Mixing HOL and Coq in Dedukti (Rough Diamond)

Ali Assaf, Raphaël Cauderlier

Mixing HOL and Coq in Dedukti (Rough Diamond)

Ali Assaf[1,2] and Raphaël Cauderlier[3]

[1] INRIA Paris-Rocquencourt, France
[2] École Polytechnique, France
[3] Laboratoire CEDRIC, CNAM, France

Abstract. We use Dedukti as a logical framework for interoperability. We use automated tools to translate different developments made in HOL and in Coq to Dedukti and we combine them to prove new results.

1 Intro

Interoperability is an emerging problem in the world of proof systems. Whether they are interactive or automated, theorem provers are developed independently and cannot usually be used together effectively. The theorems of one system can rarely be used in another, and it can be very expensive to redo the proofs manually. The obstacles are many, ranging from differences in the logical theory and in the representation of data types, to the lack of a standard and effective way of retrieving proofs.

One solution to this problem is to use a logical framework like Dedukti [8]. The idea is to have a language that is expressive and flexible enough to define various logics and to faithfully express proofs in those logics, at a relatively low cost. Translating all the different systems to this common framework is a first step in bringing them closer together.

Several tools have been developed [3,2,5,4] to translate the proofs of various systems to Dedukti. The proofs, represented as terms of the $\lambda\Pi$-calculus modulo rewriting, can be checked independently by Dedukti, adding another layer of confidence over the original systems. This approach has been successfully used to verify the formalization of several libraries and the proof traces of theorem provers on large problem sets.

In this paper, we take one step further and show that we can combine the proofs coming from different systems in this same framework. A theorem can therefore be split into smaller blocks to be proved separately using different systems, and large libraries formalized in one system can be reused for the benefit of developments made in another one.

We used Holide and Coqine to translate proofs of HOL and Coq respectively to Dedukti. We examined the logical theories behind those two systems to determine how we can combine them in a single unified theory while addressing the problems mentioned above. Finally, we used the resulting theory to certify the correctness of a sorting algorithm involving Coq lists of HOL natural numbers[4].

[4] Our code is available online at `http://dedukti-interop.gforge.inria.fr/`.

2 Tools used

Dedukti

Dedukti[5] is a functional language with dependent types based on the $\lambda\Pi$-calculus modulo rewriting [8,9]. The type-checker/interpreter for Dedukti is called `dkcheck`. It accepts files written in the Dedukti format (`.dk`) containing declarations, definitions, and rewrite rules, and checks whether they are well-typed.

Following the LF tradition, Dedukti acts as a logical framework to define logics and express proofs in those logics. The approach consists in representing propositions as types and proofs as terms inhabiting those types, as in the Curry-Howard correspondence. Assuming the representation is correct, a proof is valid if and only if its corresponding proof term is well-typed. That way we can use Dedukti as an independent proof checker.

Holide

Holide[6] translates HOL proofs to the Dedukti language. It accepts proofs in the OpenTheory format (`.art`) [6], and generates files in the Dedukti format (`.dk`). These files can then be verified by Dedukti to check that the proofs are indeed valid. The translation is described in details in [2].

The generated files depend on a handwritten file called `hol.dk`. This file describes the theory of HOL, that is the types, the terms, and the derivation rules of HOL. The types of HOL are those of the simply-typed λ-calculus. The propositions are the terms of type bool.

$$\begin{array}{ll} \mathsf{type} : \mathsf{Type}. & \mathsf{term} : \mathsf{type} \to \mathsf{Type}. \\ \mathsf{bool} : \mathsf{type}. & \mathsf{proof} : \mathsf{term}\ \mathsf{bool} \to \mathsf{type}. \\ \mathsf{arrow} : \mathsf{type} \to \mathsf{type} \to \mathsf{type}. & \ldots \end{array}$$

Coqine

Coqine[7] translates Coq proofs to the Dedukti language. It takes the form of a Coq plugin that can be called to export loaded libraries (`.vo`) to generate files in the Dedukti format (`.dk`). These files can then be verified by Dedukti to check that the proofs are indeed valid.

A previous version of the translation is described in [3]. However, that translation is outdated, as it does not support the universe hierarchy and universe subtyping of Coq. The translation has since been updated to support both features following the ideas in [1].

The generated files depend on a handwritten file describing the theory of the calculus of inductive constructions (CIC) called `coq.dk`. There is a type prop

[5] Available at: `http://dedukti.gforge.inria.fr/`
[6] Available at: `https://www.rocq.inria.fr/deducteam/Holide/`
[7] Available at: `http://www.ensiie.fr/~guillaume.burel/blackandwhite_coqInE/`

that represents the universe of propositions and a type $\mathsf{type}\ i$ for every natural number i that represents the i-th universe of types. We will write type_i and term_i for $\mathsf{type}\ i$ and $\mathsf{term}\ i$ respectively.

$$\begin{array}{ll} \mathsf{type} : \mathsf{nat} \to \mathsf{Type}. & \mathsf{term}\ : \Pi i : \mathsf{nat}.\ \mathsf{type}\ i \to \mathsf{Type}. \\ \mathsf{prop} : \mathsf{Type}. & \mathsf{proof} : \mathsf{prop} \to \mathsf{Type}. \\ \ldots & \end{array}$$

3 Mixing HOL and Coq

HOL and Coq use very different logical theories. The first is based on Church's simple type theory, is implemented using the LCF approach, and its proofs are built by combining sequents in a bottom-up fashion. The second is based on the calculus of inductive constructions and checks proofs represented as lambda-terms in a top-down fashion. Translating these two systems to Dedukti was a first step to bringing them closer together but there are still important differences that sets them apart. In this section, we examine these differences and show how we were able to bridge these gaps.

Type inhabitation

The notion of types is different between HOL and Coq. In HOL, types are those of the simply-typed λ-calculus where every type is inhabited. In contrast, Coq allows the definition of empty types, which in fact play an important role as they are used to represent falsehood. A naïve reunion of the two theories would therefore be inconsistent: the formula $\exists x : \alpha, \top$, where α is a free type variable, is provable in HOL but its negation $\neg\forall\alpha : \mathsf{Type}, \exists x : \alpha, \top$ is provable in Coq.

Instead, we match the notion of HOL types with that of Coq's *inhabited* types, as done in [7]. We define inhabited types in the Coq module `holtypes`:

```
Inductive type : Type := inhabited : forall (A : Type), A -> type.
```

It is then easy to prove in Coq that given inhabited types A and B, the arrow type $A \to B$ is also inhabited:

```
Definition carrier (A : type) : Type :=
  match A with inhabited B b => B end.

Definition witness (A : type) : carrier A :=
  match A with inhabited B b => b end.

Definition arrow (A : type) (B : type) : type :=
  inhabited (carrier A -> carrier B) (fun _ => witness B).
```

This is all that we need to interpret hol.type, hol.term, and hol.arrow:

$$\begin{array}{ll} \mathsf{hol.type} & \rightsquigarrow \mathsf{coq.term}_1\ \mathsf{holtypes.type}. \\ \mathsf{hol.arrow}\ a\ b & \rightsquigarrow \mathsf{holtypes.arrow}\ a\ b. \\ \mathsf{hol.term}\ a & \rightsquigarrow \mathsf{coq.term}_1\ (\mathsf{holtypes.carrier}\ a). \end{array}$$

Booleans and propositions

In Coq, there is a clear distinction between booleans and propositions. Booleans are defined as an inductive type bool with two constructors true and false. The type bool lives in the universe Set. In contrast, following the Curry-Howard correspondence, propositions are represented as types with proofs as their inhabitants. These types live in the universe Prop. Both Set and Prop live in the universe Type_1. As a consequence, Prop is not on the same level as other types such as bool or nat (the type of natural numbers), a notorious feature of the calculus of constructions. Moreover, since Coq is an intuitionistic system, there is no bijection between booleans and propositions. The excluded middle does not hold, though it can be assumed as an axiom.

In HOL, there is no distinction between booleans and propositions and they are both represented as a single type bool. Because the system is classical, it can be proved that there are only two inhabitants $\top$ and $\bot$, hence the name. Moreover, the type bool is just another simple type and lives on the same level as other types such as nat.

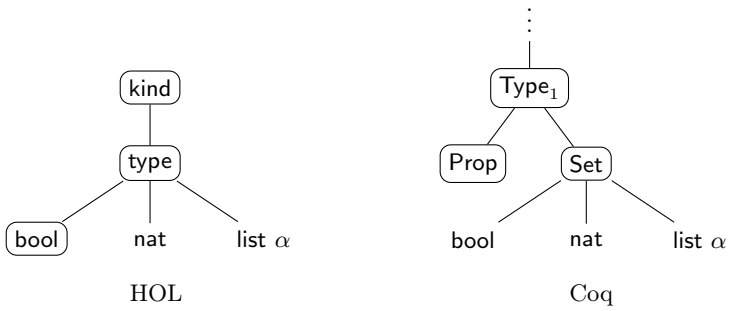

Fig. 1. Booleans and propositions in HOL and Coq. Boxes represent universes.

To combine the two theories, one must therefore reconcile the two pictures in Figure 1. One solution is to interpret the types of HOL as types in Set. To do this, we must rely on a reflection mechanism that interprets booleans as propositions, so that we can retrieve the theorems of HOL and interpret them as theorems in Coq. In our case, it consists of a function istrue of type hol.bool $\to$ coq.prop, which we use to define hol.proof.

$$\mathsf{hol.proof}\ b \rightsquigarrow \mathsf{coq.proof}\ (\mathsf{istrue}\ b).$$

Another solution is to translate hol.bool as coq.prop. To do this, we must therefore translate the types of HOL as types in Type_1 instead of Type_0. In particular, if we want to identify hol.nat and coq.nat, we must have coq.nat in Type_1. Fortunately, we have this for free with cumulativity since any element of Type_0 is also an element of Type_1.

We choose the first approach as it is more flexible and places less restrictions (e.g. regarding Prop elimination in Coq) on what we can do with booleans. In particular, it allows us to build lists by case analysis on booleans, which is needed in the sorting algorithm.

4 A concrete example: sorting Coq lists of HOL numbers

We proved in Coq the correction of the insertion sort algorithm on polymorphic lists and we instantiated it with the canonical ordering of natural numbers defined in HOL. More precisely, on the Coq side, we defined polymorphic lists, the insertion sort function, the sorted predicate, and the permutation relation. We then proved the following two theorems:

```
Theorem sorted_insertion_sort: forall l, sorted (insertion_sort l).
Theorem perm_insertion_sort: forall l, perm l (insertion_sort l).
```

On the HOL side, we used booleans, natural numbers and the ordering relation on natural number as defined in the OpenTheory packages `bool.art` and `natural.art`. By composing the results, we obtain two theorems:

$$\Pi l : \mathsf{coq.term}_1\ (\mathsf{coq\_list\ hol\_nat}).\ \mathsf{proof}\ (\mathsf{sorted}\ (\mathsf{insertion\_sort\ compare}\ l)).$$
$$\Pi l : \mathsf{coq.term}_1\ (\mathsf{coq\_list\ hol\_nat}).\ \mathsf{proof}\ (\mathsf{perm}\ l\ (\mathsf{insertion\_sort\ compare}\ l)).$$

where compare is the translation of HOL comparison to Coq booleans.

The composition takes place in a Dedukti file named `interop.dk`. This files takes care of matching the interfaces of the proofs coming from Coq with the proofs coming from HOL. Most of the work went into proving that HOL's comparison is indeed a total order in Coq:

$$\Pi m\ n : \mathsf{nat}.\ \mathsf{if}\ (\mathsf{compare}\ m\ n)\ \mathsf{then}\ m \leq n\ \mathsf{else}\ n \leq m.$$

where $\mathsf{nat} := \mathsf{holtypes.carrier\ hol\_nat}$. We prove it using the following theorems from OpenTheory:

$$\Pi m\ n : \mathsf{hol\_nat}.\ m < n \to m \leq n$$
$$\Pi m\ n : \mathsf{hol\_nat}.\ m \not\leq n \leftrightarrow n < m$$

and some lemmas on if ... then ... else.

We chose this example because the interaction between Coq and HOL types is very limited thanks to polymorphism: there is no need to reason about HOL natural numbers on the Coq side and no need to reason about lists on the HOL side so the only interaction takes place at the level of booleans which we wanted to study. The resulting implementation is illustrated in Figure 2. All components were successfully verified by Dedukti.

5 Limitations

Scalability Our experiment, while successful, required a lot of manual tinkering. We attribute this to some technical limitations of the tools we used, which should be addressed before using this approach on a larger scale:

- The translations produce code intended for machines that is not very usable by humans. The linking of theories together should therefore either be more automated or benefit from a more readable output.
- The current implementation of Coqine does not support modules and universe polymorphism. We must therefore redefine some types that are in the standard library to avoid using these features.

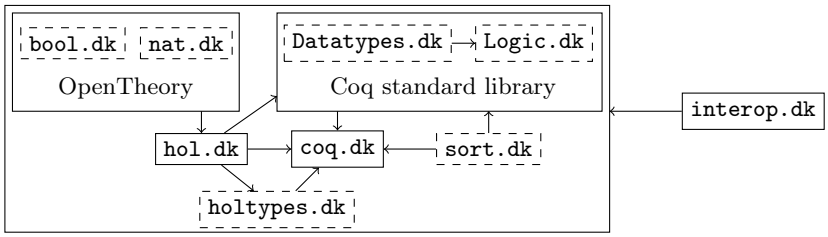

Fig. 2. Components of the implementation. Solid frames represent source files. Dashed frames represent automatically generated files. Arrows represent dependencies.

Executability Even though we have constructed a sorting "algorithm" on lists of HOL natural numbers and we have proved it correct, there is no way to actually execute this algorithm. Indeed, there is no notion of computation in HOL, so when the sorting algorithm asks compare for a comparison between two numbers, it will not return something which will unblock the computation. Therefore, insertion_sort $[4, 1, 3, 2]$ is not *computationally* equal to $[1, 2, 3, 4]$. However, the result is still *provably* equal to what is expected: we can show that insertion_sort $[4, 1, 3, 2]$ is equal to $[1, 2, 3, 4]$. A constructivization of HOL will be necessary before we can obtain truly executable code. Holide is a good starting point for such a project.

References

1. Ali Assaf. A calculus of constructions with explicit subtyping. Submitted to Postproceedings of Types 2014, available at `https://hal.inria.fr/hal-01097401`, 2014.
2. Ali Assaf and Guillaume Burel. Translating HOL to Dedukti. Draft, available at `https://hal.inria.fr/hal-01097412`, 2014.
3. Mathieu Boespflug and Guillaume Burel. CoqInE: Translating the calculus of inductive constructions into the λΠ-calculus modulo. In *PxTP*, page 44, 2012.
4. Guillaume Burel. A shallow embedding of resolution and superposition proofs into the λΠ-calculus modulo. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP*, volume 14 of *PxTP 2013*, pages 43–57. EasyChair, June 2013.
5. David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand, and Olivier Hermant. Zenon modulo: When Achilles outruns the tortoise using deduction modulo. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *LPAR*, number 8312 in LNCS, pages 274–290. Springer Berlin Heidelberg, 2013.
6. Joe Hurd. The OpenTheory Standard Theory Library. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NFM*, number 6617 in LNCS, pages 177–191. Springer, 2011.
7. Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP*, number 6172 in LNCS, pages 307–322. Springer Berlin Heidelberg, 2010.
8. Ronan Saillard. Dedukti: a universal proof checker. In *Foundation of Mathematics for Computer-Aided Formalization Workshop*, 2013.
9. Ronan Saillard. Towards explicit rewrite rules in the λΠ-calculus modulo. In *IWIL - 10th International Workshop on the Implementation of Logics*, 2013.