Importing HOL Light into Coq

Chantal Keller, Benjamin Werner

Importing HOL Light into Coq

Chantal Keller[1,2] and Benjamin Werner[2]

[1] ENS Lyon
[2] INRIA Saclay–Île-de-France at
École polytechnique
Laboratoire d'informatique (LIX)
91128 Palaiseau Cedex
France
keller@lix.polytechnique.fr werner@lix.polytechnique.fr

Abstract. We present a new scheme to translate mathematical developments from HOL Light to Coq, where they can be re-used and re-checked. By relying on a carefully chosen embedding of Higher-Order Logic into Type Theory, we try to avoid some pitfalls of inter-operation between proof systems. In particular, our translation keeps the mathematical statements intelligible. This translation has been implemented and allows the importation of the HOL Light basic library into Coq.

1 Introduction

1.1 The curse of Babel?

Proof-systems are software dedicated to the development of mechanically checked formal mathematics. Each such system comes with its own logical formalism, its own mathematical language, its own proof language and proof format, its own libraries. A consequence is that it is largely impossible to reuse a formal development of one system in another, at least not without a re-implementation requiring important amounts of human work and skills.

This situation is about as old as proof-systems themselves, has often been deplored and is mostly felt as a modern form of the curse of Babel.

On the other hand, if the large number of human languages sometimes hinders comprehension between people, it also gives them a broader set of means to express themselves. It is well known that many subtleties of an original text are often "lost in translation". A similar point can be made in the case of formal mathematics: certain formalisms and systems can allow smoother developments of, say, real analysis, while others will be more at ease with fields involving large combinatorial case analyzes.

For these reasons, automatic translation between proof-systems is a tempting and old idea. It has, however, been hindered by various theoretical and practical obstacles. We here describe a new attempt that opens new possibilities. The ideas underlying this work are:

– We focus on one particular case, the translation from HOL Light to Coq.

– This work is specific to this case, and builds upon a careful study of the logical formalisms of both systems and the way they are implemented.
– In this particular case, we provide a *good* translation, in the following sense: the statements of the theorems translated from HOL Light remain intelligible and can be incorporated into further developments made in Coq.

1.2 Embedding Higher-Order Logic into Coq

As it is often the case when logic meets implementation, there are two aspects in this work:

1. The choice of logical embedding: in our case, statements and proofs of HOL Light have to be translated into counterparts in Coq's type theory. For instance, one often distinguishes between *deep* and *shallow* embeddings. The choice of this translation is central and will be discussed below.
2. The way to actually implement this translation. This will depend on issues like the way the two systems represent proofs, whether the translating function processes proof objects or proof scripts, etc...

Deep and shallow embeddings In order to represent terms from one logical framework A inside another formalism B, we have two possible ways:

– A *deep* embedding: define data-types in B that represent types and terms of A; we can then define, inside B, what it means to be provable in A.
– A *shallow* embedding: represent types and terms of A using their counterparts in B; this translation must preserve provability.

A deep embedding can ease things on the implementation side: we have access to the structure of the terms, and we can reason about them. Furthermore, the data types serve as a well-defined interface between the two systems.

However, our ultimate aim is to obtain actual Coq theorems[3]. For that, we concretely need a shallow embedding. In a previous work by F. Wiedijk [20], theorems from HOL Light were directly translated through a shallow encoding. Wiedijk observed that automation was difficult to perform that way. Furthermore, and although he used a shallow embedding, the theorems he obtained were still somewhat awkward in Coq. For instance, we would like to see the theorem $\forall a, \exists b, a = b$ to be translated in Coq in `forall (A:Type)(a:A), exists b, a = b` whereas Wiedijk obtains `forall (A:hol_type), hol_thm (hol''forall A (hol_Abs A hol''bool (fun (a:A)=> hol''exists A (hol_Abs hol''bool (fun (x:A)=> hol''eq A x a)))))`.

To avoid these pitfalls, we are going to obtain this shallow encoding going through a deep one. Frameworks like type theories allow to lift a deep embedding into a shallow one. This operation is called *reflection* from a proof theoretic point

[3] If John Harrison provides us with, say, Wiles' proof of Fermat's theorem in HOL Light, we want to translate it to the "real" Coq formulation of this famous result, and not the theorem "Fermat's theorem as stated in HOL is provable in HOL".

of view; through a proofs-as-programs perspective it can be understood as a compilation operation and corresponds to one of the two steps of *normalization by evaluation* (NbE) [7]. In this work we adapt a previous formalization of NbE in Coq [12].

To conclude, from HOL Light, we get deeply embedded terms; then we translate them into Coq theorems using a process similar to the computation part of normalization by evaluation.

Motivations and difficulties Embedding the Higher-Order Logic into Coq means defining a model of this logical framework, and so proving its *coherence*. The confidence we can have about the HOL Light theorem prover is thus increased. And this is enforced by the ability we have to check in Coq the theorems that were proved in HOL Light.

When translating theorems from HOL Light to Coq, we will have to take into account the differences between the two systems: as they often make different choices in the way a same mathematical corpus is formalized, one can therefore fear that a translated theorem is difficult to reuse in the target system, or that its statement becomes obscured.

Finally, one expects such a translation to be robust to changes in both proof-systems, and to be as efficient as possible in terms of time and memory consumption.

1.3 Related work

Interaction between HOL and Coq There already have been some attempts to share proofs between HOL systems and Coq, some of which are being developed now. One solution is to rely on an independent tool that will, at the same time, check the proof and perform the translation [9, 2]. Closer to our approach, it is conceivable to transform a theorem in the HOL system into a theorem in Coq, and check it in Coq, using its HOL proof as a guideline. F. Wiedijk [20] observed that directly using a shallow encoding was hard to automate and resulted in rather unintelligible theorems in Coq.

Recording and exporting HOL proofs Efficient systems to record and export HOL proofs have already been developed, both for the HOL prover [21] and for HOL Light [19, 17]. These works pursue the same motivations as ours: to import HOL proofs into a theorem prover. Since Obua's tool [19] is far more stable and easier for a direct exportation to Coq, we reused his code on the HOL Light side (proof recording), and changed the exportation side to a Coq format. These changes are now distributed with the development version of HOL Light[4].

[4] The development version of HOL Light is currently available at `http://hol-light.googlecode.com/svn/trunk`

1.4 Outlines

In the next section, we see the main characteristics of HOL Light and Coq that serve our work. Section 3 is dedicated to the major part of our work: constructing a deep embedding of the HOL Light logic into Coq, that is defining data structures that represent types, terms and derivations of HOL Light. To relate this to our main goal, we have to work on two levels:

– Upstream: we record and export HOL Light proofs into this encoding.
– Downstream: inside Coq, we construct a *lifting* from the deep embedding to a shallow embedding. Each HOL Light construction is mapped to its Coq counterpart. Besides, we prove that each time we have a Coq object representing a correct HOL Light derivation, its translation is a correct Coq theorem (Section 4).

This way, from a proof tree imported from HOL Light, we can reconstruct a theorem statement and its proof, using reflection.

We develop some aspects of the way to obtain reasonable performances in Section 5, thus obtaining the results described in Section 6. Finally, we discuss our approach and its perspectives.

2 HOL Light and Coq

HOL Light [15] and Coq [6] are two interactive theorem provers written in OCaml [18]. Although the ancestries of Coq and HOL Light can both be traced back to LCF, there are important differences, between the logical formalisms as well as in the way they are implemented. For obvious matters of space, we here do not give a complete description of the two systems, but underline differences which are crucial for the translation.

A more detailed comparison between HOL [14] and Coq has been established in [22] and enhanced in [9]. These studies also apply to HOL Light, which mainly differs from HOL by its smaller implementation.

2.1 The status of proofs

Proof systems like Coq and HOL Light share the same goal: to construct a formal proof. However, the status of these constructions is different in the two systems. This difference is directly related to the two formalisms.

HOL Light implements a variant of Church's Higher-Order Logic. A proof is a derivation in natural deduction. In the example below, Γ and Δ are sets of hypotheses; s, t, u and x are objects of some well chosen types:

$$\dfrac{\dfrac{\Gamma \vdash s = t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\,\text{TRANS} \qquad \dfrac{}{\vdash x = x}\,\text{REFL}}{\Gamma \cup \Delta \vdash s(x) = u(x)}\,\text{MK\_COMB}$$

While such a derivation has an obvious tree structure, it is not constructed as such in the system's memory. HOL Light represents statements like $\Gamma \vdash s = t$ as objects of an ML abstract data type `thm`. The fact that this data type is abstract is the key to the system's safety[5]: the only way to construct objects of type `thm` are well-understood *primitive tactics* corresponding to primitive inference rules. Thus, if a statement $\Gamma \vdash P$ can be represented in `thm`, it *is indeed* a theorem.

Coq implements a *Type Theory* where proofs are objects of the formalism[6]. More precisely, "being a proof of P" is identical with "being of type P", and as a consequence, the statements of the theory are of the form $\Gamma \vdash t : P$.

This results in a very different architecture and safety model:

- The type-checker is the safety critical part of the system, its *trusted computing base.*
- Theorems are constants: the statement is the type and the body is the proof. Proofs are thus kept in the system's memory and can be re-checked.

Consequences Since Coq keeps proof-objects while HOL Light does not, we need to build these objects at some point in the translation process. The translated proofs can have a critical size. We will see below that this requires special care.

2.2 Computations and equality

In both formalisms, the objects are strongly normalizing typed λ-terms. The way normalization is performed is very different however. Coq allows to type actual functional programs: its objects include a purely functional and terminating kernel of ML. This leads to some computation over terms, and a notion of *convertibility* between β-equivalent terms. In HOL Light, no computation is performed, and two β-equivalent terms are only *provably equal.*

Let us take an example.

In Coq, addition over unary natural numbers is defined as the usual program:

```
Fixpoint plus (n m : nat) : nat := match n
  with | O => m | S p => S (plus p m) end.
```

Such programs come with a notion of computation, which defines a notion of intentional equality. In this case, extended β-reduction yields (with some syntactic sugar) $2+2 \triangleright_\beta 4$ and thus $2+2 =_c 4$.

Like in all Martin-Lf type theories, these, possibly complex, computations are taken into account in the formalism by the specific *conversion rule*:

$$(\textsc{Conv})\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ (\text{if } A =_c B)$$

[5] And to the safety of related systems (LCF, HOL).

[6] This is one aspect of the *Curry-Howard isomorphism*; another aspect is that the Type Theory is constructive, but this point is less central here.

As a consequence, computations are omitted in the proof; for instance the propositions even$(2+2)$ and even(4) have exactly the same proofs.

Another consequence is that Coq and similar type theories need a notion of *propositional equality* which is distinct from the *computational relation* $=_c$.

In HOL Light, on the other hand, one will *prove* the existence of addition, that is of a function verifying the properties $0+x=x$ and $S(x)+y=S(x+y)$. The equality predicate is the way to construct atomic propositions, and all constructions ($\wedge, \vee, \Rightarrow$ connectors, quantifiers…) are defined through combinations of equality and, in some cases, the ε-operator.

HOL Light's equality has specific rules corresponding to reflexivity, transitivity, extentionality, β and η-equivalence.

Consequences On the one hand, in Coq, the length of a proof can be reduced by maximizing the computational part, which does not appear in proofs. This principle is called *computational reflection*. In our development, proofs of the theorems that are imported from HOL Light use these computational capabilities of Coq.

On the other hand, in HOL Light, the absence of computation leads to big proofs. The advantage is that we avoid the implementation of a β-reduction over our representation of HOL Light's λ-terms and a conversion rule, really simplifying *de facto* proof checking.

2.3 Treatment of constants

Constants, that is the ability to have definitions, are essential to any mathematical development. Precisely because of the different statuses of equality, constants are treated differently in the two systems.

In Coq, constants are treated through the computational equality and the conversion rule. Whenever a constant is defined:

```
Definition c : A := body.
```

a new object `c` of type `A` is added to the environment, and the computational equality is extended by `c` $=_c$ `body`.

In HOL Light, constant unfolding is explicitly handled by the equality predicate. The corresponding definition will also yield an object `c` of type `A`, together with a *theorem* stating that `c` $=$ `body`.

Consequences The fact that constants in HOL Light are unfolded only by invoking a specific theorem will turn out to be very convenient: it will allow us to map specific constants to previously defined Coq objects (see Section 4.3).

2.4 Classical logic

Whereas **Coq**'s intentional type theory is constructive, **HOL Light**'s logic is:

– Extensional: the η-equivalence is assumed (and functional extensionality can be deduced from it in HOL Light).
– Classical: propositions are actually boolean terms; thus propositional extensionality is true by definition; furthermore the axiom of choice is assumed via the introduction of a primitive Hilbert ε-operator.

Consequences Given the fact that the classical reasoning is deeply embedded in HOL Light logic, there seems to be no alternative but to assume corresponding axioms in Coq (excluded middle, choice operator, extentionality). Indeed, we need to apply them when proving the coherence of HOL Light in Coq.

2.5 Type definitions

In Coq, type definitions are just regular definitions or inductive definitions. **In HOL**, there is no primitive notion of induction, and since types have a different status than terms, type definitions have to be handled specifically.

A user can define new types in different ways involving facilities such as induction, but they all rely on one mechanism: the schema of specification. This is implemented by the primitive rule `new_basic_type_definition`, which, given a property $P : A \to \text{bool}$, a term $x : A$ such that $\vdash P\ x$ holds, defines the type $B = \{y : A | P\ y\}$, and two constants (in the sense of section 2.3): the canonical injection from B to A and the injection from A to B whose behavior is specified only for elements y such that $P\ y$ holds.

Consequences All the types in HOL Light are inhabited, since the base types are inhabited and a type definition has to be inhabited because we require $\vdash P\ x$. This is useful to define the ε operator in a proper way without a condition of non-emptiness on the type.

3 Deep embedding: representing Higher-Order Logic in Coq

In this section, we represent Higher-Order Logic in Coq. The main originality is the computational definition of deduction at the end of the section. The basic definitions are quite standard, but some care is needed in order to make the later developments tractable and to keep memory consumption as low as possible as well as the computations efficient enough. In particular, it is mandatory to have explicit definitions.

Our encoding uses the SSReflect tactics and libraries package developed by G. Gonthier et al. [13], but the level of details of this paper does not allow us to make clear how crucial this is.

3.1 Types and terms

Names We need names for variables and definitions both for types and terms. These names have to be in an infinite countable set over which equality is decidable. For efficiency reasons, we chose `positive`, the Coq representation of binary natural numbers greater than 1.

In the rest of the paper, `idT`, `defT`, `idV` and `defV`, respectively representing types variables, types constants, terms variables and terms constants, stand for `positive`.

Types For types, we need variables and definitions, and we give a specific status to the two primitive definitions of HOL Light: `bool` and the functional arrow. A previously defined type is accessed through its name and the list of its arguments.

```
Inductive type : Type :=
| TVar : idT → type | Bool : type
| Arrow : type → type → type
| TDef : defT → list_type → type
with list_type : Type :=
| Tnil : list_type
| Tcons : type → list_type → list_type.
```

Terms Terms are defined straightforwardly as an inductive type. For bound variables, we use a locally nameless representation [5], which is simpler to reason about than a named representation. As for types, we distinguish some primitive term definitions: the equality, the ε choice operator, and the logical connectives. We group them together under the type `cst`, and obtain for terms this definition:

```
Inductive term : Type :=
| Dbr : nat → term | Var : idV → type → term
| Cst : cst → term | Def : defV → type → term
| App : term → term → term
| Abs : type → term → term.
```

Typing This is our first use of computational reflection. Rather than defining typing judgments as an inductive relation, we directly define [12] a typing algorithm. Given a De Bruijn `context` (a list of types) `g` and a term `t`, the function `infer` returns the type of `t` under the context `g` if it exists, and fails otherwise. `wt g t A` means that this term has type `A` under `g`, and is just a shortcut for "`infer g t` returns `A`". Since we consider simply typed terms, the definition of `infer` is easy.

3.2 Derivations

We now define HOL Light's logical framework. Typically, Coq's induction process and dependant types are well-suited to represent such judgments. Here is an extract of the inductive data-type that represents HOL Light derivations):

```
Inductive deriv : hyp_set → term → Prop :=
| Drefl : forall t A, wt nil t A → deriv
    hyp_empty (heq A t t)
| Dconj: forall h1 h2 a b, deriv h1 a → deriv
    h2 b → deriv (hyp_union h1 h2) (a hand b)
| Dconj1: forall h a b, deriv h (a hand b) →
    deriv h a
| Dconj2: forall h a b, deriv h (a hand b) →
    deriv h b
| ...
```

It would however be impractical to have HOL Light generate such derivations. They would be too verbose and difficult to build (for `Drefl`, how to prove `wt nil t A`?). Obua [19] notices that it is sufficient to build a far more compact *skeleton* of the tree. This skeleton carries only minimal information; typically which inference rules have been used.

Following his code, we record proofs in HOL Light using an ML recursive type that represents this skeleton (which means this structure uses no dependent types anymore). We then export it straightforwardly into its twin Coq inductive type:

```
Inductive proof : Type :=
| Prefl : term → proof
| Pconj : proof → proof → proof
| Pconjunct1 : proof → proof
| Pconjunct2 : proof → proof
| ...
```

These twin OCaml and Coq types thus establish the bridge between HOL Light and Coq.

This structure is typed too loosely to guarantee the correctness of the derivation. Some objects of type `proof` do not correspond to actual proofs. For instance, `Pconjunct1` stands for the elimination rule of $\wedge$ on the left. Its argument should be a proof of a theorem that looks like $\Gamma \vdash A \wedge B$, but this is not enforced in the `proof` inductive type.

However, a skeleton is sufficient to reconstruct a complete derivation when it exists, by making the necessary verifications. This is the job of the `proof2deriv` function:

– If `p` is a well-formed proof, then `proof2deriv p` returns `h` and `t`, and we can establish that `deriv h t` stands (this is lemma `proof2deriv_correct`).
– If not, `proof2deriv` fails.

Once more computational reflection is used to deduce `deriv h t` from a `proof`.

4 Going from the deep embedding to Coq terms

In the previous section, we show how to export HOL Light proofs and obtain in Coq objects of type `deriv h t` for a certain `h` and `t`, that is to say deeply written theorems. To interact with Coq theorems, we want to have a shallow reading of these theorems. That is why we define a translation from deep to shallow.

This translation has already been implemented in [12] for a simply typed λ-calculus with only named variables. Here it is trickier with De Bruijn indices and definitions.

4.1 General idea

For the moment, we suppose given an interpretation function $\mathcal{I}$ to interpret variables and definitions names. Its meaning is detailed in Section 4.3.

We first define $[\bullet]_{\mathcal{I}}$, a translation function on types, that maps, for instance, `Bool` to `Prop` and the arrow to the arrow of Coq (see its precise typing below). We then define $|\bullet|_{\mathcal{I}}$, a translation function on terms, that respects their types. Informally, it means that when a term t has type T, then $|t|_{\mathcal{I}}$ belongs to $[T]_{\mathcal{I}}$:

$$\text{if } \Gamma \vdash t : T \text{ then } |t|_{\mathcal{I}} \in [T]_{\mathcal{I}}$$

4.2 Implementation

It is important for all the types in HOL Light to be inhabited, as we noticed in Section 2.5. We now cope with this by stating that the translation of a type must be an inhabited `Type`. Inhabited `Type`s can be represented by a record:

```
Record type_translation : Type :=
  mkTT {ttrans :> Type; tinhab : ttrans}.
```

The translation function $[\bullet]_{\mathcal{I}}$, a.k.a. `tr_type`, maps a HOL type to a `type_translation`:

```
tr_type: forall 𝓘, type → type_translation
```

The translation function $|\bullet|_{\mathcal{I}}$, a.k.a. `sem_term`, is as a refinement of the typing function `infer` (see Section 3.1). In addition to typing a term, it gives its Coq translation (note that its definition is eased by the use of dependant types):

```
sem_term : context → term →
  option {ty: type & forall 𝓘, tr_type 𝓘 ty}
```

4.3 The interpretation functions

Variables We have three kinds of variables to interpret: type variables, named term variables and De Bruijn indices. We map each of them to Coq objects with respect to their types.

Definitions The interpretation of definitions is a key point to preserve the intelligibility of theorems. Interpretation for types and terms definitions are respectively defined by:

```
Definition type_definition_tr := defT →
  list type_translation → type_translation.
Definition term_definition_tr :=
  defV → forall (A: type), tr_type 𝓘 A.
```

Imagine we have a HOL Light term `t` bringing into play objects of type `num`, the type for natural numbers defined with zero (`_0`) and successor (`SUC`). If we apply `sem_term` to `t` with an object of type `type_definition_tr` that maps `(num,[])` to `nat` and a object of type `term_definition_tr` that maps `(_0,num)` to `0` and `(SUC,num→num)` to `S`, we obtain a Coq term that corresponds to `t`, but it is a standard Coq theorem, and it would have been written that way directly in Coq.

Besides, this process is fully *flexible*: if instead of `nat`, we map `num` to `N`, the Coq type for binary natural numbers, then we get the same term in this different formalism of naturals.

Notation In the remaining of the article, the interpretation functions are still abbreviated as $\mathcal{I}$.

4.4 Adequacy of derivations w.r.t. semantics

We can now establish that this translation applied to correct HOL Light derivations produce correct Coq theorems. By a correct Coq theorem, we mean a term that is locally closed, has type `Bool`, and whose translation is a correct Coq proposition whatever the interpretation functions might be:

```
Definition has_sem (t: term) : Prop :=
  match sem_term nil t with
    | Some (existT Bool evT) ⇒ forall 𝓘, evT 𝓘
    | _ ⇒ False end.
```

We can establish the following theorem:

```
Theorem sem_deriv : forall (h: hyp_set) (t:
    term), deriv h t → forall 𝓘, sem_hyp 𝓘 h →
    has_sem 𝓘 t.
```

where `sem_hyp` $\mathcal{I}$ `h` checks that `has_sem` $\mathcal{I}$ holds for each term of `h`. Since `has_sem` is a function, the proofs of the translated theorems are computationally reflexive again.

Conclusion of parts 3 and 4 When a theorem is being proved in HOL Light, its proof is recorded. It can be exported as an object `p: proof` that we expect to be correct (because it comes from a HOL Light proof). Given this object, we define a set of hypotheses h and a conclusion t with proof2deriv. proof2deriv_correct gives a computationally reflexive proof that these objects correspond to a derivation. If h is empty, we can apply has_sem to t, to get the Coq version of this theorem (which is very close to one would have written directly in Coq), and we have a computationally reflexive proof of this theorem applying sem_deriv.

5 Improving efficiency

As such this process allows, in principle, to import HOL Light theorems and to check them. . . In practice, it would take too much time and use too much memory. We stopped when the exported files reached 200 Gb; Obua reports a similar experience [19].

5.1 Sharing

Proofs Obua uses the natural sharing for proofs that comes from the user: when a piece of proof is used more than twice, it is shared. This sharing is not optimal, and it depends on the user's implementation, but it is very simple, it does not need too much memory (using a hash-table to perform hash-consing on proofs would really increase the memory consumption), and it is sufficient to considerably reduce the size of the exported files.

In Coq, this sharing is kept by adding one constructor to the inductive type proof:

```
Inductive proof : Type :=
|...| Poracle : forall h t, deriv h t → proof.
```

For each proof coming from HOL Light, the corresponding derivation is immediately computed. It can be called in a following proof thanks to the constructor Poracle.

Types and terms We share types and terms using standard hash-consing presented in [10].

5.2 Opaque proofs

In Coq, even with sharing, objects of type proof can be arbitrary big. Our idea to avoid keeping them in memory is to:

- distribute the theorems coming from HOL Light into separate files, and when compiling the $(n+1)^{\text{th}}$ file, load only the statements of the theorems of the first n files, but not the opaque proofs (this can be done with the option -dont-load-proofs of Coq's compiler);
- put the objects of type proof inside Coq opaque proofs.

5.3 Computation

In addition to its internal reduction mechanism, Coq includes an optimized bytecode-based virtual machine to evaluate terms. It is less tunable, but rather more efficient that the internal mechanism. It is well suited for the full evaluation required by computational reflection.

6 Results

6.1 Implementation

Our implementation is free software and can be found online [1]. Looking back at the two main objectives of this work, efficiency and usability, we observe some limitations to the first goal, while the second one is rather fulfilled.

6.2 Tests

The tests were realized on a virtual machine that is installed on a DELL server PowerEdge 2950, with 2 processors Intel Xeon E5405 (quad-core, 2GHz) and 8 Gb RAM, with CentOS (64 bits). We are able to import and to check in Coq HOL Light proofs from:

- The standard library: what is loaded by default when launching HOL Light.
- The Model directory: a proof of consistency and soundness of HOL Light in HOL Light [16] (which, again, enforces the confidence in HOL Light).
- The elementary linear algebra tools developed in `Multivariate/vectors.ml`.

The results are summed up in **Table 1**. For each benchmark, we report the number of theorems that were exported, the number of lemmas generated by sharing, the time to interpret theorems and record their proofs in HOL Light, the time to export theorems to Coq, the time of compilation in Coq, the size of the generated Coq files, the maximal virtual memory used by OCaml, and the maximal virtual memory used by Coq. In the next two paragraphs, we analyze the origins of compilation time and memory consumption, and present some possible solutions.

Time and memory in Coq Our proof sharing system has the major drawback to lead to a complete blow-up of the number of exported statements, as the first two columns of **Table 1** attest. Moreover, all these statements need to be kept in memory because all the theorems depend on one another.

The time of Coq's compilation thus cannot be less than quadratic in the number of Coq files, since compiling the $(n+1)^{\text{th}}$ file imports files 1 to n. The other operations that are expensive in time are:

- the parsing of the proof objects;
- the evaluation of the computationally reflexive proofs.

Bench.	Number		Time			Memory		
	Theorems	Lemmas	Rec.	Exp.	Comp.	H.D.D.	Virt. OCaml	Virt. Coq
Stdlib	1,726	195,317	2 min 30	6 min 30	10h	218 Mb	1.8 Gb	4.5 Gb
Model	2,121	322,428	6 min 30	29 min	44h	372 Mb	5.0 Gb	7.6 Gb
Vectors	2,606	338,087	6 min 30	21 min	39h	329 Mb	3.0 Gb	7.5 Gb

Table 1. Benchmarking the standard library and Model.

Concerning this last item, it is important to notice that Coq's virtual machine can run such a big computation. Computational reflection sounds thus a good way to import proofs, at least in Coq.

In other words: sharing limits the memory consumed by *proof objects*, but the resulting number of *statements* then becomes a problem. The compilation time is not too restrictive, since the incoming theorems have to be compiled once for all. Moreover, it requires far less human work to automatically export some theorems and compile it with our tool than to prove them from scratch in Coq. Memory is a large limitation for a user though, since he or she needs to import all the Coq files even to use only the last theorem. It would be convenient to be able not to load the intermediary lemmas, but it does not seem possible with our present proof objects implementation.

Memory in OCaml The fact that proofs are kept is not the only limiting factor for OCaml's memory: during exportation, we create big hash-tables to perform hash-consing and to remember theorem statements. If we keep the present proof format, we definitely would have to reduce the extra-objects we construct for exportation.

Conclusion Now that we have something reliable for a rather simple proof format, a next step is to switch to a more compact format such as the ones presented in [8] or [17].

6.3 Usability

We now give an example from integer arithmetic of an interaction between HOL Light and Coq's theorems. We map HOL Light's definitions as stated in **Table 2**.

Given the usual notation to say "a divides b":

```
Notation "a|b" := (Nmod b a = 0).
```

we import the theorem MOD_EQ_0 from HOL Light:

HOL Light	num	+	*	DIV	MOD
Coq	N	Nplus	Nmult	Ndiv	Nmod

Table 2. Mapping of definitions. In Coq, `N` is the type of binary natural numbers, and is defined in `NArith`. In HOL Light, `num` is the type of unary natural numbers.

```
Theorem hollight_MOD_EQ_0_thm :
  forall x x0 : N, x0 <> 0 →
    x0 | x = (exists a : N, x = a * x0).
```

and combine it with one Coq's theorem using tactics to prove:

```
Lemma div_mult : forall a b, a <> 0 → a | b →
    forall k, a | k*b.
```

The proof is only five lines long, because it is straightforward from `hollight_MOD_EQ_0_thm`. As Coq's standard library does not have any lemmas about division and modulo in `N`, proving this statement from scratch would certainly not be trivial.

7 Conclusion and future work

The new way to translate theorems and proofs from HOL Light to Coq presented in this paper fills the gap between those two interactive theorem provers. We solve both theoretical and practical problems coming from the different frameworks the two provers are based on. Relying on the computational power of Coq (reflection), our translation is both able to restore theorems meanings in Coq and give a small proof of them.

The implementation scales up to non trivial libraries. We are limited by usual performance issues, but at a much later point. To manage to import even larger developments, like the geometric properties used in the Flyspeck project [3], we need to reduce compilation time and virtual memory used. This may be possible by improving the proof format and changing the way we perform sharing.

Possible future directions include:

- translating HOL proofs to other systems with rich computation capabilities;
- integrating other external tools in Coq, such as other interactive theorem provers or automatic theorem provers, without compromising its soundness.

Acknowledgments We are particularly grateful to Carlos Simpson who suggested the use of normalization by evaluation for translating HOL proofs to Coq and provided the stimulus for this work. We also thank John Harrison and Steven Obua who provided useful help with the code of HOL Light. We finally thank the anonymous reviewers for their useful remarks.

References

1. Our implementation, http://perso.ens-lyon.fr/chantal.keller/Recherche/hollightcoq.html
2. Dedukti, a universal proof checker, http://www.lix.polytechnique.fr/dedukti
3. The Flyspeck project, http://www.flyspeck-blog.blogspot.com
4. Aagaard, M., Harrison, J. (eds.): Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings, Lecture Notes in Computer Science, vol. 1869. Springer (2000)
5. Aydemir, B., Charguéraud, A., Pierce, B., Pollack, R., Weirich, S.: Engineering formal metatheory. In: Necula, G., Wadler, P. (eds.) POPL. pp. 3–15. ACM (2008)
6. Barras, B., Boutin, S., Cornes, C., Courant, J., Filliatre, J., Gimenez, E., Herbelin, H., Huet, G., Munoz, C., Murthy, C., et al.: The Coq proof assistant: reference manual. Rapport technique - INRIA (2000)
7. Berger, U., Schwichtenberg, H.: An inverse of the evaluation functional for typed lambda-calculus. In: LICS. pp. 203–211. IEEE Computer Society (1991)
8. Berghofer, S., Nipkow, T.: Proof terms for simply typed higher order logic. In: Aagaard and Harrison [4], pp. 38–52
9. Denney, E.: A prototype proof translator from hol to coq. In: Aagaard and Harrison [4], pp. 108–125
10. Filliâtre, J.C., Conchon, S.: Type-safe modular hash-consing. In: Kennedy, A., Pottier, F. (eds.) ML. pp. 12–19. ACM (2006)
11. Furbach, U., Shankar, N. (eds.): Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings, Lecture Notes in Computer Science, vol. 4130. Springer (2006)
12. Garillot, F., Werner, B.: Simple types in type theory: Deep and shallow encodings. In: Schneider, K., Brandt, J. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 4732, pp. 368–382. Springer (2007)
13. Gonthier, G., Mahboubi, A.: A Small Scale Reflection Extension for the Coq system. Tech. rep., INRIA (2007)
14. Gordon, M., Melham, T.: Introduction to HOL: A theorem proving environment for higher order logic. Cambridge University Press New York (1993)
15. Harrison, J.: HOL Light: A tutorial introduction. In: Srivas, M., Camilleri, A. (eds.) FMCAD. Lecture Notes in Computer Science, vol. 1166, pp. 265–269. Springer (1996)
16. Harrison, J.: Towards self-verification of HOL Light. In: Furbach and Shankar [11], pp. 177–191
17. Hurd, J.: OpenTheory: Package Management for Higher Order Logic Theories. PLMMS09 p. 31 (2009)
18. Leroy, X.: The OCaml Programming Language (1998)
19. Obua, S., Skalberg, S.: Importing HOL into Isabelle/HOL. In: Furbach and Shankar [11], pp. 298–302
20. Wiedijk, F.: Encoding the HOL Light logic in Coq. Unpublished notes (2007)
21. Wong, W.: Recording and checking HOL proofs. In: Schubert, E., Windley, P., Alves-Foss, J. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 971, pp. 353–368. Springer (1995)
22. Zammit, V.: A comparative study of Coq and HOL. In: Gunter, E., Felty, A. (eds.) TPHOLs. Lecture Notes in Computer Science, vol. 1275, pp. 323–337. Springer (1997)