Exporting an Arithmetic Library from Dedukti to HOL

François Thiré

Exporting an Arithmetic Library from Dedukti to HOL*

François Thiré[1]

1 **LSV**
2 **ENS Paris-Saclay**
3 **INRIA**

Abstract

Today, we observe a large diversity of proof systems. This diversity has the negative consequence that a lot of theorems are proved many times. Unlike programming languages, it is difficult for these systems to co-operate because they do not implement the same logic. Logical frameworks are a class of theorems provers that overcome this issue by their capacity of implementing various logics. In this work, we study the STT$\forall_{\beta\delta}$ logic, an extension of the Simple Type Theory that has been encoded in the logical framework Dedukti [1]. We show that this new logic is a good candidate to export proofs to other provers. As an example, we show how this logic has been encoded into Dedukti and how we used it to export proofs to the HOL family provers via OpenTheory [2].

Keywords and phrases Dedukti, OpenTheory, HOL, Coq, Interoperability

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction

There are nowadays, many proof systems, e.g. Matita, Coq, HOL Light, etc. but although they share the same goal, these systems are based on different logics. In these systems, Fermat's little theorem requires to prove about 300 hundreds lemmas that are part of the arithmetic library of the system. Elaborate such a library can be a tiresome work and we would like to be able to translate proofs from one system to another.

The aim of this paper is to investigate the possibility to export an arithmetic library from one proof system to others. To achieve this goal, we use the logical framework Dedukti that is able to express several logics. In particular, we will explain how the logic STT$\forall_{\beta\delta}$, an extension of STT defined in this paper is convenient to express arithmetic statements and can be easily encoded in Dedukti to be afterwords exported to other proof systems. In particular, we will show how we have exported an arithmetic library from STT$\forall_{\beta\delta}$ to the proof systems OpenTheory.

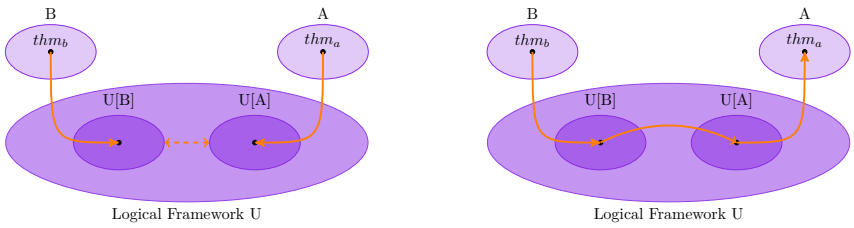
1.1 Why interoperability is a problem

The lack of standard in the field makes interoperability between proof systems difficult. There are several problems that may arise at different levels. The first level is on the logical point of view where a statement can have several equivalent definitions. As an example, the well-foudness property for an order can be defined either as:

1. Every subset of E as a minimal element

* A full version of the paper is available at [TODO:]

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



(a) Interoperability: first solution **(b)** Interoperability: second solution

Figure 1 Interoperability with a logical framework

2. There is not infinite decreasing sequence

For example, the proof system HOL Light implements the first definition while the Coq system implements the second. Moreover, the equivalence between these definitions depends on the logic. Showing the equivalence needs one formulation of the axiom of choice that is not provable in Coq for example. The second level is that two statements can be syntactically different but convertible such as:

1. $\forall n, m \in \mathbb{N}, n + m = m + n$
2. commutative $+$

where commutative is defined as $\lambda f.\ \forall n\ m, f\ n\ m = f\ m\ n$. Finally at the third level, there is an issue with the name of the statements themselves. For example, the commutativity of addition is called `add_comm` in Coq v8.6 or `plus_comm` in HOL4.

But developing from scratch a whole library is difficult and time-consuming. For example proving the Feit-Thompson theorem [3] took several man-years and it is not imaginable to prove such theorems from scratch in several proof systems. It is therefore necessary to develop solutions to make interoperability between proof systems easier.

1.2 Logical Frameworks and Interoperability

One one hand, interoperability is not possible in general, since some logics are more expressive than others. For example, one may quantify over proofs in Coq that it is not possible in HOL. On the other hand, it is not conceivable to develop for every pair of proof systems, a specific translation between them because there would be a quadratic number of translations.

Logical frameworks is a solution to overcome these two issues. They are a special kind of formal provers where different logics and proof systems can be specified. Logical frameworks can be used in two different ways to solve this problem that are exposed in Fig. 1 and explained below. Suppose that in the proof system A a theorem thm_a needs a proof of theorem thm_b proved in the proof system B. In the following, if the system U is a logical framework and A is a proof system, we denote $U[A]$ the encoding of A in U.

A first solution is to use the logical framework to encode both proofs and combined them. Then check that the combined proof (understandable by the logical framework) is correct. This solution raises several problems: there are two definitions for the same objects, and one needs to prove that both representations are isomorphic; a second problem is that the proof is also expressed in a logic that is not obviously consistent even if it is the union of two consistent logics. This solution has already been explored by Cauderlier in [4].

In this paper, we are interested in a second solution. This solution as described in second picture of figure 1 can be decomposed in three steps. We first translate thm_A from the proof

system A to its encoding in $U[A]$. Then translate thm_a from $U[A]$ to $U[B]$. And finally translate thm_a from $U[B]$ to the proof system B. In this process, the first and last steps are total functions. However, the second step is a partial function since the translation is not always possible. For example, proof irrelevance is not expressable in HOL but it is in CiC. Also, one might think that translating proofs from $U[A]$ to $U[B]$ is quite similar to translating proofs from A to B, and therefore this step is also specific for the two proof systems A and B. This is not completely true for the following reasons:

- Working inside a logical framework may unify the transformations: Inductive types from Matita or Coq will be encoded the same way in a logical framework. Moreover, if one wishes to transform inductive types in the theory encoded in that logical framework, that transformation does not depend on the prover where the terms are coming from but only on the encoding.
- It is possible to cut the translation in smaller steps that can be shared between several transformations.
- If a system A is a subsystem of a system B, then one can directly translate proofs from the encoding of A to the system B directly.

1.3 Contribution

For this work, we are going to use Dedukti as our logical framework. Dedukti is based upon the $\lambda\Pi$-calculus modulo theory, an extension of LF [5] with rewrite rules [1]. This extension has three main advantages:

- It is easier to express different systems in Dedukti than in LF
- Proofs encoded in Dedukti are smaller thanks to rewrite rules.
- Encodings can be *shallow* (defined in section 2) in most cases (such as STT$\forall_{\beta\delta}$)

The last property mentioned above is important since one problem that could arise with translations and encodings is to pollute statements so that statements become unreadable and unusable. This can be avoided thanks to shallow encodings.

Our aim is to fully export the Fermat's little theorem coming from an arithmetic library encoded in STT$\forall_{\beta\delta}$ to OpenTheory and Coq. OpenTheory is already an interoperability tool that allows to share proofs between several implementations of HOL. Therefore, targeting OpenTheory allows us to get the proofs in every system that support OpenTheory. We have decided to take an arithmetic library for two reasons. First, because arithmetic is a branch of mathematics that is very common and therefore we can expect that every formal provers is capable of proving arithmetic statements. Second, this library is neither too small nor too big to make experimentations (340 lemmas).

STT$\forall_{\beta\delta}$ is an extension of STT with prenex polymorphism and is modulo $\beta\delta$[1]. We think that this logic is particularly well suited to export arithmetic proofs to other proof systems.

2 Dedukti and STT$\forall_{\beta\delta}$

2.1 Dedukti

Dedukti is a logical framework that implements the $\lambda\Pi$-calculus modulo theory, a calculus that extends LF with user-defined rewrite rules. These rules might are used in the convertibility test. The syntax and the type system of $\lambda\Pi$-calculus modulo theory are presented in

[1] δ meaning the (un-)folding of constants

$$\begin{array}{lccl} \text{Terms} & \text{A,B,t,u} & ::= & \textbf{Kind} \mid \textbf{Type} \mid \Pi \text{x} : \text{A}.\ \text{K} \mid A\ B \mid \lambda x^A.\ B \mid \text{x} \\ \text{Contexts} & \Gamma & ::= & \emptyset \mid \Gamma, x : A \mid \Gamma, t \hookrightarrow u \end{array}$$

Figure 2 Dedukti syntax

$$\frac{}{[\,]\ \text{well-formed}}\ \text{EMPTY CTX} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A\ \text{well-formed}}\ \text{CTX SORT}$$

$$\frac{\Gamma \vdash t \hookrightarrow\ \text{well-formed}}{\Gamma, t \hookrightarrow u\ \text{well-formed}}\ \text{CTX} \hookrightarrow \qquad \frac{\Gamma\ \text{well-formed}}{\Gamma \vdash \textbf{Type} : \textbf{Kind}}\ \text{CTX } \textbf{Type Kind}$$

$$\frac{\Gamma\ \text{well-formed} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{VARIABLE} \qquad \frac{\Gamma \vdash A : \textbf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x^A.\ B : s}\ \text{PROD}$$

$$\frac{\Gamma \vdash A : s \qquad \Gamma, x : A \vdash B : s \qquad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.\ t : \Pi x^A.\ B}\ \text{ABS}$$

$$\frac{\Gamma \vdash t : \Pi x^A.\ B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\ t' : B}\ \text{APP} \qquad \frac{\Gamma \vdash A : s \qquad \Gamma \vdash t : A \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}\ \text{CONV}$$

Fig. 3 $\lambda\Pi$ typing system

Fig 2 and 3. The type system is not complete because we do not present how to derive the judgment $\Gamma \vdash t \hookrightarrow u$. A complete description of the system can be found in [6]. Roughly, a rewrite rule $t \hookrightarrow u$ is well typed when the types of t and u are the same. One advantage of using rewrite rules is that more systems can be encoded in Dedukti using a *shallow* encoding. Such encoding has the two following properties: A binder of the source language is translated as a binder in the second language (using HOAS for example) and the typing judgment from the source language is translated as a typing judgment for Dedukti. This means that we can use the type checker of Dedukti to check directly if a term encoded in Dedukti is well-typed. The next two paragraphs are dedicated to the STT$\forall_{\beta\delta}$ system and its (shallow) encoding in Dedukti.

2.2 STT$\forall_{\beta\delta}$

Before going down to a full description of STT$\forall_{\beta\delta}$, we want to stress one our two points about interoperability. This will explain some choices made in STT$\forall_{\beta\delta}$ such has it is possible to declare constants.

2.2.1 How interoperability should work

Let's say that in the system $\mathcal{A}$, I have a proof of $\forall\ (x\ y : \mathbb{N}), x + y = y + x$, in general the system do not define natively the type $\mathbb{N}$ nor the operator $+$, these types and operators are user-defined. In Coq, $\mathbb{N}$ is a inductive type, but in HOL, it is a type operator. The same occurs for $+$ where in Coq it is define has a fixed point but in HOL it is defined using two

equalities (there is no fixed point operator). Let denotes $+_A$ the operator $+$ in the system $\mathcal{A}$. A naive inspection on interoperability could be seen as how to translate the theorem

$$\Sigma_A; \emptyset \vdash \forall\ (x\ y : \mathbb{N}_A), x_A + y = y +_A x$$

to

$$\Sigma_B, \emptyset \vdash \forall\ (x\ y : \mathbb{N}_B), x_B + y = y +_B x$$

. But this is problematic for the following reasons:

- Definition of a type is not unique. For example, in Coq, natural numbers have three different definitions. Which one should be used?
- If one use an intermediate system to translate these proofs, the encoding from the first system to the intermediate may degrade the original definition. Reconstitute the original type or constant might be difficult. For example, the inductive of the natural numbers is translated as four declarations ($\mathbb{N}$, 0, S, the recursor definition) and two rewrite rules in Dedukti. But from these declarations and these rewrite rules, it is difficult to identify that this correspond to inductive definition of natural numbers because we cannot rely on the names.
- Finally, if one wants to automatize this translation, it is in general impossible to guess the definition of every types and constants involved in the development inside the target system. For example, if one identify in the previous example that this is the natural numbers, how to automatize the translation to the target system without hard-coding the definitions of some pre-define types and pre-define constants?

For all of these reasons, our translation process produces a library where the definition of some constants are axiomatized.

- The target user can use the definition he wants as long as the axioms related to the constant are provable
- The intermediate do not care of the *meaning* of some declarations nor some axioms. It translates simply what it sees.
- There is no more problems regarding automation since there is no *guess* nor hard-coding to do

2.2.2 A description of STT$\forall_{\beta\delta}$

STT$\forall_{\beta\delta}$ is an extension of STT with prenex polymorphism and type operators. A type operator is constructed using a name and an arity. This allows to declare types such as `bool`, `nat` or α `list`. The polymorphism of STT$\forall_{\beta\delta}$ is restricted to prenex polymorphism as full polymorphism would make this logic inconsistent[2] [7]. The STT$\forall_{\beta\delta}$ syntax is presented in Fig. 4. Introducing types operator makes **Prop** and $\to$ redondante since they can be declared as type operators, respectively of arity 0 and 2. But these two types have a particular role in the typing judgment that is why we let them in the syntax. Also, STT$\forall_{\beta\delta}$ allows to declare and define constants. Declaring constants is better for interoperability as discussed in section 2.2.1. The typing system and the proof system are as expected and presented in Fig. 5 and Fig. 6. Also notice that we identify in STT$\forall_{\beta\delta}$ the terms t and t' if they are convertible modulo β and δ (unfolding of constants).

The meta theory of STT$\forall_{\beta\delta}$ is proved in the full version of this paper.

[2] This papers shows also that omit types annotations for polytypes would make the logic inconsistent

Type operators	p		
Type variables	X		
Monotypes	A, B	$:\equiv$	X $\mid$ **Prop** $\mid A \to B \mid p\ A_1\ ...\ A_n$
Polytypes	T	$:\equiv$	A $\mid \forall X.\ T$
Constants	cst		
Constants terms	c	$:\equiv$	cst $\mid$ c A
Terms variables	x		
Monoterms	t,u	$:\equiv$	x $\mid$ c $\mid \lambda x^A.\ t \mid t\ u \mid t \Rightarrow u \mid \forall x^A.\ t \mid \lambda X.\ t$
Polyterms	τ	$:\equiv$	$t \mid \forall X.\ \tau$
Typing Context	Γ	$:\equiv$	$\emptyset \mid \Gamma, t : T \mid \Gamma, X$
Proof Context	Ξ	$:\equiv$	$\emptyset \mid \Xi, t$
Constant Context	Σ	$:\equiv$	$\emptyset \mid \Sigma, cst = \tau : T \mid \Sigma, cst : T \mid \Sigma, (p : n)$
Typing Judgment	$\mathcal{T}$	$:\equiv$	$\Sigma; \Gamma \vdash \tau : T$
Proof Judgment	$\mathcal{P}$	$:\equiv$	$\Sigma; \Gamma; \Xi \vdash \tau$
Type well-formed			$\Sigma; \Gamma \vdash A$ **wf**
Typing context well-formed			$\Sigma \vdash \Gamma$ **wf**
Constant context well-formed			Σ **wf**

Figure 4 STT$\forall_{\beta\delta}$ syntax

▶ **Theorem 1.** *STT$\forall_{\beta\delta}$ is consistent.*

▶ **Theorem 2.** *The type checking and proof checking in STT$\forall_{\beta\delta}$ is decidable.*

2.2.3 Example: Equality in STT$\forall_{\beta\delta}$

This paragraph explains how to implement Leibniz equality (denote $=_{\mathcal{L}}$) in STT$\forall_{\beta\delta}$. Its type will be $\forall X.\ X \to X \to \mathbf{Prop}$ and can be inplemented as: $\lambda X.\ \lambda x^X.\ \lambda y^X.\ \forall P^{P \to \mathbf{Prop}}.\ P\ x \Rightarrow P\ y$. From this definition, it is possible to prove that $=_{\mathcal{L}}$ is reflexive expressed by the following statement

$$\forall X.\ \forall x^X.\ x =_{\mathcal{L}} x$$

using the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{=_{\mathcal{L}}; X, x : X; P\ x \vdash P\ x}\ \text{ASSUME}}{=_{\mathcal{L}}; X, x : X; \emptyset \vdash P\ x \to P\ x}\ \Rightarrow_I}{=_{\mathcal{L}}; X, x : X; \emptyset \vdash x =_{\mathcal{L}} x}\ \text{CONV}}{=_{\mathcal{L}}; X; \emptyset \vdash \forall x^X.\ x =_{\mathcal{L}} x}\ \forall_I}{=_{\mathcal{L}}; \emptyset; \emptyset \vdash \forall X.\ \forall x^X.\ x =_{\mathcal{L}} x}\ \forall_I$$

2.3 Dedukti[STT$\forall_{\beta\delta}$]

Thanks to Higher-order Abstract syntax (HOAS), we have a shallow encoding of STT$\forall_{\beta\delta}$ in Dedukti shown in 7.

For the types, we declare two constants *type* and *ptype* that are respectively the types for the types schema and the type of STT$\forall_{\beta\delta}$. Therefore every type of STT$\forall_{\beta\delta}$ will be encoded

$$\frac{X \in \Gamma}{\Sigma; \Gamma \vdash X \ \mathbf{wf}} \text{ WF VAR} \qquad \frac{}{\Sigma; \Gamma \vdash \mathbf{Prop} \ \mathbf{wf}} \text{ WF PROP}$$

$$\frac{\Sigma; \Gamma \vdash A \ \mathbf{wf} \qquad \Sigma; \Gamma \vdash B \ \mathbf{wf}}{\Sigma; \Gamma \vdash A \to B \ \mathbf{wf}} \text{ WF FUN} \qquad \frac{\Sigma \vdash \Gamma \ \mathbf{wf}}{\Sigma \vdash \Gamma, X \ \mathbf{wf}} \text{ WF CTX VAR}$$

$$\frac{(p : n) \in \Sigma \qquad \Sigma; \Gamma \vdash A_i \ \mathbf{wf}}{\Sigma; \Gamma \vdash p \ A_1 \ \ldots \ A_n \ \mathbf{wf}} \text{ WF TYOP APP} \qquad \frac{}{\emptyset \ \mathbf{wf}} \text{ WF EMPTY}$$

$$\frac{\Sigma \vdash \Gamma \ \mathbf{wf} \qquad \Sigma; \Gamma \vdash A \ \mathbf{wf}}{\Sigma \vdash \Gamma, x : A \ \mathbf{wf}} \text{ WF CTX VAR} \qquad \frac{\Sigma \ \mathbf{wf} \qquad p \notin Dom(\Sigma)}{\Sigma, (p : n) \ \mathbf{wf}} \text{ WF TYOP}$$

$$\frac{\Sigma \ \mathbf{wf} \qquad cst \notin Dom(\Sigma) \qquad FV(T) = \emptyset}{\Sigma, cst : T \ \mathbf{wf}} \text{ WF CST DECL}$$

$$\frac{\Sigma \ \mathbf{wf} \qquad cst \notin Dom(\Sigma) \qquad \Sigma; \emptyset; \emptyset \vdash \tau : T}{\Sigma, cst = \tau : T \ \mathbf{wf}} \text{ WF CST DEFN} \qquad \frac{\Sigma \ \mathbf{wf} \qquad \Sigma \vdash \Gamma \ \mathbf{wf}}{\mathcal{C}, x : A \vdash x : A} \text{ VAR}$$

$$\frac{\mathcal{C} \vdash f : A \to B \qquad \mathcal{C} \vdash t : A}{\mathcal{C} \vdash f \ t : B} \text{ APP} \qquad \frac{\mathcal{C}, x : A \vdash t : B}{\mathcal{C} \vdash \lambda x^A.\ t : A \to B} \text{ ABS}$$

$$\frac{\mathcal{C} \vdash t : \mathbf{Prop} \qquad \mathcal{C} \vdash u : \mathbf{Prop}}{\mathcal{C} \vdash t \Rightarrow u : \mathbf{Prop}} \text{ IMP} \qquad \frac{\mathcal{C}, x : A \vdash t : \mathbf{Prop}}{\mathcal{C} \vdash \forall x^A.\ t : \mathbf{Prop}} \text{ FORALL}$$

$$\frac{\mathcal{C}, X \vdash t : T}{\mathcal{C} \vdash \lambda X.\ t : \forall X.\ T} \text{ POLY INTRO} \qquad \frac{FV(B) \subseteq \mathcal{C} \qquad \mathcal{C} \vdash c : \forall X.\ T}{\mathcal{C} \vdash c \ B : T[X := B]} \text{ CST APP}$$

$$\frac{typeof(\mathcal{C}, cst) = T}{\mathcal{C} \vdash cst : T} \text{ CST} \qquad \frac{\mathcal{C}, X \vdash \tau : \mathbf{Prop}}{\mathcal{C} \vdash \forall X.\ \tau : \mathbf{Prop}} \text{ POLY PROP}$$

Fig. 5 STT$\forall_{\beta\delta}$ typing system

$$\frac{\mathcal{C} \vdash t : \mathbf{Prop}}{\mathcal{C}, t \vdash t}\ \text{ASSUME}$$

$$\frac{\mathcal{C} \vdash t \qquad \mathcal{C} \vdash t \Rightarrow u}{\mathcal{C} \vdash u}\ \Rightarrow_E \qquad \frac{\mathcal{C}, t \vdash u}{\mathcal{C} \vdash t \Rightarrow u}\ \Rightarrow_I \qquad \frac{\mathcal{C} \vdash \forall x^A.\ t \qquad \mathcal{C} \vdash u : A}{\mathcal{C} \vdash t[x := u]}\ \forall_E$$

$$\frac{\mathcal{C}, x : A \vdash t \qquad x \notin \mathcal{C}}{\mathcal{C} \vdash \forall x^A.\ t}\ \forall_I \qquad \frac{\mathcal{C} \vdash \forall X.\ \tau \qquad FV(A) \subseteq \Gamma}{\mathcal{C} \vdash \tau[X := A]}\ \forall_E \qquad \frac{\mathcal{C}, X \vdash \tau}{\mathcal{C} \vdash \forall X.\ \tau}\ \forall_I$$

$$\frac{\mathcal{C} \vdash t \qquad t \equiv_{\beta\delta} t'}{\mathcal{C} \vdash t'}\ \text{CONV}$$

Fig. 6 Proof system

as an object in Dedukti. That is why we need a coercion encoded by the constant *etap* to coerce a STT$\forall_{\beta\delta}$ type to a Dedukti type. The constant *eta* is not necessary but it makes the encoding clearer. To go from *type* to *ptype* we declare the constant p. Then we need constants to represent type constructors of STT$\forall_{\beta\delta}$: *bool* for **Prop** and $\dot{\to}$ for $\to$. Each type constructor is encoded by a Dedukti constant of type $type \to \ldots type$ with $n+1$ occurences of *type* if the arity of p is n. Finally, we use the constant $forallK_{type}$ to encode polymorhic types.

For the terms, since the encoding is shallow, we do not need constants for abstractions and applications. We only need two constants for $\forall$ and $\Rightarrow$ that encode respectively $forall$ and $impl$. We also need another constant to encode polymorphic propositions with $forallK_{bool}$. Finally, we add rewrite rules to make the encoding works. For example we want that an encoding of a term of type **Prop** $\to$ **Prop** of STT$\forall_{\beta\delta}$ is a abstraction in Dedukti. This is possible thanks to the rewrite rule:

```
etap (p (arr l r)) --> eta l -> eta r
```

A STT$\forall_{\beta\delta}$ constant will be encoded by a Dedukti constant. We explain below how to translate Leibniz equality and a reflexivity proof in Dedukti[STT$\forall_{\beta\delta}$]. The full translation can be found in the full version of this paper.

2.3.1 A proof of reflexivity:

The translation of Leibniz equality in Dedukti[STT$\forall_{\beta\delta}$] is as follow. First the type of $=_{\mathcal{L}}$ is translated as:

```
etap (forallKtype (X:type => arr X (arr X bool)))
```

then its definition is translated as

```
A:type => x:(eta A) => y:(eta A) =>
forall (arrow A prop) (P:(eta (arrow A prop)) => impl (P x) (P y)).
```

Finally, the proof of `refl` is translated as

$$
\begin{array}{rr}
type : & \texttt{Type} \\
\dot{\to} : & type \to type \to type \\
bool : & type \\
eta : & type \to \texttt{Type} \\
ptype : & \texttt{Type} \\
p : & type \to ptype \\
etap : & ptype \to \texttt{Type} \\
forallK_{type} : & (type \to ptype) \to ptype \\
eps : & eta\ bool \to \texttt{Type} \\
impl : & eta(bool \dot{\to} bool \dot{\to} bool) \\
forall : & \Pi t : type.\ eta((t \dot{\to} bool) \dot{\to} bool) \\
forallK_{bool} : & (type \to eta\ bool) \to eta\ bool
\end{array}
\qquad
\begin{array}{rl}
eta \hookrightarrow & \lambda t.\ etap(p\ t) \\
etap(p(arr\ l\ r)) \hookrightarrow & eta\ l \to eta\ r \\
etap(forallK_{type}\ f) \hookrightarrow & x^{type} \to etap\ (f\ x) \\
eps(forall\ t\ f) \hookrightarrow & x^{eta\ t} \to eps\ (f\ x) \\
eps(impl\ l\ r) \hookrightarrow & eps\ l \to eps\ r \\
eps(forallK_{bool}\ f) \hookrightarrow & x^{type} \to eps\ (f\ x)
\end{array}
$$

Figure 7 Signature for STT$\forall_{\beta\delta}$ in Dedukti

```
A:type => x:(eta A) => P : (eta (arr A bool)) => h : eps (P x) => h.
```

that is of type

```
eps (forallKbool (X:type => forall X (x:(eta X) => leibniz X x x))
```

which is the translation of

$$\forall X.\ \forall x^X.\ x =_s x$$

Notice that each introduction rule leads to the introduction of an abstraction while an elimination rule leads to the introduction of an application.

One can prove that this encoding is sound and complete: every statement thm are provable in STT$\forall_{\beta\delta}$ if and only if the $[\![thm]\!]$ is provable in Dedukti[STT$\forall_{\beta\delta}$].

▶ **Theorem 3** (Soundness and completness). *This encoding is sound and complete*

Proof. By mutual induction on the derivability of each judgment. ◀

3 OpenTheory

HOL (for higher order logic) is a logic that is implemented in several systems with some minor differences. OpenTheory is a tool that allows to share proofs between several implementations of HOL. Since we are targetting OpenTheory, we will mostly refer to the logic defined by OpenTheory. It is a logic inspired by $\mathcal{Q}_0$ where the only logical connective is the equality. Terms are the one of STLC with an equality symbol while the type system extends the one of STLC by declaring type operators and prenex polymorphism. The proof system of OpenTheory can be found in Fig. 8.

$$\frac{}{t \vdash t}\ \textsc{assume} \qquad \frac{\Gamma \vdash t = u}{\Gamma \vdash \lambda v.t = \lambda v.u}\ \textsc{absThm} \qquad \frac{\Gamma \vdash t_1 = u_1 \qquad \Delta \vdash t_2 = u_2}{\Gamma \cup \Delta \vdash t_1\ t_2 = u_1\ u_2}\ \textsc{appThm}$$

$$\frac{}{t_1, \ldots, t_n \vdash u}\ \textsc{axiom} \qquad \frac{}{\vdash (\lambda x.t)u = t[u/x]}\ \beta$$

$$\frac{\Gamma \vdash t \qquad \Delta \vdash u}{(\Gamma - u) \cup (\Delta - t) \vdash t = u}\ \textsc{deductAntiSym} \qquad \frac{\Gamma \vdash t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash u}\ \textsc{eqMp}$$

$$\frac{\Gamma \vdash t \qquad \Delta \vdash u}{(\Gamma - u) \cup \Delta \vdash t}\ \textsc{proveHyp} \qquad \frac{}{\vdash t = t}\ \textsc{refl} \qquad \frac{\Gamma \vdash t}{\Gamma\sigma \vdash t\sigma}\ \textsc{subst} \qquad \frac{\Gamma \vdash t = u}{\Gamma \vdash u = t}\ \textsc{sym}$$

$$\frac{\Gamma \vdash t_1 = t_2 \qquad \Delta \vdash t_2 = t_3}{\Gamma \cup \Delta \vdash t_1 = t_3}\ \textsc{trans}$$

Fig. 8 OpenTheory proof system

3.1 STT$\forall_{\beta\delta}$ vs OpenTheory

One may notice that the two systems STT$\forall_{\beta\delta}$ and OpenTheory are quite close. However, there are some differences that makes the translation from STT$\forall_{\beta\delta}$ and OpenTheory tedious:

- Terms in OpenTheory are only convertible up to α conversion while in STT$\forall_{\beta\delta}$ it is up to α, β, δ conversion
- All the connectives of OpenTheory are defined upon the equality symbol, while in STT$\forall_{\beta\delta}$ they are defined upon $\forall$ and $\Rightarrow$ connectives
- Prenex polymorphism in OpenTheory is implicit. All free variables in a type are implicitly quantified while in STT$\forall_{\beta\delta}$ it is explicit

These differences lead to three different proof transformations:

1. Encode the $\forall$ and $\Rightarrow$ connectives using the equality of OpenTheory
2. Explicit each application of the conversion rule
3. Finally, we get rid of the type quantifier

3.1.1 Encoding $\forall$ and $\Rightarrow$ using the equality

A first idea to encode STT$\forall_{\beta\delta}$ logic in OpenTheory would be to axiomitize all the rules of STT$\forall_{\beta\delta}$ and then translate the proofs using these axioms. But translating a rule to an axiom in OpenTheory requires to use the implication. But since OpenTheory does not know what is an implication, suche axioms would be unusable since it would not be possible to use the modus ponens to eliminate the implication itself. Therefore, one must find an encoding of the $\forall$ and $\Rightarrow$ connectives such that the rules of STT$\forall_{\beta\delta}$ ara admissible. Such encoding is already known from $\mathcal{Q}_0$ [8]. This encoding preseneted in Fig. 9 uses two other connectives: $\top$ and $\wedge$ that can be defined in OpenTheory. Note that in this encoding, the definition of $\top$ may remain abstract but the definition of $\wedge$ is needed to prove the admissibility of the rules of STT$\forall_{\beta\delta}$. These definitions can be easely translated as axiom in OpenTheory.

$$
\begin{array}{rl}
\top & \\
x \wedge y :\equiv & \lambda f.\ f\ x\ y = \lambda f.\ f\ \top\ \top \\
x \to y :\equiv & x \wedge y = x \\
\forall x.P :\equiv & \lambda x.\ P = \lambda x.\ \top
\end{array}
$$

Figure 9 Encoding of $\forall$ and $\to$ in $\mathcal{Q}_0$

We stress here that it is really important to axiomatize these definitions and not to define new constants. The difference is that it will be possible to instantiate latter these connectives by the *true* connectives of HOL as long as these axioms can be proved regardless of their definition in HOL. Also, these axioms are not too strong to satisfy because in HOL, propositional extensionality[3] is admissible. Using this encoding, it is possible to derive all the rules of STT$\forall_{\beta\delta}$ in OpenTheory using the four axioms below. Below, we proof the admissibility of the $\forall_I$ rule:

$$\dfrac{\dfrac{\Pi}{\mathcal{C}, x : A \vdash t} \qquad x \notin \mathcal{C}}{\mathcal{C} \vdash \forall x^A.\ t}$$

Below, Γ is the translation of $\mathcal{C}$ in OpenTheory[4].

$$\dfrac{\dfrac{\dfrac{\dfrac{\Pi}{\Gamma \vdash t} \quad \dfrac{}{\Gamma \vdash \top}}{\Gamma \vdash t = \top}}{\Gamma \vdash \lambda x.\ t = \lambda x.\ \top} \quad \dfrac{\dfrac{}{\Gamma \vdash \forall x.\ t\ x = (\lambda x.\ t = \lambda x.\ \top)}}{\Gamma \vdash (\lambda x.\ t = \lambda x.\ \top) = \forall x.\ t\ x}}{\Gamma \vdash \forall x.\ t\ x}$$

The right branch is closed thanks to the forall axiom. After this translation, the syntax of the terms is the same as STT$\forall_{\beta\delta}$ except that we add an equality symbol and the connectives $\forall$ and $\Rightarrow$ become defined constants.

3.1.2 Eliminate β, δ reductions

The decidability of STT$\forall_{\beta\delta}$ relies on the fact that the term rewriting system defined by $\hookrightarrow_\beta$ and $\hookrightarrow_\delta$ is convergent. This means that to decide is $t \equiv_{\beta\delta} u$ we can compute the normal form of t and u and check that they are equal up to α. This means that the conversion test

$$\dfrac{\dfrac{}{\Gamma \vdash t} \qquad \dfrac{}{\Gamma \vdash t \equiv_{\beta\delta} u}}{\Gamma \vdash u}$$

can be reduced to

[3] $\forall P, Q, (\forall x, P\ x = Q\ x \Rightarrow P = Q)$

[4] In OpenTheory, free variables such as x does need to appear inside the context

$$\frac{\overline{\Gamma \vdash t} \qquad \Gamma \vdash t \hookrightarrow^*_{\beta\delta} t' \qquad \Gamma \vdash u \hookrightarrow^*_{\beta\delta} u' \qquad \Gamma \vdash t' =_\alpha u'}{\Gamma \vdash u}$$

Since equality in OpenTheory is also modulo α, the last judgment is not a problem anymore. The problem remains to translate $t \hookrightarrow_{\beta\delta} u$ in OpenTheory.

To handle β and δ OpenTheory has two rules:

$$\frac{}{\Gamma \vdash c = t} \qquad\qquad \frac{}{\Gamma \vdash \lambda x.\ t\ u = t[x := u]}$$

However we need to apply these rules inside a context. In STT$\forall_{\beta\delta}$, a context can be described by the following grammar:

$$C ::= \cdot \mid C\ u \mid t\ C \mid \lambda x.\ C \mid \forall X.\ C \mid C \Rightarrow u \mid t \Rightarrow C \mid \forall x^A.\ C$$

The goal is to derive the following rule for every context C:

$$\frac{\Gamma \vdash t \hookrightarrow_{\beta\delta} u}{\Gamma \vdash C[t] \hookrightarrow_{\beta\delta} C[u]}$$

This is done inductively on the structure of C. To do that, we need an equality judgment that goes through the context. In OpenTheory, there are two rules to handle abstractions and applications. We need to derive such rules for the connectives of STT$\forall_{\beta\delta}$. Here we show only the admissibility of

$$\frac{\Gamma \vdash p = p' \qquad \Gamma \vdash q = q'}{\Gamma \vdash p \Rightarrow q = p' \Rightarrow q'}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, p \Rightarrow q \vdash p \Rightarrow q} \qquad \dfrac{\dfrac{}{\Gamma, p' \vdash p'} \qquad \dfrac{\Gamma \vdash p = p'}{\Gamma \vdash p' = p}}{\Gamma, p' \vdash p}}{\Gamma, p \Rightarrow q, p' \vdash q} \qquad \Gamma \vdash q = q'}{\Gamma, p \Rightarrow q, p' \vdash q'}}{\Gamma, p \Rightarrow q \vdash p' \Rightarrow q'} \quad \vdots}{\Gamma \vdash p \Rightarrow q = p' \Rightarrow q'}$$

We put dots for the right part of the DeductAntiSym rule because the proof is symmetric.

From this rule, it is easy to derive the rule for context:

$$\frac{\Gamma \vdash C[t] = C[t'] \qquad \overline{\Gamma \vdash u = u}}{\Gamma \vdash C[t] \Rightarrow u = C[t'] \Rightarrow u}$$

3.1.3 Suppress the type quantifier

Since OpenTheory implicitely quantifies over free types variables, we need to remove the $\forall$ constructor on types and the $\forall'$ connective. But we cannot just remove these symbols, we also need to avoid capture. To do that, we need to do two things:

- Ensure that in each type, all the bound variables have a different name
- Replace each occurence of the $\forall'_E$ by a renaming step with the SUBST rule of OpenTheory

So

$$\frac{\mathcal{C} \vdash \forall' X.\ \tau \qquad FV(A) \subseteq \Gamma}{\mathcal{C} \vdash \tau[X := A]}\ \forall'_E$$

is translated as

$$\frac{\dfrac{\mathcal{C} \vdash \tau \qquad Z\ fresh}{\mathcal{C} \vdash \tau[X := Z]}}{\mathcal{C} \vdash \tau[Z := A]}$$

4 From Dedukti[STT$\forall_{\beta\delta}$] to Coq

Going from STT$\forall_{\beta\delta}$ to Coq is really easy since all the constructions in STT$\forall_{\beta\delta}$ are possible in Coq. However, some subtleties comes from types operators and constant declarations. A type operator such as `nat` and the two declared constants `0:nat` and `S : nat -> nat` would be ideally translated as an inductive type. However, it is really difficult (maybe impossible?) to do that for all inductive types in an automatic way. But this is not a real problem since it is possible to declare axioms and parameters in Coq. A more general observation is made in 2.2.1. Moreover, using the Curry-De Bruijn correspondance, it is possible to translate every proof STT$\forall_{\beta\delta}$ into a Coq terms. In our case, since we start from Dedukti[STT$\forall_{\beta\delta}$], we already have the lambda term.

Getting back to our reflexivity proof. One can translate the equality $=_s$ as the following definition in Coq

```
Definition = : forall X:Type, X -> X -> Prop :=
fun X:Type =>
    fun x:X =>
        fun y:X => forall (P:X -> Prop), P x -> P y.
```

Then the proof of reflexivity can be translated from the Dedukti[STT$\forall_{\beta\delta}$] as the coq definition

```
Definition refl = : forall X:Type, forall x:X x = x :=
fun X:Type => fun x:X => fun h:P x => h.
```

One may notice that going from Dedukti[STT$\forall_{\beta\delta}$] consists mostly to remove *etap* annotations.

	Dedukti[STT]	OpenTheory	Coq
size (mb)	1.5	41	1
translation time (s)	-	18	3
time to check (s)	0.1	13	6

Table 1 Arithmetic library translation

5 The arithmetic library

We have implemented these transformations and we have used them on an arithmetic library encoded in Dedukti[STT$\forall_{\beta\delta}$]. The results can be found in the table 1.

These results show that the difficult translation needed for OpenTheory impacts on the type checking time and the size of the library. Though, this time seems reasonable. The final statement for Fermat's litthe theorem we get in Coq is the following:

```
Definition congruent_exp_pred_S0 :
forall p a : nat,
prime p -> Not (divides p a) -> congruent (exp a (pred p)) (S O) p.
```

In the library, the constants `prime`, `congruent` and `pred` are defined by the library while the constants `exp`, `Not`, `O` and `S` are declared by the library and should be instantiated by the user. Once these constants are instantiated (with a functor mechanism of Coq for example), the theorem is ready to use. The instantiation for these constants cannot be anythyng, they have to satisfy some axioms. For the constant `exp`, the two following axioms has to be satisfied:

```
Axiom sym_eq_exp_body_O :
forall n : nat, (S O) = (exp n O).
Axiom sym_eq_exp_body_S :
forall n m : nat, (times (exp n m) n) = (exp n (S m)).
```

It is quite easy to come with the following definition for `exp`:

```
Fixpoint exp (n m : nat) : Datatypes.nat :=
  match m with
  | O => S O
  | S m => n * exp n m
  end
```

For the arithmetic library, one has to give 40 definitions for constants and prove 80 axioms. All the constants definitions follow from their name or from the axioms they have to satisfy, and hence the axioms are easy to prove.

6 Conclusion

In this paper, we used Dedukti to export an arithmetic library from the STT$\forall_{\beta\delta}$ logic to the proof system Coq and the proof system OpenTheory. We showed how STT$\forall_{\beta\delta}$ is a simple logic that can be encoded easely in Dedukti and is powerful enough to express arithmetic proofs. The differences between OpenTheory and STT$\forall_{\beta\delta}$ makes the translation from the later to the former a bit tedious. To overcome this issue, we have isolated three main

differences between the two systems and for each difference, proposed a translation. While the translation to OpenTheory is tedious, we show how the translation to Coq is very easy since it is basically just an embedding. Finally, we propose a framework for interoperability between systems so that it is effectively possible to reuse a developement from one system to another.

6.1 Future work

We would like to export this library to other proof systems such as PVS or Agda. While for Agda, the translation should be similar to the one of Coq, for PVS this is a challenge since there is no proof term but only tactics. In other word, each rule should be translated by an application of one or more tactics and the behaviour can make this translation a bit tricky.

We also would like to export more arithmetic proofs and other proofs that could be encoded in the logical framework Dedukti. Though this work requires to be able to translate these proofs in Dedukti[STT].

Finally, we hope that this work is the beginning of a process that could lead to a standardization of proofs. A first step would be to standardize this arithmetic library so that every proof systems share the same arithmetic library.

References

1 Cousineau, D., Dowek, G.: Embedding pure type systems in the lambda-pi-calculus modulo. In Rocca, S.R.D., ed.: Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings. Volume 4583 of Lecture Notes in Computer Science., Springer (2007) 102–117

2 Hurd, J.: The OpenTheory standard theory library. In Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R., eds.: Third International Symposium on NASA Formal Methods (NFM 2011). Volume 6617 of Lecture Notes in Computer Science., Springer (April 2011) 177–191

3 Gonthier, G.: Engineering mathematics: the odd order theorem proof. In Giacobazzi, R., Cousot, R., eds.: The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013, ACM (2013) 1–2

4 Cauderlier, R., Dubois, C.: Focalize and dedukti to the rescue for proof interoperability. In Ayala-Rincón, M., Muñoz, C.A., eds.: Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings. Volume 10499 of Lecture Notes in Computer Science., Springer (2017) 131–147

5 Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. J. ACM **40**(1) (January 1993) 143–184

6 Saillard, R.: Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique). PhD thesis, Mines ParisTech, France (2015)

7 Coquand, T.: An analysis of girard's paradox. In: LICS, IEEE Computer Society (1986) 227–236

8 Andrews, P.B.: An introduction to mathematical logic and type theory - to truth through proof. Computer science and applied mathematics. Academic Press (1986)