num)` in the Coq model of HOL which is different from the Coq type of the natural numbers `nat`. However `(hol_elts hol_'num)` turns out to be isomorphic to `nat`, so statements about the first type do apply to the second type.

Coq is a constructive system while HOL is classical. In order to be able to do the HOL constructions we will need to add two axioms to the Coq logic: a version of the axiom of choice and an extensionality axiom.

The formalization of the HOL logic in Coq will allow us to translate HOL proofs to Coq. This will not be done on the level of the tactics but on the level of the basic inferences. So the HOL tactics will *not* become available to Coq, nor will the Coq translation resemble the HOL original.

The translation scheme that we present in this paper can be automatized but we have not implemented this. Instead we have only translated a sample proof manually. This showed that the translation of a realistic HOL theorem would become big, after which we decided not to proceed with an implementation.

1.3 Related work

When this paper was almost finished my attention was drawn to the work of Ewen Denney for translating HOL98 proofs to Coq [4] and to the work of Pavel Naumov, Mark-Oliver Stehr and José Mesequer for translating HOL98 proofs to Nuprl [10]. The work in this paper is analogous to this but for HOL Light instead of HOL98.

1.4 Contribution

This paper presents three results:

– It shows how to model the HOL Light logic in the Coq system.
– It presents a way to translate HOL Light mathematics to Coq.
– It shows a way to move mathematics between systems without the translated mathematics becoming 'foreign' to the importing system.

This paper is the explanation of a Coq file `holl2coq.v` that contains the formalization of HOL Light in Coq. It is available on the web as:

`http://www.cs.kun.nl/~freek/notes/holl2coq.v`

1.5 Outline

This paper has the following structure. First we present the HOL and Coq systems. Then we show how to model the HOL logic with Coq. Applying this we translate a small HOL proof to Coq. Finally we show that the translated HOL natural numbers are isomorphic to the Coq natural numbers.

2 HOL

The HOL system that we use in this paper is HOL Light by John Harrison [7]. This is one of the cleanest implementations of HOL [6]. We will now outline the logic of this system.

2.1 The HOL logic

The HOL system has three kinds of notions called *types*, *terms* and *theorems*.

The terms of HOL are the terms of simply typed lambda calculus. A HOL term either is a variable, a constant, a function application

$$(t\,u)$$

or a lambda abstraction

$$\lambda x.t$$

For instance a HOL term might look like $(f\,\lambda a.(e\,\lambda x.((i\,x)\,a)))$ where f, e and i are constants and a and x are variables. The difference between HOL terms and those of simply typed lambda calculus is that HOL terms can be polymorphic (in the style of ML). For instance the constant $=$ has type $\alpha \to (\alpha \to \texttt{bool})$ where α is a type variable. This means that $=$ corresponds to a collection of constants $=_A$, one for every type A.

The types of HOL are the types of simply typed lambda calculus. There are two primitive types:

$$\texttt{bool}$$
$$\texttt{ind}$$

The first is the type of the two truth values `T` and `F`. The second is the type of *individuals* (the only thing that is given about this type is that it is infinite). One can define new types by taking an existing type A and a predicate P on it. The new type corresponds to the elements x of A that satisfy Px, so it will behave like:

$$\{x : A \mid Px\}$$

Apart from these scalar types (the two basic types and the defined types) the HOL logic has function types:

$$A \to B$$

for all types A and B.

The two basic constants of HOL are the equality predicate and the choice operator:

$$= : \alpha \to (\alpha \to \texttt{bool})$$
$$\texttt{@} : (\alpha \to \texttt{bool}) \to \alpha$$

The choice operator selects an object satisfying a predicate. So $\texttt{@}x : A.\, Px$ is some x of type A that satisfies Px (in the case that one or more such x exist; if none exist it is some arbitrary x of type A). For every defined type there are two constants that map the defined type and the type it's carved out of to each other. Every other constant in the system (not one of the two basic constants nor one of a pair of mappings associated to a type definition) is defined as an abbreviation of a HOL term.

The theorems of HOL are certain sequents of the shape:

$$t_1,\, t_2,\, \ldots,\, t_n \vdash t$$

where $t_1, \ldots, t_n$ and t are terms of type `bool`. A theorem is either one of three axioms, a theorem corresponding to a defined constant (stating its definition), one of two theorems corresponding to a defined type (stating that the mappings are inverse to each other) or it is derived from other theorems by one of ten inference rules. The three axioms are called:

```
ETA_AX
SELECT_AX
INFINITY_AX
```

These respectively are an extensionality axiom in the form of the η conversion rule, the axiom of choice in terms of the `@` operator and an infinity axiom for the `ind` type. The ten inference rules are called `ABS`, `ASSUME`, `BETA`, `DEDUCT_ANTISYM_RULE`, `EQ_MP`, `INST`, `INST_TYPE`, `MK_COMB`, `REFL` and `TRANS`. For the details of the axioms and the inference rules we refer to the HOL Light documentation [7].

2.2 The theorems of HOL Light

The HOL Light system has been implemented in the functional language CAML Light which is a dialect of ML. When the HOL Light system starts it processes its basic library. This takes about ten minutes.[1] During this time HOL Light generates about 1.5 million objects of the ML data type `thm`. This data type corresponds to the theorems of the HOL Light logic as outlined in the previous subsection. Many of these `thm`s get garbage collected after a while. So the memory used by the HOL Light system does not need to have room for all 1.5 million `thm`s.

In order to get to an advanced theorem more `thm`s have to be generated. When processing the files `analysis.ml` and `transc.ml` the system generates another million `thm`s. The fundamental theorem of calculus is one of the last theorems in the `transc.ml`. The `thm` that is the theorem `FTC1` is the 2535069th `thm` that is generated by HOL Light.

3 Coq

The Coq system [2] from France is similar to HOL. Both are implemented in ML and are tactic based proof assistants in LCF-style. We will not discuss the Coq logic in detail. We will only focus on the difference in the way that HOL and Coq treat proofs.

3.1 Coq proofs

HOL and Coq both generate *proof objects* for proved statements. Proof objects are compound objects built from a small number of primitives. In the case of HOL those primitives are the ten inference rules. In the case of Coq those are eighteen term building primitives called `Rel`, `Var`, `Meta`, `Evar`, `Sort/Prop/Pos`, `Sort/Prop/Null`, `Sort/Type`, `Cast`, `Prod`, `Lambda`, `LetIn`, `App`, `Const`, `Ind`, `Construct`, `Case`, `Fix` and `CoFix`.

There are two differences in the way HOL and Coq treat proofs:

- HOL's proofs are checkable in *linear time*. This means that the time needed to check the proof is linear in the size of the proof object. Every primitive in the proof object only takes a small amount of time to process.
  In contrast with this Coq proof objects can be much smaller than the time needed to check them. That is because the system has to verify *convertibility* of terms (to be specific: $\beta\delta\iota$-convertibility) without there being anything in the proof object corresponding to that. In other words, in the Coq logic 'calculation needs no proof'.[2]

[1] On a system with a performance of about 10 SPECint95. The fastest current computers are about 50 SPECint95.
[2] This is what Henk Barendregt calls the 'Poincaré principle' [1] because Henri Poincaré wrote in [13] about showing the correctness of the calculation $2 + 2 = 4$:

This means that there is no recursive function that gives a bound on the 'checking time' of a Coq proof object as function of its size. Some people think that this is an undesirable property for a notion of 'proof'. Other people think that it is desirable because it keeps the size of the proof object down.
- In HOL the parts of a proof object are processed 'on the fly' and are not stored in memory, while in Coq proof objects are stored. In HOL a theorem is just a statement of which it is known that it has been proved. In Coq a theorem is a statement together with a *proof term*, a term that represents the proof object. This is essential to the Coq logic. Checking Coq proofs without keeping those proof terms is not possible.

3.2 The features of Coq's logic

HOL is classical and Coq is constructive, so one might expect that HOL's logic is more complex than Coq's. In fact it is the other way around. If one ignores the classical parts then HOL's logic is a subset of Coq's. Both logics are typed lambda calculi but Coq's logic has many features that HOL's has not. The Coq logic contains:

- full polymorphism (not just ML-style polymorphism)
- dependent types
- the distinction between computational and non-computational propositions
- an infinite hierarchy of type universes
- inductive and co-inductive types built into the logic

Since Coq's logic is more complicated than HOL's it's not surprising that in this paper we embed HOL in Coq. The other way around would be much harder.

3.3 Using Coq as a proof checker

In this paper we make some claims about Coq's logic that have only be established by experimenting with the Coq system. So if there is a bug in Coq then there might be incorrect claims here.

4 Encoding the HOL logic in Coq

We will now explain how we encoded the HOL logic in Coq. We will first present the axioms that we added to Coq to get classical reasoning. Then we will show how we encoded HOL's types, terms and theorems in Coq.

'Ce n'est pas une démonstration proprement dite, [...] *c'est une vérification'.* [...] *La vérification diffère précisément de la véritable démonstration, parce qu'elle est purement analytique et parce qu'elle est stérile.*

4.1 The axioms

HOL is a classical system and Coq is not. So we need to add classical axioms to Coq to be able to translate HOL proofs. We expected to have to add three axioms to Coq but it turned out that the first followed from the second. So we only have to add two axioms. However we will show all three statements here:

Classical logic. The difference between constructive and classical predicate logic is the double negation law. This becomes in Coq:

```
Lemma double_negation :
  (A:Prop)~(A->False)->A.
```

We wrote `~(A->False)` instead of `~~A` to make this an analogue of the next axiom.

The axiom of choice. The HOL logic has a *choice operator* `@`. To model this we want to have an operator in Coq that selects an element from any non-empty `Set`. This operator is given by:

```
Axiom choice :
  (A:Set)~(A->False)->A.
```

The type `A->False` is inhabited if and only if `A` is empty so one can read

`A->False`

as:

the set `A` *is empty*

So this axiom is an operator that takes a proof that a set is not empty and returns an element of that set. Which is a choice operator.

This axiom is *not* what constructivists call the axiom of choice. For constructivists the axiom of choice is an operator associating a function to a relation. Constructivists probably see the condition of our axiom as the double negation of the *real* way to say that the set is non-empty which is by giving an element.

However the resemblance to a choice operator is striking which is why we didn't call this the double negation law for `Set` but the axiom of choice.

Extensionality. The HOL logic is extensional. If two functions have equal values everywhere then they are equal. In Coq this becomes:

```
Axiom extensionality :
  (A,B:Set; f,g:A->B)((x:A)(f x)=(g x))->f=g.
```

The equality `=` in this axiom is Coq's standard Leibniz equality.

4.2 Encoding the types

A HOL type almost corresponds to a Coq `Set`. However a HOL type always has to be non-empty while a Coq `Set` can be empty. Therefore we can't represent a HOL type by a Coq `Set`. Instead we define:

```
Record hol_type : Type :=
  { hol_elts  :> Set;
    hol_inhab :  hol_elts
  }.
```

and let a HOL type be represented by an object of type `hol_type`. The function `hol_elts` is defined to be a coercion so it can be omitted and we can write `x:A` instead of `x:(hol_elts A)`.

There are three kinds of HOL type that we need to define: basic types, function types and defined types.

Basic types. The two basic HOL types `bool` and `ind` correspond naturally to the Coq `Set`s `bool` and `nat`:

```
Definition hol_'bool : hol_type :=
  (Build_hol_type bool false).

Definition hol_'ind : hol_type :=
  (Build_hol_type nat O).
```

The HOL type `bool` is for HOL what `Prop` is for Coq. Hence we need to connect these to each other. We define:

```
Definition hol_thm : hol_'bool->Prop :=
  [p:hol_'bool](if p then True else False).
```

So if `A` is a translation in `hol_'bool` of some HOL formula then:

```
(hol_thm A)
```

is the Coq way of saying that this formula holds.

Function types. The HOL function types are just the Coq function types:

```
Definition hol_'fun : hol_type->hol_type->hol_type :=
  [A,B:hol_type](Build_hol_type A->B [x:A](hol_inhab B)).
```

Note that in this definition `A->B` really means `(hol_elts A)->(hol_elts B)`. So we have that:

```
(hol_elts (hol_'fun A B)) = (hol_elts A)->(hol_elts B)
```

Although we use an abstract representation `hol_type` for the HOL types, if we apply `hol_elts` we get the `Set`s that naturally corresponds to them.

Type definitions. Finally there are the defined HOL types, which are carved out of an already existing type by a predicate on that type. Modeling these in Coq turned out to be more difficult than expected.

Naively one would expect that we could represent the defined types by a Σ type. If `A` is some HOL type and `P:A->Prop` is some HOL predicate on it then:

$$\Sigma x : A.\, P x$$

which in Coq syntax is:

```
(sig A P)
```

seems to represent the set of elements in `A` that satisfy `P`. However this approach doesn't work because the proposition type `(P x)` can have many different inhabitants. This means that for each `x:A` there will be many elements in `(sig A P)`. Which is wrong.

The way to solve this is to use the `Set` variant of the Σ type construction (which in Coq is called `sigS`). If a proposition is true (we can decide that because we have the classical axioms) we associate a `Set` with one element with it. That way we can define a 'clone' of `sig` called `sig'` that behaves classically:

```
Definition irrelevant : Prop->Set :=
  [A:Prop]
    Cases (excluded_middle A) of
      (left _) => unit
    | (right _) => Empty_set
    end.

Definition sig' : (A:Set)(A->Prop)->Set :=
  [A:Set; P:A->Prop](sigS A [x:A](irrelevant (P x))).
```

Using this we can now give the Coq version of the HOL type definitions:

```
Definition hol_typedef :=
  [A:hol_type; P:(hol_'fun A hol_'bool);
      i:A; H:(hol_thm (P i))]
    (Build_hol_type (sig' A (Hol_thm' P)) (exist' ?? i H)).
```

Here `Hol_thm'` maps HOL predicates of type `(hol_'fun A hol_'bool)` to Coq predicates of type `A->Prop`. It is defined such that `(Hol_thm' P x)` is the same as `(hol_thm (P x))`. The element `i:A` is needed to show that the defined type is non-empty. It is needed in the HOL version of a HOL type definition as well.

4.3 Encoding the terms

HOL terms straightforwardly translate to Coq terms. For an example, the HOL term corresponding to:

$$(f\, \lambda a.(e\, \lambda x.((i\, x)\, a)))$$

which in HOL syntax is:

```
f \a. e \x. i x a
```

becomes:

```
(hol'f (Hol_Abs [a:A](hol'e (Hol_Abs [x:A](hol'i x a)))))
```

Here `A` is the `hol_type` of the variables `a` and `x`. We prefix the HOL names with '`hol'`' to indicate that they are translations of HOL constants.

There are two elements in this translation that needs to be explained: how abstraction is handled and how constants are modeled.

Abstraction. As will be clear from the example, function application in HOL translates to function application in Coq. However for abstraction we have a typing problem. The HOL functions have to have type `(hol_'fun A B)` but Coq abstraction has type `A->B`. To solve this we have the `Hol_Abs` function:

```
Definition hol_Abs :
  (A,B:hol_type)(A->B)->(hol_'fun A B) :=
    [A,B:hol_type; f:A->B]f.

Syntactic Definition Hol_Abs := (hol_Abs ??).
```

This `Hol_Abs` doesn't do anything apart from changing the type from:

```
A->B
```

to the equivalent:

```
(hol_elts (hol_'fun A B))
```

Although these are $\beta\delta\iota$-convertible, Coq isn't smart enough to figure this type conversion out by itself. If we remove the `Hol_Abs` functions we get typing errors.

Equality. The first basic HOL constant is equality. We model it by making it correspond to the usual equality in Coq which is the Leibniz equality `=`.

However the Coq equality is in `Prop` while HOL predicates need to be in `hol_'bool`. Therefore we need to map `Prop` to `hol_'bool` (an inverse to the `hol_thm` function):

```
Definition hol_value : Prop->hol_'bool :=
  [A:Prop]
    Cases (excluded_middle A) of
      (left _) => true
    | (right _) => false
    end.
```

Using this the definition of the HOL equality `hol''eq` becomes straightforward:

```
Definition hol''eq :
  (A:hol_type)(hol_'fun A (hol_'fun A hol_'bool)) :=
    [A:hol_type](Hol_Abs [x:A](Hol_Abs [y:A]
      (hol_value x=y))).

Syntactic Definition Hol''eq := (hol''eq ?).
```

The HOL type of the HOL constant `(=)` is the polymorphic:

```
A->(A->bool)
```

As can be seen from the definition this corresponds in Coq to:

```
(A:hol_type)(hol_'fun A (hol_'fun A hol_'bool))
```

The choice operator. The other basic HOL constant is the choice operator. Its definition is straightforward too:

```
Definition hol''select :
  (A:hol_type)(hol_'fun (hol_'fun A hol_'bool) A) :=
    [A:hol_type](Hol_Abs [P:(hol_'fun A hol_'bool)]
      Cases (inhabited {x:A | (hol_thm (P x))}) of
        (inl x) => (proj1_sig ?? x)
      | (inr _) => (hol_inhab A)
      end).

Syntactic Definition Hol''select := (hol''select ?).
```

The `inhabited` function is defined using the `choice` axiom and decides whether a `Set` is inhabited or not. It is the `Set` equivalent of the law of the excluded middle.

Mapping defined types. Every HOL type definition produces two constants that map the defined type (the *abstract* type) to the type it is carved out of (the *representing* type) and back. The Coq equivalents of these constants are given by functions:

```
hol_typedef_abs :
  (A:hol_type; P:(hol_'fun A hol_'bool);
    i:A; H:(hol_thm (P i)))
      A->(hol_typedef A P i H)
```

and

```
hol_typedef_rep :
  (A:hol_type; P:(hol_'fun A hol_'bool);
    i:A; H:(hol_thm (P i)))
      (hol_typedef A P i H)->A
```

Their definition is straightforward. The only subtlety is that `hol_typedef_abs` returns the constant `i` in the case that the element doesn't satisfy `P`.

Defined constants. All other HOL constants are just abbreviations of HOL terms. These straightforwardly translate to Coq `Definitions`. The only subtlety is that type variables need to be given explicitly as arguments.

As an example consider the HOL universal quantifier `(!)`. It is defined by:

```
(!) = \P:A->bool. P = \x. T
```

This becomes in Coq:

```
Definition hol''forall :
  (A:hol_type)
    (hol_'fun (hol_'fun A hol_'bool) hol_'bool) :=
      [A:hol_type](Hol_Abs [P:(hol_'fun A hol_'bool)]
        (Hol''eq P (Hol_Abs [x:A]hol'T))).

Syntactic Definition Hol''forall := (hol''forall ?).
```

In this kind of translated definitions we use the `Syntactic Definitions` to establish the type variables. So we write `(Hol''eq ...)` instead of `(hol''eq A ...)`. This is similar to doing ML style type inference. Sometimes it does not work. In that case the type variable arguments need to be given explicitly.

4.4 Encoding the theorems

To conclude our description of the Coq implementation of the HOL logic we show how HOL theorems are modeled in Coq. A HOL theorem:

$$t_1, t_2, \ldots, t_n \vdash t$$

becomes a Coq `Prop`:

$\mathcal{Q}$ `(hol_thm` t_1`)->(hol_thm` t_2`)->`$\ldots$`(hol_thm` t_n`)->(hol_thm` t`)`

where $\mathcal{Q}$ quantifies over the type variables and term variables in the theorem.

For instance the HOL theorem:

```
x = y |- y = x
```

(where `x` and `y` have type `A`) becomes the Coq `Prop`:

```
(A:hol_type; x,y:A)
  (hol_thm (Hol''eq x y))->(hol_thm (Hol''eq y x))
```

Inference rules. We now show how the ten HOL basic inference rules are implemented. Those rules operate on HOL theorems with a list of assumptions before the $\vdash$. But the Coq representation of theorems is *implicit* in the sense that there is not a Coq data type representing the HOL theorems. Therefore the assumptions are not present in the Coq representations of the HOL inference rules and have to be manipulated outside those rules.

For instance the DEDUCT_ANTISYM_RULE rule is:

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

The Coq implementation ignores the $(\Gamma - \{q\}) \cup (\Delta - \{p\})$ assumptions. It is:

```
hol_DEDUCT_ANTISYM_RULE : (p,q:hol_'bool)
  ((hol_thm q)->(hol_thm p))->
  ((hol_thm p)->(hol_thm q))->
    (hol_thm (Hol''eq p q))
```

This says that logical equivalence implies equality of the truth values.

The ASSUME rule which deals with assumptions and the INST and INST_TYPE rules which are about substitution don't have a Coq counterpart. The other rules become:

```
hol_REFL : (A:hol_type; x:A)(hol_thm (Hol''eq x x))

hol_TRANS : (A:hol_type; x,y,z:A)
  (hol_thm (Hol''eq x y))->
  (hol_thm (Hol''eq y z))->
    (hol_thm (Hol''eq x z))

hol_MK_COMB : (A,B:hol_type; f,g:(hol_'fun A B); x,y:A)
  (hol_thm (Hol''eq f g))->(hol_thm (Hol''eq x y))->
    (hol_thm (Hol''eq (f x) (g y)))

hol_ABS : (A,B:hol_type; f,g:A->B)
  ((x:A)(hol_thm (Hol''eq (f x) (g x))))->
    (hol_thm (Hol''eq (Hol_Abs f) (Hol_Abs g)))

hol_BETA : (A,B:hol_type; f:A->B; x:A)
  (hol_thm (Hol''eq (Hol_Abs f x) (f x)))

hol_EQ_MP : (p,q:hol_'bool)
  (hol_thm (Hol''eq p q))->(hol_thm p)->(hol_thm q)
```

(The hol_BETA rule is almost the same as the hol_REFL rule because Coq does the β-reduction.)

Note that all these rules are about HOL equality.

The axioms and the theorems for type and constant definitions. The theorems that are not derived using one of the inference rules all translate in a straightforward way to Coq and can be proved easily. The Coq functions are for the axioms:

```
hol_ETA_AX : (A,B:hol_type)(hol_thm
  (Hol''forall (Hol_Abs [t:(hol_'fun A B)]
    (Hol''eq (Hol_Abs [x:A](t x)) t))))
```

```
hol_SELECT_AX : (A:hol_type)(hol_thm
  (Hol''forall (Hol_Abs [P:(hol_'fun A hol_'bool)]
    (Hol''forall (Hol_Abs [x:A]
      (hol''imp (P x) (P (Hol''select P))))))))

hol_INFINITY_AX : (hol_thm
  (Hol''exists (Hol_Abs [f:(hol_'fun hol_'ind hol_'ind)]
    (hol''and (Hol'ONE_ONE f) (hol''not (Hol'ONTO f))))))
```

and for the theorems about the mappings from and to defined types:

```
hol_typedef_absrep :
  (A:hol_type; P:(hol_'fun A hol_'bool);
    i:A; H:(hol_thm (P i)); a:(hol_typedef A P i H))
      (hol_thm (Hol''eq (Hol_typedef_abs
                          (Hol_typedef_rep a)) a))

hol_typedef_repabs :
  (A:hol_type; P:(hol_'fun A hol_'bool);
    i:A; H:(hol_thm (P i)); r:A)
      (hol_thm (Hol''eq
        (P r)
        (Hol''eq (Hol_typedef_rep
                   (hol_typedef_abs A P i H r)) r)))
```

The theorem for the constant definitions is `hol_REFL` applying the δ-reduction of Coq.

5 Translating a HOL proof to Coq

We manually translated a small HOL proof to Coq to see whether the formalization of the HOL logic in Coq was adequate. For this we looked in the basic HOL Light library for a proof that was shorter than 40 basic inference steps and that contained all inference step types. We could not find such a proof but we found a proof of 32 steps that contained all step types but `ASSUME`.

This is that proof:

```
let EXISTS_REFL = PROVE
 (`!a:A. ?x. x = a`,
  GEN_TAC THEN EXISTS_TAC `a:A` THEN REFL_TAC);;
```

It is the HOL equivalent of the Coq proof:

```
Lemma EXISTS_REFL : (A:Set; a:A)(Ex [x:A]x=a).
Proof.
Intros. Exists a. Reflexivity.
Qed.
```

So it is a proof of:

$$\forall a.\, \exists x.\, x = a$$

The 4 `thms` that are referred to from this proof are:

```
 17  |- t = t = T
171  p ==> q, p |- q
248  |- (P = (\x. T)) ==> (!) P
313  |- P x ==> (?) P
```

In this we numbered the `thms` that HOL Light generates. The HOL Light kernel doesn't have names for `thms`. Only outside the HOL Light kernel ML names are given to `thms`.

The proof of `EXISTS_REFL` consists of `thms` numbered 18749 until 18780:

```
18749  |- a = a                                     REFL
18750  |- (\x. x = a) x = x = a                     BETA
18751  |- (\x. x = a) a = a = a                     INST  18750
18752  |- P x ==> (?) P                        INST_TYPE  313
18753  |- (\x. x = a) a ==> (?x. x = a)             INST  18752
18754  |- (\x. x = a) a = (\x. x = a) a             REFL
18755  |- (=) = (=)                                 REFL
18756  |- (=) ((\x. x = a) a) = (=) (a = a)      MK_COMB  18755 18751
18757  |- ((\x. x = a) a = (\x. x = a) a) =      MK_COMB  18756 18754
            (a = a) = (\x. x = a) a
18758  |- (a = a) = (\x. x = a) a                  EQ_MP  18757 18754
18759  |- (\x. x = a) a                            EQ_MP  18758 18749
18760  (\x. x = a) a ==> (?x. x = a),               INST  171
       (\x. x = a) a
       |- ?x. x = a
18761  (\x. x = a) a                           DEDUCT...  18753 18760
       |- (\x. x = a) a ==> (?x. x = a) =
            (?x. x = a)
18762  (\x. x = a) a                               EQ_MP  18761 18753
       |- ?x. x = a
18763  |- (\x. x = a) a = (?x. x = a)          DEDUCT...  18759 18762
18764  |- ?x. x = a                                EQ_MP  18763 18759
18765  |- (?x. x = a) = (?x. x = a) = T             INST  17
18766  |- (?x. x = a) = T                          EQ_MP  18765 18764
18767  |- (\a. ?x. x = a) = (\a. T)                  ABS  18766
18768  |- (P = (\x. T)) ==> (!) P              INST_TYPE  248
18769  |- ((\a. ?x. x = a) = (\x. T)) ==>           INST  18768
            (!a. ?x. x = a)
18770  ((\a. ?x. x = a) = (\x. T)) ==>              INST  171
         (!a. ?x. x = a),
       (\a. ?x. x = a) = (\x. T)
       |- !a. ?x. x = a
18771  (\a. ?x. x = a) = (\x. T)               DEDUCT...  18769 18770
       |- ((\a. ?x. x = a) = (\x. T)) ==>
            (!a. ?x. x = a) = (!a. ?x. x = a)
```

```
18772 (\a. ?x. x = a) = (\x. T)              EQ_MP  18771 18769
      |- !a. ?x. x = a
18773 |- ((\a. ?x. x = a) = (\a. T)) =     DEDUCT... 18767 18772
          (!a. ?x. x = a)
18774 |- !a. ?x. x = a                       EQ_MP  18773 18767
18775 |- (\a. ?x. x = a) = (\a. ?x. x = a)    REFL
18776 |- (\a. ?x. x = a) = (\a. ?x. x = a)    REFL
18777 |- (\a. ?x. x = a) = (\a. ?x. x = a)   TRANS  18776 18775
18778 |- (!) = (!)                            REFL
18779 |- (!a. ?x. x = a) = (!a. ?x. x = a) MK_COMB  18778 18777
18780 |- !a. ?x. x = a                       EQ_MP  18779 18774
```

This derivation is not optimal. For instance it contains the final `thm` already as `thm` number 18774. Also `thms` 18775 until 18777 are all the same.

The Coq translation of these steps is simple. It is for each step an application of the appropriate function:

```
Definition hol__18749 := [A:hol_type; a:A]
  (Hol_REFL a).

Definition hol__18750 := [A:hol_type; x,a:A]
  (Hol_BETA [x:A](Hol''eq x a) x).

Definition hol__18751 := [A:hol_type; a:A]
  (hol__18750 ? a a).

Definition hol__18752 :=
    [A:hol_type; x:A; P:(hol_'fun A hol_'bool)]
  (hol__313 ? x P).

Definition hol__18753 := [A:hol_type; a:A]
  (hol__18752 ? a (Hol_Abs [x:A](Hol''eq x a))).

Definition hol__18754 := [A:hol_type; a:A]
  (Hol_REFL (Hol_Abs [x:A](Hol''eq x a) a)).

Definition hol__18755 := [A:hol_type]
  (Hol_REFL (hol''eq A)).

Definition hol__18756 := [A:hol_type; a:A]
  (Hol_MK_COMB (hol__18755 ?) (hol__18751 ? a)).

Definition hol__18757 := [A:hol_type; a:A]
  (Hol_MK_COMB (hol__18756 ? a) (hol__18754 ? a)).

Definition hol__18758 := [A:hol_type; a:A]
  (Hol_EQ_MP (hol__18757 ? a) (hol__18754 ? a)).

Definition hol__18759 := [A:hol_type; a:A]
  (Hol_EQ_MP (hol__18758 ? a) (hol__18749 ? a)).

Definition hol__18760 := [A:hol_type; a:A]
  (hol__171 (Hol_Abs [x:A](Hol''eq x a) a)
    (Hol''exists (Hol_Abs [x:A](Hol''eq x a)))).

Definition hol__18761 := [A:hol_type; a:A; H0:?]
```

```
  (Hol_DEDUCT_ANTISYM_RULE [H:?](hol__18753 ? a)
    [H:?](hol__18760 ? a H0 H)).

Definition hol__18762 := [A:hol_type; a:A; H0:?]
  (Hol_EQ_MP (hol__18761 ? a H0) (hol__18753 ? a)).

Definition hol__18763 := [A:hol_type; a:A]
  (Hol_DEDUCT_ANTISYM_RULE [H:?](hol__18759 ? a)
    [H:?](hol__18762 ? a H)).

Definition hol__18764 := [A:hol_type; a:A]
  (Hol_EQ_MP (hol__18763 ? a) (hol__18759 ? a)).

Definition hol__18765 := [A:hol_type; a:A]
  (hol__17 (Hol''exists (Hol_Abs [x:A](Hol''eq x a)))).

Definition hol__18766 := [A:hol_type; a:A]
  (Hol_EQ_MP (hol__18765 ? a) (hol__18764 ? a)).

Definition hol__18767 := [A:hol_type]
  (Hol_ABS [x:A](hol__18766 ? x)).

Definition hol__18768 := [A:hol_type; P:(hol_'fun A hol_'bool)]
  (hol__248 ? P).

Definition hol__18769 := [A:hol_type]
  (hol__18768 ? (Hol_Abs [a:A](Hol''exists (Hol_Abs [x:A]
    (Hol''eq x a))))).

Definition hol__18770 := [A:hol_type]
  (hol__171 (Hol''eq (Hol_Abs [a:A](Hol''exists (Hol_Abs [x:A]
      (Hol''eq x a)))) (Hol_Abs [x:A]hol'T))
    (Hol''forall (Hol_Abs [a:A](Hol''exists (Hol_Abs [x:A]
      (Hol''eq x a)))))).

Definition hol__18771 := [A:hol_type; H0:?]
  (Hol_DEDUCT_ANTISYM_RULE [H:?](hol__18769 A)
    [H:?](hol__18770 ? H0 H)).

Definition hol__18772 := [A:hol_type; H0:?]
  (Hol_EQ_MP (hol__18771 ? H0) (hol__18769 A)).

Definition hol__18773 := [A:hol_type]
  (Hol_DEDUCT_ANTISYM_RULE [H:?](hol__18767 A)
    [H:?](hol__18772 ? H)).

Definition hol__18774 := [A:hol_type]
  (Hol_EQ_MP (hol__18773 A) (hol__18767 ?)).

Definition hol__18775 := [A:hol_type]
  (Hol_REFL (Hol_Abs [a:A](Hol''exists (Hol_Abs [x:A]
    (Hol''eq x a))))).

Definition hol__18776 := [A:hol_type]
  (Hol_REFL (Hol_Abs [a:A](Hol''exists (Hol_Abs [x:A]
    (Hol''eq x a))))).

Definition hol__18777 := [A:hol_type]
```

```
  (Hol_TRANS (hol__18776 A) (hol__18775 ?)).

Definition hol__18778 := [A:hol_type]
  (Hol_REFL (hol''forall A)).

Definition hol__18779 := [A:hol_type]
  (Hol_MK_COMB (hol__18778 A) (hol__18777 ?)).

Definition hol__18780 := [A:hol_type]
  (Hol_EQ_MP (hol__18779 A) (hol__18774 ?)).
```

These Coq statements take approximately 100 bytes Coq source code per HOL step. So to represent the 1.5 million steps from the basic HOL Light library in this way would take a Coq file of about 150M.

If we check the type of `hol__18780` it indeed is the `Prop` that corresponds to `EXISTS_REFL`:

```
Coq < Check hol__18780.
hol__18780
     : (A:hol_type)
        (hol_thm
          (hol''forall A
            (hol_Abs A hol_'bool
              [a:(A)]
               (hol''exists A
                 (hol_Abs A hol_'bool [x:(A)](hol''eq A x a))))))
```

6 Encoded HOL data types versus Coq data types

We can translate HOL theorems about the HOL type `num` to Coq. These translations will be about the Coq type `(hol_elts (hol_'num))`. However the Coq type for the natural numbers is the Coq type `nat`. We will now show that how to define an isomorphism between `(hol_elts (hol_'num))` and `nat`.

It is straightforward to define a function `num_of_nat`:

```
Fixpoint num_of_nat [n:nat] : hol_'num :=
  Cases n of
    O => hol'_0
  | (S m) => (hol'SUC (num_of_nat m))
  end.
```

We show that this function is a surjection by proving:

```
nat_of_num_exists :
  (n:hol_'num)(Ex [m:nat](num_of_nat m)=n)
```

with induction on the Coq version of `num` using the translation of the HOL induction principle `num_INDUCTION`:

```
|- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> (!n. P n)
```

We use `nat_num_exists` to define an inverse

```
nat_of_num : hol_'num->nat
```

satisfying:

```
num_of_nat_of_num :
  (n:hol_'num)(num_of_nat (nat_of_num n))=n
```

Then using the translations of HOL's theorems `NOT_SUC` and `SUC_INJ`:

```
|- !n. ~(SUC n = 0)
|- !m n. (SUC m = SUC n) = (m = n)
```

it follows that this function is injective:

```
num_of_nat_injective :
  (m,n:nat)(num_of_nat m)=(num_of_nat n)->m=n
```

From that we derive that `nat_of_num` is the inverse to `num_of_nat`.

These bijections commute with the corresponding versions of the various functions on the natural numbers. For instance it commutes with the addition functions `hol''add` and `plus`. We prove:

```
plus_add : (m,n:nat)
  (num_of_nat (plus m n))=
    (hol''add (num_of_nat m) (num_of_nat n))

add_plus : (m,n:hol_'num)
  (nat_of_num (hol''add m n))=
    (plus (nat_of_num m) (nat_of_num n))
```

7 Conclusion

7.1 Discussion

This paper shows:

- It is straightforward to formalize the HOL Light semantics in Coq. This can be used as the basis of a proof converter that has the property that the converted HOL data types are isomorphic to their natural Coq counterparts.
- The conversion of the basic HOL Light library will take about 150M of Coq source code. This means that the approach from this paper will not give an efficient way to move mathematics from HOL to Coq.

The first is surprisingly nice but the second is discouraging.

7.2 Future work

There are several things that can be done to continue the work in this paper:

- The classical axioms from section 4.1:

  ```
  Axiom choice : (A:Set)~(A->False)->A.

  Axiom extensionality :
    (A,B:Set; f,g:A->B)((x:A)(f x)=(g x))->f=g.
  ```

  are not consistent with the Coq logic (the 'Calculus of Inductive Constructions'). It might be interesting to see whether the Coq people can be convinced to make their logic consistent with these axioms by removing the impredicativity of `Set`.
- We didn't automate the translation from HOL to Coq as outlined in section 5. It might be interesting to do this and investigate the performance difference between checking the basic HOL Light library in HOL Light and checking its translation in Coq.
- It might be interesting to restrict the translation to just translating the statements and omitting the proofs. That would be analogous to the approach of [8]. In such a way importing HOL Light mathematics in Coq might become realistic.
- We related the natural numbers of the two systems. It might be interesting to relate the counterparts of other data types.
- It might be interesting to work out a systematic way to prove the Coq analogue of a HOL Light statement from its direct translation. For instance the direct translation of HOL's theorem `ADD_SYM`:

  ```
  |- !m n. m + n = n + m
  ```

  is:

  ```
  (hol_thm (Hol''forall (Hol_Abs [m:hol_'num]
      (Hol''forall (Hol_Abs [n:hol_'num]
        (Hol''eq (hol''add m n) (hol''add n m)))))))
  ```

  However this is not meaningful to a Coq user. A Coq user wants to see:

  ```
  (m,n:nat)(plus m n)=(plus n m)
  ```

  This paper shows that it is possible to derive this from the previous statement. But it does not give a systematic description of what the structure of this kind of transformation is.

7.3 Acknowledgments

Thanks to Herman Geuvers and John Harrison for helpful discussions. Thanks to John Harrison for drawing my attention to [4] and [10].

References

1. Henk Barendregt. The impact of the lambda calculus. *Bulletin of Symbolic Logic*, 3(2), 1997.
2. Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Yann Coscoy, David Delahaye, Daniel de Rauglaudre, Jean-Christophe Filliâtre, Eduardo Giménez, Hugo Herbelin, Gérard Huet, Henri Laulhère, César Muñoz, Chetan Murthy, Catherine Parent-Virouroux, Patrick Loiseleur, Christine Paulin-Mohring, Amokrane Saïbi, and Benjamin Werner. *The Coq Proof Assistant Reference Manual*, 2000.
URL: `<ftp://ftp.inria.fr/INRIA/coq/V6.3.1/doc/Reference-Manual-all.ps.gz>`.
3. Robert L. Constable, Stuart F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, Douglas J. Howe, T.B. Knoblock, N.P. Mendler, P. Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, NJ, 1986.
4. Ewen Denney. A Prototype Proof Translator from HOl to Coq. In Mark Aagaard and John Harrison, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2000, Portland*, pages 108–125, Berlin, 2000. Springer-Verlag.
5. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL*. Cambridge University Press, Cambridge, 1993.
6. John Harrison. HOL Done Right.
URL: `<http://www.cl.cam.ac.uk/users/jrh/papers/holright.ps.gz>`, 1995.
7. John Harrison. *The HOL Light manual (1.1)*, 2000.
URL: `<http://www.cl.cam.ac.uk/users/jrh/hol-light/manual-1.1.ps.gz>`.
8. D. Howe. Importing mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics*, volume 1125 of *LNCS*, pages 267–282, Berlin, 1996. Springer-Verlag.
9. Matt Kaufmann, Panagiotis Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, Boston, 2000.
10. Pavel Naumov, Mark-Oliver Stehr, and José Mesequer. A Proof-Theoretic Approach to the HOL-Nuprl Connection with Applications to Proof Translation.
URL: `<http://www.csl.sri.com/users/stehr/holnuprl-ext.ps.gz>`, 2001.
11. S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752, Berlin, Heidelberg, New York, 1992. Springer-Verlag.
12. Lawrence C. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *LNCS*. Springer-Verlag, New York, 1994.
13. Henri Poincaré. *La Science et l'Hypothèse*. Flammarion, Paris, 1902.
14. Robert Pollack. How to Believe a Machine-Checked Proof. In G. Sambin and J. Smith, editors, *Twenty-Five Years of Constructive Type Theory*. Oxford University Press, Oxford, 1998.