Mixing HOL and Coq in Dedukti
(Extended Abstract)

Ali Assaf
INRIA Paris-Rocquencourt, France
École Polytechnique, France

Raphaël Cauderlier
INRIA Paris-Rocquencourt, France
Laboratoire CEDRIC, CNAM, France

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. We illustrate our approach with a concrete example where we instantiate a sorting algorithm written in Coq with the natural numbers of HOL.

1 Introduction

Interoperability is an emerging problem in the world of proof systems. Interactive 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. Obstacles for a large-scale interoperability are many, ranging from differences in the logical theory and the representation of data types, to the lack of a standard and effective way of retrieving proofs. For systems based on a common logical formalism, exchange formats for proofs have appeared like the TPTP derivation format [26] for traces of automated first-order theorem provers and OpenTheory [17] for HOL interactive theorem provers. However, combining systems working in different logical theories is harder.

A solution to this problem is to use a logical framework. The idea is to have a small and simple 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. This is the idea behind LF [14], implemented in Twelf [22], which has been used as a framework for interoperability in various projects [25, 16].

We propose to use a variant of Twelf called Dedukti. The reason for using Dedukti is that it implements an extension of LF called the $\lambda\Pi$-*calculus modulo rewriting* [5, 10], which adds term rewriting to the calculus. This extension not only allows for a more compact representation of proofs, but also enables the encoding of richer theories, such as the calculus of constructions. This cannot be done in LF efficiently because computation would have to be represented as a relation and every conversion made explicit. We thus use Dedukti as our logical framework.

Several tools have been developed to translate the proofs of various systems to Dedukti [6, 3, 11, 8]. The translations are based on the encodings of Cousineau and Dowek in the $\lambda\Pi$-calculus modulo rewriting [10]. 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 (of the order of several gigabytes).

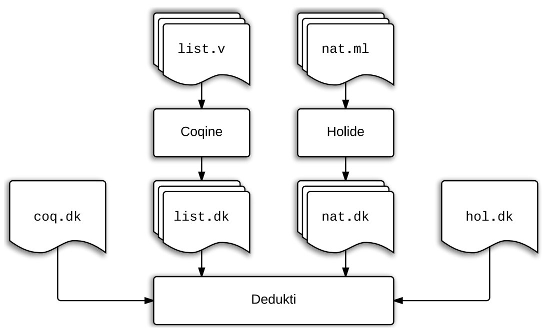
In this paper, we take one step further and show that we can combine the proofs coming from different systems in this same framework. A theorem can therefore be split into smaller blocks to be proved separately using different systems, and large libraries formalized in one system can be reused for the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

Submitted to:
PxTP 2015

This approach has several advantages. First, we can use Dedukti as an independent proof checker. The $\lambda\Pi$-calculus modulo rewriting is fairly simple, and the kernel implementation is relatively small [23, 24] compared to systems like Coq. The soundness and completeness of the translations have been studied and proved [2, 10, 12], giving us further confidence. Compared to direct one-to-one translations [18, 19, 21, 20], we avoid the quadratic blowup of the number of translations needed to translate n systems. In that scenario, if a new proof system enters the market, we would need to design n new translations. Moreover, some systems such as Coq have complex foundations that are difficult to translate to other formalisms. Another possibility would be to compose existing translations, provided that they are scalable and composable. This avenue has not been investigated. In our approach, we instead translate the different systems to one common framework. We do not propose translations back into other systems, as we can use Dedukti as a low-level assembly language, akin to machine language when we compile and link programs coming from different programming languages.

Contributions

We used Holide and Coqine to translate proofs of HOL [15] and Coq [27], 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. Our code is available online at `http://dedukti-interop.gforge.inria.fr/`.



2 Tools used

Dedukti

Dedukti[1] is a functional language with dependent types based on the $\lambda\Pi$-calculus modulo rewriting [23, 24]. 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.

[1] Available at: `http://dedukti.gforge.inria.fr/`

Holide

Holide[2] translates HOL proofs to the Dedukti language. It accepts proofs in the OpenTheory format (`.art`) [17], 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 detail in [3].

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. We represent them as terms of type type (not to be confused with Type, the "type of types" of Dedukti). We represent the propositions as terms of type bool.

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

Coqine

Coqine[3] 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 [6]. However, that translation is outdated, as it does not support the universe hierarchy and universe subtyping of Coq. A *universe* is just another name for a "'type of types". To avoid paradoxes, they are stratified into an infinite hierarchy [4], but that hierarchy is ignored by the first implementation of Coqine. The translation has since been updated to support both features following the ideas in [1], although some other features such as the module system are still missing.

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 that represents the universe of propositions and a type type i for every natural number i that represents the ith universe of types. We will write type_i and term_i for, respectively, type i and term i.

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

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 λ-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 set them apart. In this section, we examine these differences and show how we were able to bridge these gaps.

[2] Available at: https://www.rocq.inria.fr/deducteam/Holide/
[3] Available at: http://www.ensiie.fr/~guillaume.burel/blackandwhite_coqInE.html.en

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 by Keller and Werner [19]. 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 using rewrite rules:

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

Booleans and propositions

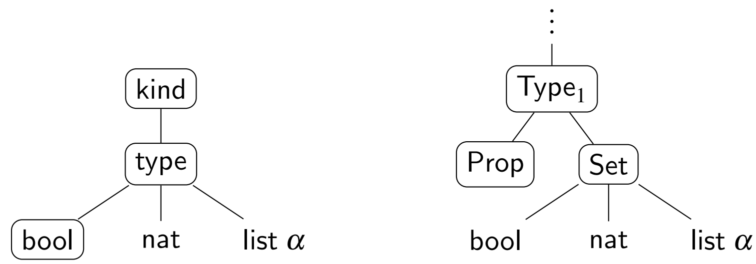
In Coq, there is a clear distinction between booleans and propositions. Booleans are defined as an inductive type bool with two constructors true and false. The type bool lives in the universe Set (which is another name for the universe Type_0). In contrast, following the Curry-Howard correspondence, 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.

To combine the two theories, one must therefore reconcile the two pictures in Figure 1, which show how the types of HOL and Coq are organized.[4] 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 $\mathsf{hol.bool} \to \mathsf{coq.prop}$, which we use to define hol.proof:

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

[4] Since bool is the type of propositions, and propositions are the types of proofs in the Curry-Howard correspondence, bool can be viewed as a universe [4, 13].



HOL Coq

Figure 1: Booleans and propositions in HOL and Coq. Boxes represent universes.

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 our case study.

4 Case study: sorting Coq lists of HOL numbers

We proved in Coq the correctness of the insertion sort algorithm on polymorphic lists and we instantiated it with the canonical order 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, permutation l (insertion_sort l).
```

with respect to a given (partial) order:

```
Variable A : Set.
Variable compare : A -> A -> bool.
Variable leq : A -> A -> Prop.
Hypothesis leq_trans : forall a b c, leq a b -> leq b c -> leq a c.
Hypothesis leq_total : forall a b, if compare a b then leq a b else leq b a.
```

The order comes in two flavors: a relation leq used for proofs, and a decidable version compare which we can destruct for building lists. The totality assumptions relates leq and compare and can be seen as a specification of compare.

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

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

The composition takes place in a Dedukti file named `interop.dk`. This file 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:

$$\Pi m\ n : \mathsf{holtypes.carrier\ hol\_nat}.\ \mathsf{if}\ (\mathsf{compare}\ m\ n)\ \mathsf{then}\ m \le n\ \mathsf{else}\ n \le m.$$

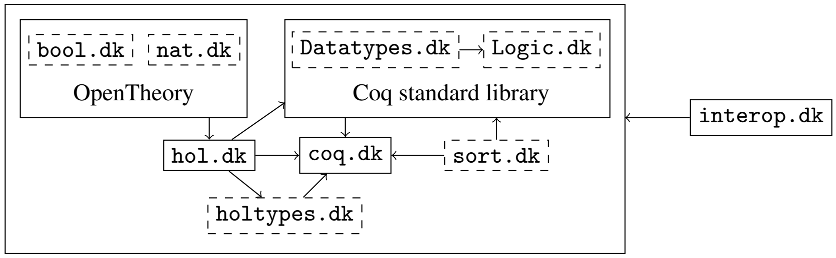


Figure 2: Components of the implementation. Solid frames represent source files. Dashed frames represent automatically generated files. Arrows represent dependencies.

We prove it using the following theorems from OpenTheory:

$$\forall m\, n : \mathsf{hol\_nat}.\ m < n \Rightarrow m \leq n$$
$$\forall m\, n : \mathsf{hol\_nat}.\ m \not\leq n \Leftrightarrow n < m$$

and some additional lemmas on if ... then ... else. Because of the verbosity of Dedukti and small style differences between HOL and Coq, this proof is long (several hundreds of lines) for such a simple fact. However, most of it is first-order reasoning and we believe that it could be automatically proved by the theorem prover Zenon [7] which can output proofs in the Dedukti format [9, 11].

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. We think it would have been harder for example to translate and link theorems about natural numbers in HOL and theorems about natural numbers in Coq. Our implementation is illustrated in Figure 2. All components were successfully verified by Dedukti.

5 Conclusion

We successfully translated a small Coq development to Dedukti and instanciated it with the HOL definition of natural numbers. The results have been validated by Dedukti. Mixing the underlying theories of Coq and HOL raised interesting questions but did not require a lot of human work: the file `hol.dk` is very close to the version included with Holide and the file `holtypes.v` is very small. In retrospect, the result looks a lot like an embedding of HOL in Coq but performed in Dedukti. This is not surprising, as the theory of HOL is fairly simple compared to Coq and is in fact a subset of the logic of Coq [4, 13, 19].

The interoperability layer `interop.dk` which is specific to our case study required a lot of work which should be automated before using this approach on larger scale; our next step on this front will be to integrate Zenon to solve the proof obligations when they happen to be in the first-order fragment. Interoperability raises more issues than mere proof rechecking and our translators to Dedukti need to be improved. 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. We expect more complex examples of interoperability to require some form of parametrization in the translators: when the developer wants the translator to map a given symbol to a specific Dedukti definition, he should be able to alter the behaviour of the translator by annotations in some source file, as done by Keller and Werner [19] and by Hurd [17].

Another limitation of this example of interoperability is the lack of 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 constructive and computational presentation of HOL will be necessary before we can obtain truly executable code. The pure type system presentation of HOL [4, 13] is a reasonable candidate for that but the proofs of OpenTheory will need to be adapted. Holide seems like a good starting point for such a transformation and is the subject of current ongoing work.

References

[1] Ali Assaf (2014): *A calculus of constructions with explicit subtyping*. Available at https://hal.inria.fr/hal-01097401. Accepted in Postproceedings of Types 2014.

[2] Ali Assaf (2015): *Conservativity of embeddings in the lambda-Pi calculus modulo rewriting*. Available at https://hal.inria.fr/hal-01084165. Accepted in TLCA 2015.

[3] Ali Assaf & Guillaume Burel (2015): *Translating HOL to Dedukti*. Available at https://hal.inria.fr/hal-01097412. Accepted in PxTP 2015.

[4] Henk Barendregt (1992): *Lambda calculi with types*. In Samson Abramsky, Dov M. Gabbay & Thomas S. E. Maibaum, editors: *Handbook of Logic in Computer Science*, 2, Oxford University Press, pp. 117–309.

[5] M. Boespflug, Q. Carbonneaux & O. Hermant (2012): *The lambda-Pi-calculus modulo as a universal proof language*. In: *Proof Exchange for Theorem Proving - Second International Workshop, PxTP 2012*, pp. 28–43.

[6] Mathieu Boespflug & Guillaume Burel (2012): *CoqInE: Translating the calculus of inductive constructions into the $\lambda\Pi$-calculus modulo*. In: *Proof Exchange for Theorem Proving - Second International Workshop, PxTP 2012*, p. 44.

[7] Richard Bonichon, David Delahaye & Damien Doligez (2007): *Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs*. In: *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, *LNCS/LNAI* 4790, Springer, pp. 151–165, doi:10.1007/978-3-540-75560-9_13.

[8] Guillaume Burel (2013): *A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo*. In Jasmin Christian Blanchette & Josef Urban, editors: *Proof Exchange for Theorem Proving - Third International Workshop, PxTP 2013*, *PxTP 2013* 14, EasyChair, pp. 43–57.

[9] Raphaël Cauderlier & Pierre Halmagrand (2015): *Checking Zenon Modulo proofs in Dedukti*. Accepted in PxTP 2015.

[10] Denis Cousineau & Gilles Dowek (2007): *Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo*. In Simona Ronchi Della Rocca, editor: *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, *LNCS* 4583, Springer, pp. 102–117.

[11] David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand & Olivier Hermant (2013): *Zenon Modulo: When Achilles Outruns the Tortoise Using Deduction Modulo*. In Ken McMillan, Aart Middeldorp & Andrei Voronkov, editors: *LPAR*, *LNCS* 8312, Springer Berlin Heidelberg, pp. 274–290.

[12] Gilles Dowek (2014): *Models and termination of proof-reduction in the lambda-Pi-calculus modulo theory*. Technical report, Inria, Paris. Available at https://who.rocq.inria.fr/Gilles.Dowek/Publi/superpi.pdf.

[13] Herman Geuvers (1993): *Logics and type systems*. PhD thesis, University of Nijmegen.

[14] Robert Harper, Furio Honsell & Gordon Plotkin (1993): *A framework for defining logics*. *Journal of the ACM* 40(1), pp. 143–184, doi:10.1145/138027.138060.

[15] John Harrison (2009): *HOL Light: An Overview*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics*, *LNCS* 5674, Springer Berlin Heidelberg, pp. 60–66, doi:10.1007/978-3-642-03359-9_4.

[16] Fulya Horozal & Florian Rabe (2011): *Representing model theory in a type-theoretical logical framework*. *Theoretical Computer Science* 412, pp. 4919–4945, doi:10.1016/j.tcs.2011.03.022.

[17] Joe Hurd (2011): *The OpenTheory Standard Theory Library*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *NFM*, *LNCS* 6617, Springer, pp. 177–191.

[18] Cezary Kaliszyk & Alexander Krauss (2013): *Scalable LCF-style proof translation*. In Sandrine Blazy, Christine Paulin-Mohring & David Pichardie, editors: *Interactive Theorem Proving*, *LNCS* 7998, Springer Berlin Heidelberg, pp. 51–66, doi:10.1007/978-3-642-39634-2_7.

[19] Chantal Keller & Benjamin Werner (2010): *Importing HOL Light into Coq*. In Matt Kaufmann & Lawrence C. Paulson, editors: *ITP*, *LNCS* 6172, Springer Berlin Heidelberg, pp. 307–322.

[20] Pavel Naumov, Mark-Oliver Stehr & José Meseguer (2001): *The HOL/NuPRL proof translator*. In Richard J. Boulton & Paul B. Jackson, editors: *Theorem Proving in Higher Order Logics*, *LNCS* 2152, Springer Berlin Heidelberg, pp. 329–345, doi:10.1007/3-540-44755-5_23.

[21] Steven Obua & Sebastian Skalberg (2006): *Importing HOL into Isabelle/HOL*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning*, *LNCS* 4130, Springer Berlin Heidelberg, pp. 298–302, doi:10.1007/11814771_27.

[22] Frank Pfenning & Carsten Schürmann (1999): *System Description: Twelf — A Meta-Logical Framework for Deductive Systems*. In: *Automated Deduction — CADE-16*, *LNCS* 1632, Springer Berlin Heidelberg, pp. 202–206, doi:10.1007/3-540-48660-7_14.

[23] Ronan Saillard (2013): *Dedukti: a universal proof checker*. In: *Foundation of Mathematics for Computer-Aided Formalization Workshop*, Padova. Available at `https://hal.inria.fr/hal-00833992`.

[24] Ronan Saillard (2013): *Towards explicit rewrite rules in the λΠ-calculus modulo*. In: *IWIL - 10th International Workshop on the Implementation of Logics*. Available at `https://hal.inria.fr/hal-00921340`.

[25] Carsten Schürmann & Mark-Oliver Stehr (2006): *An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf*. In Miki Hermann & Andrei Voronkov, editors: *Logic for Programming, Artificial Intelligence, and Reasoning*, *LNCS* 4246, Springer Berlin Heidelberg, pp. 150–166, doi:10.1007/11916277_11.

[26] Geoff Sutcliffe, Stephan Schulz, Koen Claessen & Allen Van Gelder (2006): *Using the TPTP Language for Writing Derivations and Finite Interpretations*. In Ulrich Furbach & Natarajan Shankar, editors: *Automated Reasoning*, *LNCS* 4130, Springer Berlin Heidelberg, pp. 67–81, doi:10.1007/11814771_7.

[27] The Coq development team (2012): *The Coq Reference Manual, version 8.4*. Available at `http://coq.inria.fr/doc`.