FoCaLiZe and Dedukti to the Rescue for Proof Interoperability

Raphaël Cauderlier, Catherine Dubois

FoCaLiZe and Dedukti to the Rescue for Proof Interoperability *

Raphaël Cauderlier[1] and Catherine Dubois[2]

[1] ENSIIE, Samovar, Évry, France
catherine.dubois@ensiie.fr
[2] University Paris Diderot, Irif, Paris, France,
raphael.cauderlier@irif.fr

Abstract. Numerous contributions have been made for some years to allow users to exchange formal proofs between different provers. The main propositions consist in ad hoc pointwise translations, e.g. between HOL Light and Isabelle in the Flyspeck project or uses of more or less complete certificates. We propose in this paper a methodology to combine proofs coming from different theorem provers. This methodology relies on the Dedukti logical framework as a common formalism in which proofs can be translated and combined. To relate the independently developed mathematical libraries used in proof assistants, we rely on the structuring features offered by FoCaLiZe, in particular parameterized modules and inheritance to build a formal library of transfer theorems called MathTransfer. We finally illustrate this methodology on the Sieve of Eratosthenes, which we prove correct using HOL and Coq in combination.

1 Introduction

According to the IEEE Standards Glossary, interoperability can be considered as *the ability of computer systems or software to exchange and make use of information.* Prover interoperability as a way for exchanging formal proofs between different theorem provers is a research topic that received many contributions along years. The most successful approach is probably the integration of automatic theorems provers (ATP) in interactive proof assistants (ITP) like Coq [1] or Isabelle [6]. In that case more or less detailed witnesses are provided and proofs can be imported or re-built. Furthermore many ad hoc pairwise translations have been proposed e.g. between HOL Light and Isabelle in the Flyspeck project [18], between HOL Light and Coq [12,25,19] or between HOL and Nuprl[15]. To avoid the quadratic blowup in the number of translators to develop, proof formats are emerging either for proofs in a specific logic such as the OpenTheory format [17] for ITPs in the HOL family or relying on logical frameworks [20,24,14] such as λ-prolog and Twelf to represent proofs in several logics. We propose to combine

* This work has been supported in part by the VECOLIB project of the French national research organization ANR (grant ANR-14-CE28-0018).

proofs coming from different theorem provers relying on the Dedukti logical framework [23], a typed λ-calculus with dependent types and rewriting, as a common formalism in which proofs can be translated and combined.

In [5], Assaf and Cauderlier describe a manual attempt of interoperability between HOL and Coq where they prove in Coq the correctness of the insertion sort algorithm on polymorphic lists and instantiate it with HOL natural numbers. This experiment relies on a translation to Dedukti for both the sorting function and the definition of HOL natural numbers (using respectively Coqine and Holide) and the result is checked by Dedukti. The interaction between both parts only happens at the level of booleans. However, for such a simple fact the proof is very long and verbose (around 700 Dedukti lines).

The goal of this paper is to make prover interoperability reach a new scale. We can notice that the art specific to the case study of Assaf and Cauderlier required a lot of work that could be automated and *has to* be automated to scale up. For this task, we use Zenon Modulo [8], an automated theorem prover outputting Dedukti proofs.

In this work, we go beyond simple boolean interaction. When a type and operations over this type, such as natural numbers and arithmetic operations, are independently defined in two ITPs, we can translate them but we end up with distinct isomorphic structures A and B in Dedukti. A theorem φ_A proved for A does not give us for free the corresponding theorem φ_B about B in which we are interested. Two solutions to this problem have been proposed:

- modify one of the translators to make it use the type and operations of the other structure thus identifying structures A and B,
- keep structures A and B distinct and use tactics to automatically prove *transfer* theorems of the form $\varphi_A \to \varphi_B$.

The first solution is favored in several ad hoc interoperability proposals [19,17]. The main limitation of this solution is the complexification of the translators which lacks scalability: for interoperability between n proof systems independently defining a mathematical structure, $n-1$ translators need to be modified to become customizable and point to the definition of the nth proof system. The second solution has first been proposed in the context of formalization in Isabelle/HOL of quotient structures [16] and recently ported to Coq [26]. Its main limitation is that the definitions of the morphisms and the proofs that operations are preserved by morphisms are left to the human user. We propose a compromise between these two solutions: we prove transfer theorems in FoCaLiZe [21], an ITP featuring a customizable Dedukti translator and use them to relate independent developments coming from uncustomizable translators.

The first contribution of this paper is a FoCaLiZe library of mathematical structures, morphisms, and transfer theorems called MathTransfer. The second contribution is a proposed methodology for scalable interoperability relying on Dedukti, Zenon Modulo, FoCaLiZe, and MathTransfer. The third contribution is the correctness proof of the Sieve of Erathostenes considered as the combination of a lemma coming from HOL and another coming from Coq. This proof illustrates our methodology.

The paper is structured as follows. In Sections 2, 3, and 4 we present very briefly resp. the Dedukti logical framework, the FoCaLiZe system, and the MathTransfer library. These tools form the basis of our approach to interoperability presented in Section 5. Sections 6 and 7 are devoted to a case study illustrating it on a correctness proof of the Sieve of Eratosthenes. Finally, we conclude and discuss in Section 8 the generality and reusability of our development.

The MathTransfer library and our interoperability case study are distributed together at the following URL: `https://gitlab.math.univ-paris-diderot.fr/cauderlier/math_transfer`.

2 Dedukti, a Universal Proof Language

Dedukti [23] is a variant of the dependently-typed λ-calculus Twelf, a logical framework based on the Curry-Howard correspondence. Logics are encoded in Dedukti by providing a signature and then proof checking in the encoded logic is reduced to type checking in the encoding signature. For example, the conjunction in natural deduction can be encoded by the following signature:

```
Prop: Type.
proof: Prop -> Type.
and: Prop -> Prop -> Prop.
and_intro: A: Prop -> B: Prop -> proof A -> proof B ->
           proof (and A B).
and_elim1: A: Prop -> B: Prop -> proof (and A B) -> proof A.
and_elim2: A: Prop -> B: Prop -> proof (and A B) -> proof B.
```

The type `Prop` of logical propositions is first declared, then to each proposition `A` we associate the dependent type of its proofs `proof A`. The conjunction `and` is then declared and so are finally the usual elimination and introduction rules.

The dependent product $\Pi x : A.\ B$ is written `x: A -> B` in Dedukti. It is used to encode universal quantification. Dependent products and arrow types are introduced by λ-abstractions and eliminated by applications. The λ-abstraction $\lambda x : A.\ b$ is written `x: A => b` in Dedukti. For example, a proof of commutativity of conjunction in Dedukti is the term

```
A: Prop => B: Prop => H: proof (and A B) =>
and_intro B A (and_elim2 A B H) (and_elim1 A B H)
```

of type `A: Prop -> B: Prop -> proof (and A B) -> proof (and B A)`

Dedukti also features rewriting which is used to express computational higher-order logics such as the Calculus of Inductive Constructions implemented in the Coq proof assistant [2].

Translators from various ITPs to Dedukti have been developed [4]. In particular, Holide [3], Coqine [2], and Focalide [10] are translators from respectively the OpenTheory format for ITPs in the HOL family, the Coq proof assistant and the FoCaLiZe framework. Some ATPs also produce Dedukti files, e.g. iProver Modulo [7] and Zenon Modulo [8,11] which is used in this work.

Dedukti is a mere proof checker for a wide variety of logics, it is not intended for direct human use and it intentionally lacks features commonly found in similar systems such as modularity, type inference and implicit arguments. While these features are not needed in a proof checker, they are crucial for scalability of interoperability developments. We propose to compensate this lack by using FoCaLiZe as an interoperability framework for linking mathematical libraries.

3 FoCaLiZe, Zenon Modulo, and Focalide to the Rescue

FoCaLiZe (`http://focalize.inria.fr`) has been designed as a formal environment for developing certified programs and libraries. It provides a set of tools to formally specify and implement functions and prove logical statements. FoCaLiZe comes with three backends, a backend to OCaml for execution and two backends for formal verification. The historic one produces Coq code and requires the use of the ATP Zenon which can output proofs as Coq terms. A more recent backend, called Focalide, produces Dedukti code [10] and requires to use Zenon Modulo [8], an extension of Zenon which produces Dedukti proofs [11]. In this work, we only use the Focalide backend.

We present here very briefly the main ingredients of FoCaLiZe. For more details please consult [21].

In FoCaLiZe, specifications are written in a typed version of first-order logics; implementations are written with the help of a pure functional programming language very close to ML with algebraic datatypes, first class citizens functions, polymorphic types, recursion and pattern-matching. FoCaLiZe proposes a high-level proof language and discharges the logical details to Zenon or Zenon Modulo (according to the used backend). A proof in this language consists of intermediate lemmas and hints to the prover. When a proof is out of scope of the prover, a manual proof expressed in the backend language, Coq or Dedukti, is required.

A FoCaLiZe unit, named a species, is made of signatures, properties, definitions of functions and types and also proofs of user-defined properties. A species mearly defines a set of values and functions manipulating them where the meaning of the functions are given by properties. Inside a species, the type `Self` denotes the type of these values, it is usually abstract early in the development and made concrete later. When a species is complete, that is every function is definied and every property is proven, it can be turned into a collection which is close to an abstract data type. FoCaLiZe features modularity, more precisely multiple inheritance. Thus a species can be defined by inheriting from some others, allowing the reuse of all the signatures, definitions and proofs coming from them. When the definition of a function is inherited, it is possible to give it a new definition overriding the inherited one. This feature is not used here. A FoCaLiZe development appears as a hierarchy of species linked by inheritance, such as the one described in Fig. 1. Moreover species can be parameterized by collections. In that case, inside a species, the user is allowed to use functions and properties as `black boxes` as in a functor. In the following, we say a species is *instanciated* when it is applied to a particular collection.

Similarly to the possibility to prove directly a theorem in one of the target logical languages, FoCaLiZe allows the definition of global symbols by custom external expressions of the target languages (OCaml, Coq, and Dedukti). It is, with modularity, a key feature for our interoperability application. For example, addition of integers is defined in FoCaLiZe standard library as follows. It is declared with its type in the FoCaLiZe side, each branch in the definition maps + to a function written in the corresponding target language:

```
let ( + ) = internal int -> int -> int
   external
   | caml -> {* Ml_builtins.bi__int_plus *}
   | coq -> {* coq_builtins.bi__int_plus *}
   | dedukti -> {* dk_int.plus *};;
```

In this article, we use FoCaLiZe as an interoperability framework to provide the features missing in Dedukti for this task: modularity offered by FoCaLiZe inheritance, and proof automation provided by Zenon Modulo.

4 MathTransfer, a Library of Transfer Theorems

If A and B are two isomorphic mathematical structures, then for any formula φ_A expressed in the language of A, the formula $\varphi_A \to \varphi_B$ is a theorem where φ_B is the formula corresponding to φ_A in the language of B. Theorems of the form $\varphi_A \to \varphi_B$ are called transfer theorems. The use of transfer theorems is a way to formalize rigorously the mathematical habit of *reasoning modulo isomorphism*.

MathTransfer is a FoCaLiZe library of transfer theorems about natural numbers. More precisely, the MathTransfer library contains:

- definitions of the mathematical structures obtained by adding common arithmetic operations on natural numbers,
- definitions of (iso)morphisms between abstract representations of natural numbers,
- proofs that all operations are preserved by the morphisms, and
- 84 transfer theorems.

Each structure is defined as a FoCaLiZe species. Because the definitions of some operations depend on other operations, these species are organized in a hierarchy presented in Fig. 1 (where frames represent species and an arrow goes from a species S_1 to a species S_2 if S_1 directly inherits from S_2).

The species in this hierarchy contain only the axiomatisations of the operations, not their other properties. For example, the species corresponding to the multiplication ($\times$ frame in Fig. 1) contains:

- a new binary operation $\times$ representing multiplication,
- two first-order axioms: $\forall n.\ 0 \times n = 0$ and $\forall m\ n.\ \mathrm{succ}(m) \times n = n + (m \times n)$.

This species is written as follows in FoCaLiZe:

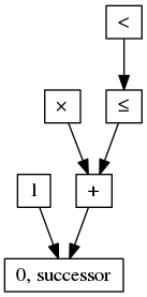

Fig. 1. The FoCaLiZe species hierarchy of MathTransfer structures

```
species NatTimes =
  inherit NatPlus;
  signature times : Self -> Self -> Self;
  property zero_times : all n : Self, times(zero, n) = zero;
  property succ_times : all m n : Self,
    times(succ(m), n) = plus(n, times(m, n));
end;;
```

On top of this small hierarchy, we build two orthogonal extensions: (a) a list of 84 statements about the arithmetic operations and (b) a hierarchy of morphisms between the structures.

The 84 chosen statements are a FoCaLiZe copy of the theorems about the operations of Fig. 1 that are proved in OpenTheory base library. Among them, 7 statements are properties of multiplication:

```
species NatTimesThm =
  inherit NatTimes;
  property times_zero : all m : Self,
    times(m, zero) = zero;
  property times_succ : all m n : Self,
    times(m, succ(n)) = plus(times(m, n), m);
  property times_assoc : all m n p : Self,
    times(times(m, n), p) = times(m, times(n, p));
  property times_commutes : all m n : Self,
    times(m, n) = times(n, m);
  property times_regular_left : all m n p : Self,
    times(m, n) = times (m, p) <-> (n = p \/ m = zero);
  property times_regular_right : all m n p : Self,
    times(m, p) = times (n, p) <-> (m = n \/ p = zero);
  property times_is_zero : all m n : Self,
    times(m, n) = zero <-> (m = zero \/ n = zero);
end;;
```

Morphisms on the other hand form a parameterized hierarchy of species. A morphism from a representation `A` of natural numbers is defined by a function `morph` of type `A -> Self` preserving zero and successors. From Peano axioms,

assumed both in `A` and in the current species, we prove that `morph` is a bijection preserving all the operations. For example, here is the parameterized species proving that multiplication is preserved by the morphism (proof is omitted):

```
species NatTimesMorph (A is NatTimes) =
  inherit NatTimes, NatPlusMorph(A);
  theorem morph_times : all a1 a2 : A,
          morph(A!times(a1, a2)) = times(morph(a1), morph(a2))
  proof = ...;
end;;
```

These proofs of preservation of operations are not fully automatized because they require reasoning by induction which is not handled by Zenon Modulo but Zenon Modulo is extensively used for the subproofs.

By inheriting from both the morphism hierarchy and the list of statements, we can state and automatically prove the transfer theorems. Below is a fragment of the species containing the 7 transfer theorems relative to the previous 7 theorems about multiplication:

```
species NatTimesTransfer (A is NatTimesThm) =
  inherit NatTimesMorph(A), NatTimesThm;
  proof of times_zero =
  by property A!times_zero, morph_zero, morph_times,
              morph_injective, morph_surjective;
  proof of times_succ =
  by property  A!times_succ, morph_succ, morph_times,
              morph_injective, morph_surjective;
  proof of times_assoc =
  by property A!times_assoc, morph_times,
              morph_injective, morph_surjective;
...
end;;
```

Each transfer proof relies on three ingredients:

- the corresponding theorem in the parameter `A`,
- bijectivity of `morph` (hypotheses `morph_injective` and `morph_surjective`),
- preservation of some operations by the morphism (hypotheses `morph_zero`, `morph_succ`, `morph_times`).

These transfer proofs are not automatically found by Zenon Modulo but are generated by a specialized transfer tactic written in Dedukti and similar to the transfer tactics for Isabelle and Coq [16,26].

5 Methodology for Dedukti-Based Interoperability

In this section, we propose an interoperability methodology based on Dedukti and MathTransfer. More precisely we detail below the different steps which must be followed when we want to use a lemma from a tool/formalism A in a formal

proof of a theorem in another formalism B. The statements of the lemma in A and B do not need to be syntactically identical but thanks to the ATP Zenon Modulo some degree of rephrasing of the lemma is tolerated.

Some prerequisites about A and B are required before applying the process. First translators from A and B to Dedukti must exist. Then we rely on the fact that formalisms A and B have already been merged in Dedukti, it means that the logical linking of both logics has been done (sources of inconsistencies have been identified and fixed).

The steps are the following ones (between parentheses appears the formalism or the tool to be used to realize the step):

1. identify the lemma L to exchange between A and B and prove it (A);
2. prove in B the target theorem with the exported A lemma L considered as an hypothesis (B);
3. translate both the A lemma L and the B development T in Dedukti (use the corresponding translators);
4. if needed, extend the FoCaLiZe hierarchies of the MathTransfer library with the operations appearing in the statement of the lemma L (FoCaLiZe);
5. instantiate the FoCaLiZe hierarchies with external definitions and proofs from A and B; if the statements do not exactly match, use Zenon Modulo (FoCaLiZe with the help of Zenon Modulo);
6. automatically transfer the lemma L (FoCaLiZe);
7. translate the whole FoCaLiZe development in Dedukti (use Focalide);
8. write the proof of the final target theorem (a trivial Dedukti proof).

In Sections 6 and 7, we apply this methodology to the correctness proof of the Sieve of Eratosthenes which is a small but typical case study where A is HOL and B is Coq.

6 Presentation of the Example: an Incomplete Coq Proof of the Sieve of Eratosthenes

In [5], Assaf and Cauderlier managed to link a Coq development with an HOL development directly in Dedukti because the example was chosen to minimize the interaction between Coq and HOL types. We now consider a more complicated example: a proved version of the Sieve of Eratosthenes. In this new proof of concept of interoperability in Dedukti, HOL and Coq have to agree on the type of natural numbers despite having slightly different definitions for it:

- in Coq, the type of natural numbers is defined as an inductive type;
- in HOL, inductive types are not primitive and natural numbers are encoded.

The Sieve of Eratosthenes is a well-known algorithm for listing all the prime numbers smaller than a given bound. In this section, we propose a certified implementation of this algorithm in the Coq proof assistant. We decompose this task in three: we have to program the sieve in Coq, to specify its correctness,

and to prove it. In Section 6.1, we program the sieve in Coq and in Section 6.2 we specify it and sketch a proof of the correctness of the algorithm. We highlight the mathematical theorems on which this proof relies. In order to experiment with interoperability, we will not prove these mathematical results in Coq but import them from the OpenTheory libraries[3].

6.1 Programming the Sieve of Eratosthenes in Coq

Divisibility plays two purposes in our development: we need a divisibility test inside the definition of the algorithm and we also need divisibility to define primality and specify the algorithm. In order to get a simple definition of primality, we introduce *strict* divisibility: we say that a is a strict divisor of b if a divides b and $1 < a < b$. Using Euclidean division, we define strict divisibility as a boolean function (`sd` in Coq, definition omitted here). A natural number $p > 1$ is then called a *prime* number if and only if it has no strict divisor.

We now have all the prerequisites for defining the sieve's core function. We use the usual *fuel* trick for avoiding a termination proof. In the following definition, `filter p l` computes the list of elements of `l` that satisfy the boolean function `p` and `negb` is boolean negation.

```
Fixpoint Sieve (l : list)(fuel : nat) {struct fuel} : list :=
  match fuel with
  | O => Nil
  | S fuel => match l with
    | Nil => Nil
    | Cons a l =>
      Cons a (Sieve (filter (fun b => negb (sd a b)) l) fuel)
    end
  end.
```

When `fuel` is bigger than the length of `l`, `Sieve l fuel` gives the expected result so the length of `l` is a convenient default value for `fuel`. Finally, the prime numbers smaller than $2 + n$ can be computed by the following function where `interval 2 n` computes the interval $[2, 2 + \texttt{n}]$.

```
Definition eratosthenes n := Sieve (interval 2 n) n.
```

6.2 Specification and Correctness Proof

The specification of the Sieve of Eratosthenes is quite simple: a number `p` is a member of the list returned by `eratosthenes n` if and only if `p` is a prime number smaller than `2 + n`.

We first define the `prime` predicate to be satisfied when its argument is a prime natural number:

[3] The purpose is to illustrate the methodology previously presented. Of course, this example is simple enough to be completely realized within Coq or done by reusing e.g. the translation from Hol Light to Coq proposed by Keller and Werner [19].

```
Inductive Istrue : bool -> Prop := ITT : Istrue true.

Definition prime p :=
  2 <= p /\ forall d, Istrue (negb (sd d p)).
```

We state the specification of the Sieve of Eratosthenes as the following three lemmata (where `In` is the list membership predicate).

A natural number returned by the function `erathostenes` is a prime number and is lower than the bound:

```
Lemma sound_1 p n : In p (eratosthenes n) -> p <= 2 + n.
Lemma sound_2 p n : In p (eratosthenes n) -> prime p.
```

Any prime number lower than the bound will be returned by the function `erathostheses`:

```
Lemma complete p n :
  prime p -> p <= 2 + n -> In p (eratosthenes n).
```

For completeness, it is enough to prove that the `Sieve` function preserves prime numbers (assuming it received enough fuel).

The first soundness lemma also relies on an invariant of the `Sieve` function, namely that the members of `Sieve l fuel` are all members of `l`. The proof is then concluded by a simple soundness property of intervals.

The second soundness lemma is where arithmetic is required. Let p be a member of `eratosthenes n`, we can easily prove that $2 \leq p$ by an argument similar to the proof of the first soundness lemma. To prove that p has no strict divisor, we use the following standard arithmetic result:

Lemma 1. *Let n be a natural number greater than 2, n has a prime divisor.*

For the sake of our proof of concept, we shall not prove this result in Coq. Fortunately, the prime divisor lemma is proved in OpenTheory `natural-prime` library so item number 1 on our interoperability checklist presented in Section 5 is skipped.

We prove the correctness of the Sieve of Eratosthenes in Coq when Lemma 1 is considered as a parameter thus completing item number 2 on our checklist. This development can be split into three parts of approximately the same size:

- straightforward arithmetic results such as commutativity of addition and multiplication, these results are proved in both Coq standard library and OpenTheory but they are so straightforward that they are easier to reprove than to import and Coqine is not yet able to translate the part of the standard library in which they are proved,
- correctness of auxiliary functions which could be reused in other developments (`modaux`, strict divisibility and functions manipulating lists), and
- correctness of the functions `Sieve` and `eratosthenes` which are specific to this problem.

As in [5], the results that we want to import from HOL are hypotheses of the final theorem that has to be provided in Dedukti.

7 Mixing the Proofs

In this section, we follow the steps outlined in Section 5 to import in our Coq development the prime divisor lemma from HOL. The prerequisites for the methodology to apply are met thanks to the work of Assaf and Cauderlier [5] that we summarize in Section 7.1. The various steps of the methodology are then followed in Sections 7.2 to 7.5. These steps are also pictured in Fig. 2.

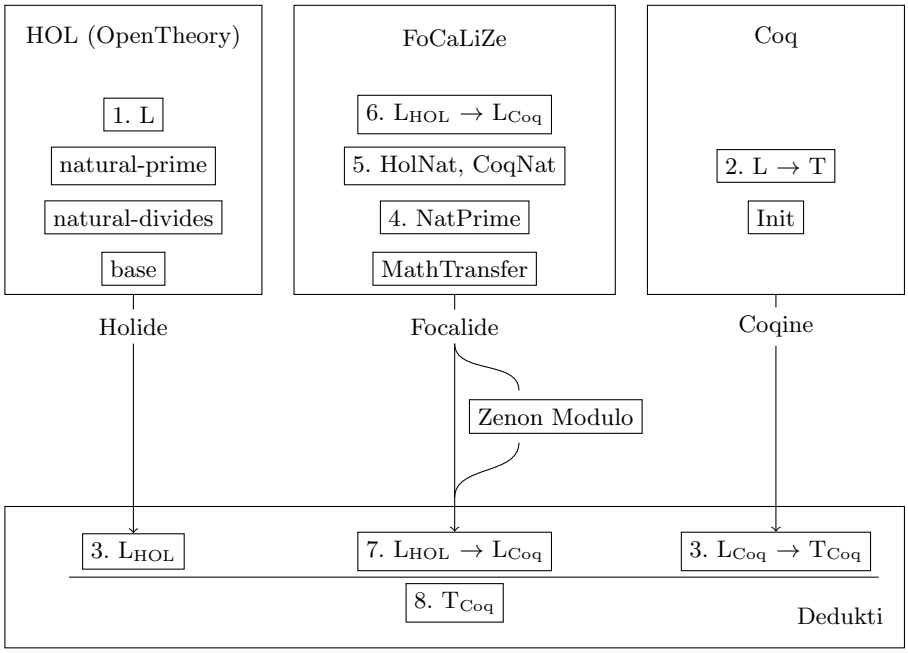

Fig. 2. The methodology in action for HOL/Coq interoperability

7.1 Linking HOL and Coq in Dedukti

In [5], Assaf and Cauderlier propose a first proof of concept of interoperability in Dedukti between HOL and Coq. The goal of this experiment was to study the logical linking of HOL and Coq logics.

Two sources of inconsistencies were identified. First, Coq and HOL do not agree on the question of type inhabitation: Coq allows empty types whereas we can prove in HOL that all types are inhabited. Second, the notions of booleans and logical propositions are identified in HOL and distinguished in Coq.

Type inhabitation is solved in [19] and [5] by identifying HOL types not with Coq types but with Coq inhabited types (in the Coq type $\Sigma A : \mathrm{Type}_0 \,.\, A$).

The difference between HOL booleans and Coq propositions is solved by identifying the type of HOL booleans with the type of Coq booleans, which are reflected as proposition by the symbol `hol_to_coq.Is_true` of type `hol.bool -> coq.prop`. This symbol is used to express provability in HOL as a special case of provability in Coq.

7.2 Extension of the MathTransfer Hierarchies up to the Prime Divisor Lemma

MathTransfer, as we have seen, contains transfer theorems corresponding to the most common arithmetic operations and relations such as found in OpenTheory base library. OpenTheory does also contain arithmetic definitions and theorems outside its base library. In particular, it defines divisibility and primality and it contains the following statement of the prime divisor lemma:

$$\forall n, n \neq 1 \rightarrow \exists p, (\text{prime}(p) \wedge p \mid n)$$

Following item number 4 on our checklist, we extend the FoCaLiZe hierarchies that we presented in Section 4 by four blocks:

- a definition of divisibility,
- a definition of strict (non-trivial) divisibility, this notion is used in the definition of primality,
- a definition of primality, this notion appears in the statement of the prime divisor lemma,
- the statement of the prime divisor lemma.

The extended hierarchy of operation definitions is shown in Fig. 3.

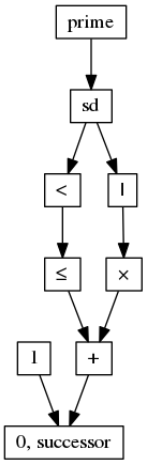

Fig. 3. The FoCaLiZe hierarchy of MathTransfer structures extended up to primality

Divisibility is required because this notion appears in the statement of the prime divisor theorem. It is defined as a binary relation $\mid$ defined by $m \mid n \leftrightarrow \exists p, m \times p = n$. Strict divisibility is used to define primality. There is also a binary relation sd defined by $m \ \text{sd} \ n \leftrightarrow (1 < m < n \wedge m \mid n)$. Primality is defined as the absence of strict divisor for numbers greater than 1. The corresponding predicate prime is defined by $\text{prime}(p) \leftrightarrow (1 < p \wedge \forall d, \neg(d \ \text{sd} \ p))$.

It is not required to state and transfer all the HOL lemmas dealing with divisibility and primality, it is enough to do so for the few ones that we are interested in such as the prime divisor lemma. The notion of isomorphism between representations of natural numbers is extended to take the new operations into account and the prime divisor lemma is automatically transferred.

7.3 Instantiation of Coq Natural Numbers

We can instantiate the hierarchy of species on the Coq side using FoCaLiZe external Dedukti definitions mapping directly the symbols to their Coqine translation in Dedukti. All the proofs required to instantiate the axioms characterizing the operations are trivial Dedukti proofs of reflexivity. For example, the species `NatTimes` is instantiated as follows:

```
species CoqTimes =
  inherit NatTimes, CoqPlus;
  let times(m : coq_nat, n : coq_nat) = internal coq_nat
    external
    | dedukti -> {* Coq__Init__Peano.mult m n *};
  proof of zero_times =
  dedukti proof definition of zero, times
  {* (n : cc.eT abst_T => logic.eq_refl abst_T abst_zero). *};
  proof of succ_times =
  dedukti proof definition of succ, plus, times
  {* (m : cc.eT abst_T => n : cc.eT abst_T =>
      logic.eq_refl abst_T (abst_times (abst_succ m) n)). *};
end;;
```

7.4 Instantiation of HOL Natural Numbers

Thanks to FoCaLiZe external definitions again, we can import in FoCaLiZe the HOL definitions of natural numbers and arithmetic operations. All the required proofs are found in the OpenTheory libraries. For example, the species `NatTimes` is instantiated as follows:

```
species HolTimes =
  inherit NatTimes, HolPlus;
  let times (p : hol_nat, q : hol_nat) = internal hol_nat
  external
  | dedukti -> {* HolNaturals.Number_2ENatural_2E_2A p q *};

  proof of zero_times =
```

```
  dedukti proof definition of zero, times
    {* HolNaturals.thm_117. *};

  theorem hol_succ_times : all m n : Self,
    times(succ(m), n) = plus(times(m, n), n)
  proof = dedukti proof definition of succ, plus, times
    {* HolNaturals.thm_157. *};
  proof of succ_times =
  <1>1 assume m n : Self,
       prove times(succ(m), n) = plus(n, times(m, n))
       <2>1 prove times(succ(m), n) = plus(times(m, n), n)
            by property hol_succ_times
       <2>2 prove plus(times(m, n), n) = plus(n, times(m, n))
            by property plus_commutes
       <2>f conclude
  <1>f conclude;
end;;
```

The theorems number 117 and 157 in the Holide output of OpenTheory base library respectively state $\forall n.\ 0 \times n = 0$ and $\forall m\ n.\ succ(m) \times n = (m \times n) + n$. The first one is exactly the statement of `zero_times` but the statement of `succ_times` is $\forall m\ n.\ succ(m) \times n = n + (m \times n)$. The gap is filled by Zenon Modulo thanks to a previous import of the commutativity of addition (property `plus_commutes`).

The hierarchy is fully implemented and can be turned in a collection, that is a species where every signature received a definition and every property has been proved.

```
species HolPrimeDiv =
 inherit NatPrimeDiv, HolPrime;
  ...
end;;
```

```
collection HolPrimeDivColl =
 implement HolPrimeDiv;
end;;
```

7.5 Instantiation of the Morphism

If f is a function of type $\alpha \to \alpha$ and n is a natural number, we note f^n the nth iteration of the function f ($f^0 = Id$, $f^n = f \circ f \circ \ldots f$, n times).

Both the Coq Init library[4] and the OpenTheory base library define this polymorphic iteration of a function f. We use them to define the isomorphism between HOL natural numbers and Coq ones. The morphism from HOL natural numbers to Coq ones is defined by an HOL iteration of the Coq successor function $\texttt{morph}(n) := \texttt{coq\_succ}^n(\texttt{coq\_zero})$ (`coq_zero` and `coq_succ` are mapped to the Dedukti translation of the Coq definitions) and its inverse is defined by a Coq iteration of the HOL successor function $\texttt{inv\_morph}(n) := \texttt{hol\_succ}^n(\texttt{hol\_zero})$.

By instantiating all the morphisms and transfer hierarchies (items 5 and 6 of our methodology), we finally obtain in FoCaLiZe the prime divisor theorem

[4] The Coq Init library is the part of Coq standard library defining logical connectives and basic datatypes such as natural numbers and lists.

on the Coq formulation of arithmetic structures. Once translated in Dedukti by Focalide, this theorem matches the assumption of the correctness proof of the Sieve of Eratosthenes translated from Coq so we obtain a Dedukti proof of the correctness of the Sieve of Eratosthenes (item number 8 of our methodology).

Quantitatively, the size of the various parts of this development are given in Fig. 4. The HOL part of the development consists in a fragment of the OpenTheory library that was developed independently and contains thousands of theorems irrelevant to our case study. The Coq development however is of reasonable size and was specifically developed for this case study. In the case of FoCaLiZe, more than half of the generated code is produced by Zenon Modulo; this shows how useful proof automation has been in this development. Finally, the small Dedukti development is taken from the merging of Coq and HOL logics in [5].

	HOL	Coq	FoCaLiZe	Zenon Modulo	Dedukti
Source Code	3.2M	31K	129K		9K
Generated Dedukti Code	90M	828K	1.3M	1.7M	

Fig. 4. Size of the various parts of the development

8 Conclusion

We achieved our goal of certifying a Coq implementation of the Sieve of Eratosthenes using arithmetic results from OpenTheory. FoCaLiZe inheritance and parametrization allowed us to devise MathTransfer, a library of mathematical structures and transfer theorems. Zenon Modulo was of great help during this formalization since a lot of small steps of equational reasoning were needed and proving them in Dedukti would have been painful. We tried to do as much work as possible in a system independent way. The MathTransfer library is independent of HOL and Coq. Thanks to the symmetry in the roles of Coq and HOL, we can not only import lemmas from HOL to Coq but also in the other direction. Moreover, the definitions of the operations do not need to be identical in both systems. It is usual in FoCaLiZe to limit the dependencies to the definitions of methods thanks to late binding [22]. For small differences Zenon Modulo can fill the gap, for bigger ones such as divisibility (which is derived from Euclidean division on the Coq side) the equivalence of the definitions can be proved in either system.

This working example of interoperability needs to be reproduced with bigger proofs but also with proofs coming from some other systems if their underlying logics can be encoded within Dedukti. We believe that the methodology illustrated in this paper is scalable. However more automation is required in particular for the extension of MathTransfer. We expect the work of Gauthier and Kaliszyk [13] on automatic discovering of isomorphic structures from different formal libraries to adapt for this task. A limitation of our approach to

interoperability in Dedukti is the trust we can have in the final proof because it is expressed in an uncommon logic whose consistency is not yet proved. Users of ITPs might expect from an interoperability development to obtain a proof in their trusted system. In order to translate back the proof in the combined logic to one of the original systems, we need to remove from the proof the use of unnecessary axioms of the other system. Preliminary work in this topic has been proposed in [9] where Cauderlier uses Dedukti rewriting to automatically remove classical axioms in Zenon proofs.

References

1. M. Armand, G. Faure, B. Grégoire, C. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In J. Jouannaud and Z. Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011.*, volume 7086 of *LNCS*, pages 135–150. Springer, 2011.
2. A. Assaf. *A Framework for Defining Computational Higher-Order Logics.* PhD thesis, École Polytechnique, 2015.
3. A. Assaf and G. Burel. Translating HOL to Dedukti. In C. Kaliszyk and A. Paskevich, editors, Proceedings Fourth Workshop on *Proof eXchange for Theorem Proving,* Berlin, Germany, August 2-3, 2015, volume 186 of *EPTCS*, pages 74–88, 2015.
4. A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, and R. Saillard. Expressing Theories in the $\lambda\Pi$-Calculus Modulo Theory and in the Dedukti System. Draft available online at `http://www.lsv.ens-cachan.fr/~dowek/Publi/expressing.pdf`, 2016.
5. A. Assaf and R. Cauderlier. Mixing HOL and Coq in Dedukti. In C. Kaliszyk and A. Paskevich, editors, 4th Workshop on *Proof eXchange for Theorem Proving,* Berlin, Germany, August 2-3, 2015, volume 186 of *EPTCS*, pages 89–96, 2015.
6. J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic Proof and Disproof in Isabelle/HOL. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011.*, volume 6989 of *LNCS*, pages 12–27. Springer, 2011.
7. G. Burel. Experimenting with Deduction Modulo. In V. Sofronie-Stokkermans and N. Bjrner, editors, *CADE 2011*, volume 6803 of *LNAI*, pages 162–176. Springer, 2011.
8. G. Bury, D. Delahaye, D. Doligez, P. Halmagrand, and O. Hermant. Automated Deduction in the B Set Theory using Typed Proof Search and Deduction Modulo. In *LPAR 20 : 20th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* , Suva, Fiji, Nov. 2015.
9. R. Cauderlier. A Rewrite System for Proof Constructivization. In *Proceedings of the 2016 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, pages 2:1–2:7. ACM, 2016.
10. R. Cauderlier and C. Dubois. ML pattern-matching, recursion, and rewriting: from FoCaLiZe to Dedukti. In *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016.*, volume 9965 of *LNCS*, pages 459–468. Springer Berlin Heidelberg, 2016.
11. R. Cauderlier and P. Halmagrand. Checking Zenon Modulo Proofs in Dedukti. In C. Kaliszyk and A. Paskevich, editor, Proceedings 4th Workshop on *Proof eXchange for Theorem Proving,* Berlin, Germany, August 2-3, 2015, volume 186 of *EPTCS*, pages 57–73, 2015.

12. E. Denney. A Prototype Proof Translator from HOL to Coq. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 13th International Conference, TPHOLs 2000, Portland, Oregon, USA, August 14-18, 2000, Proceedings*, volume 1869 of *LNCS*, pages 108–125. Springer, 2000.
13. T. Gauthier and C. Kaliszyk. Matching concepts across HOL libraries. In S. M. Watt, J. H. Davenport, A. P. Sexton, P. Sojka, and J. Urban, editors, *Intelligent Computer Mathematics - International Conference, CICM 2014, Coimbra, Portugal, July 7-11, 2014. Proceedings*, volume 8543 of *Lecture Notes in Computer Science*, pages 267–281. Springer, 2014.
14. F. Horozal and F. Rabe. Representing Model Theory in a Type-Theoretical Logical Framework. *Theoretical Computer Science*, 412:4919–4945, 2011.
15. D. J. Howe. Importing Mathematics from HOL into Nuprl. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 26-30, 1996, Proceedings*, volume 1125 of *LNCS*, pages 267–281. Springer, 1996.
16. B. Huffman and O. Kuncar. Lifting and Transfer: A Modular Design for Quotients in Isabelle/HOL. In G. Gonthier and M. Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*, volume 8307 of *LNCS*, pages 131–146. Springer, 2013.
17. J. Hurd. The OpenTheory Standard Theory Library. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011.*, volume 6617 of *LNCS*, pages 177–191. Springer, 2011.
18. C. Kaliszyk and A. Krauss. Scalable LCF-Style Proof Translation. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, number 7998 in LNCS, pages 51–66. Springer Berlin Heidelberg, 2013.
19. C. Keller and B. Werner. Importing HOL Light into Coq. In M. Kaufmann and L. C. Paulson, editors, *ITP*, number 6172 in LNCS, pages 307–322. Springer, 2010.
20. D. Miller. Foundational Proof Certificates: Making Proof Universal and Permanent. In A. Momigliano, B. Pientka, and R. Pollack, editors, *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice, LFMTP 2013, Boston, Massachusetts, USA, September 23, 2013*, pages 1–2. ACM, 2013.
21. F. Pessaux. FoCaLiZe: Inside an F-IDE. In C. Dubois, D. Giannakopoulou, and D. Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, volume 149 of *EPTCS*, pages 64–78, 2014.
22. V. Prevosto and M. Jaume. Making proofs in a hierarchy of mathematical structures. In *Proceedings of Calculemus*, Sept. 2003.
23. R. Saillard. *Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, MINES Paritech, 2015.
24. C. Schürmann and M.-O. Stehr. An Executable Formalization of the HOL/Nuprl Connection in the Metalogical Framework Twelf. In M. Hermann and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, number 4246 in LNCS, pages 150–166. Springer, 2006.
25. F. Wiedijk. Encoding the HOL Light logic in Coq, 2007. unpublished notes.
26. T. Zimmermann and H. Herbelin. Automatic and Transparent Transfer of Theorems along Isomorphisms in the Coq Proof Assistant. *CoRR*, abs/1505.05028, 2015.