Sharing a Library between Proof Assistants: Reaching out to the HOL Family *

François Thiré

Sharing a Library between Proof Assistants: Reaching out to the HOL Family*

François Thiré
ENS Paris-Saclay, LSV, CNRS, Université Paris-Saclay, INRIA, LIX
francois.thire@lsv.fr

We observe today 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 theorem 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 Simple Type Theory that has been encoded in the logical framework Dedukti [6]. We present a translation from this logic to OpenTheory [8], a proof system and interoperability tool between provers of the HOL family. We have used this translation to export an arithmetic library containing Fermat's little theorem to OpenTheory and to two other proof systems that are Coq [13] and Matita [2].

1 Introduction

Since Automath and LCF, many proof systems have been designed and used to develop computer-checked mathematics, for example Matita, Coq, HOL4, HOL Light, Isabelle/HOL... These systems sometimes implement different logics. In each of them, proving Fermat's little theorem requires proving about 300 lemmas which contribute to the arithmetic library of the system. Developing such a library can be tiresome and we may want, instead of recreating it again and again in different systems, to translate this library from one system to another.

The aim of this paper is to present the logic STT$\forall_{\beta\delta}$, an extension of Simple Type Theory that is powerful enough to express easily arithmetic theorems, but weak enough so that it is easy to export theorems from this logic to several other systems, making this logic suitable for interoperability. STT$\forall_{\beta\delta}$ has been implemented in the logical framework Dedukti [6]. In order to illustrate its adequacy for exporting theorems, we have successfully implemented a translation from STT$\forall_{\beta\delta}$ to Coq and Matita, and in order to target proof systems based on HOL (Higher-Order Logic), we have also implemented a translation from STT$\forall_{\beta\delta}$ to OpenTheory [8], which is a proof system for interoperability between the provers of the HOL family. Then, we applied our translations on an arithmetic library that was available in STT$\forall_{\beta\delta}$. The translation from STT$\forall_{\beta\delta}$ to OpenTheory is interesting because even if these two systems are very close, they are based on different design choices that make the translation harder than expected. The description of STT$\forall_{\beta\delta}$ and its translation to OpenTheory are the two main contributions of this paper.

*Associated webpage at http://www.lsv.fr/~fthire/research/sttforall/index.php.

Submitted to:
LFMTP 2018

1.1 Logical Frameworks and Interoperability

Sharing proofs between systems is not always possible since some logics are more expressive than others. For example, one may quantify over proofs in the Calculus of Constructions but not in Higher-Order Logic. Moreover, it is not conceivable to develop, for every pair of proof systems, a specific translation because there would be a quadratic number of translations.

Logical frameworks offer an approach to overcoming these two issues. They are a special kind of proof systems in which different logics and proof systems can be specified. Using a logical framework, there are two ways of sharing proofs. Suppose that the proof of a theorem thm_A expressed in a logic $\mathcal{L}_A$ needs a proof of a theorem thm_B already proven in the logic $\mathcal{L}_B$. A first solution explored, for instance, by Cauderlier and Dubois [3] is to have the combined proof inside the logical framework by encoding the proofs of thm_A and thm_B in it. In this solution, proofs are not exported outside of the logical framework.

Another solution is to translate thm_A directly from $\mathcal{L}_A$ to $\mathcal{L}_B$. This process can be decomposed in three steps. The first step translates thm_A from $\mathcal{L}_A$ to the encoding of $\mathcal{L}_A$ in the logical framework U denoted by $U[\mathcal{L}_A]$. The second step translates thm_A from $U[\mathcal{L}_A]$ to $U[\mathcal{L}_B]$. Finally, the third step translates thm_A from $U[\mathcal{L}_B]$ to the proof system $\mathcal{L}_B$. While the first and last steps are total functions, the second step is, in general, a partial function since the translation is not always possible: For example, proof irrelevance is not expressible in HOL but it is in the Calculus of Constructions.

This is the solution we used to import an arithmetic library in Dedukti[STT$\forall_{\beta\delta}$]. Originally, this library has been implemented in the proof system Matita. The translation process that goes from Dedukti[Matita] to Dedukti[STT$\forall_{\beta\delta}$] is described in a separate paper [15].

1.2 How interoperability should work

The definition of a function or a type might differ between several proof systems. For example, in Coq, inductive types such as $\mathbb{N}$ are primitive while in HOL they are encoded. This is a problem for a generic translation:

- The definition of a type is not unique. For example, in Coq, natural numbers have at least three different definitions. Which one should be used?
- If one uses an intermediate system to translate these proofs, the encoding from the first system to the intermediate may *degrade* the original definition. Recreating the original type or function definition might be difficult. For example, the translation of the inductive type $\mathbb{N}$ is translated in Dedukti as four declarations ($\mathbb{N}$, 0, S and the recursor) with two rewrite rules. But from these declarations and the rewrite rules, it is difficult to identify the definition of an inductive type.

This implies that types and functions in general will be axiomatized during the transformations. For example, the arithmetic library we export into OpenTheory comes with 40 constants and 80 axioms to define. Fortunately, all of these axioms can be proven easily. Among these axioms, one can find

$$\forall x, \forall y, x = y \Rightarrow y = x$$

or

$$\forall n, \forall P, P\ n \Rightarrow (\forall m, (n \leq m \Rightarrow P\ m \Rightarrow P\ (S\ m))) \Rightarrow \forall y, (n \leq y \Rightarrow P\ y)$$

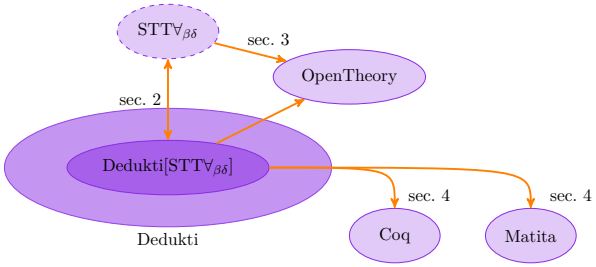

Figure 1: Contribution

Thus, the users of the library have to instantiate **once** the library with the definitions they want to use. We claim that this is the way interoperability should work because this is very flexible: There is no need to regenerate the whole library if the user wants to change one definition.

1.3 Contributions

The contributions of this paper are presented in Figure 1 and detailed below:

- We introduce a new logic namely STT$\forall_{\beta\delta}$ (Section 2) as well as its encoding in Dedukti;
- We give a translation from STT$\forall_{\beta\delta}$ to OpenTheory (Section 3);
- We describe the embedding from STT$\forall_{\beta\delta}$ to Coq and Matita (Section 4);
- We describe the translation of an arithmetic library from Dedukti[STT$\forall_{\beta\delta}$] to OpenTheory, Coq and Matita (Section 5).

As we will see in Section 2, Dedukti[STT$\forall_{\beta\delta}$] and STT$\forall_{\beta\delta}$ define the same logic. To simplify the presentation of the translation of this logic to OpenTheory, we choose to describe it as a translation from STT$\forall_{\beta\delta}$ even if the actual implementation is a translation from Dedukti[STT$\forall_{\beta\delta}$].

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

2.1 Dedukti

Dedukti is a logical framework that implements the $\lambda\Pi$-calculus modulo theory [6] [12], a calculus that extends the $\lambda\Pi$ calculus (also known as LF) [7] with rewrite rules. These rules can be used for the convertibility test. The syntax is the following:

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

and the type system of $\lambda\Pi$-calculus modulo theory are respectively presented in Figure 2. For simplicity, we do not present how to derive the judgment $\Gamma \vdash t \hookrightarrow u$ **wf** here, but it can be found in Saillard's thesis [12]. Roughly, a rewrite rule $t \hookrightarrow u$ is well typed when the types of t and u are convertible. One advantage of using rewrite rules is that more systems can be encoded in Dedukti using a *shallow* encoding where by shallow we mean an encoding having the two following properties: 1) a binder of the source language is translated as a binder in the second language (using HOAS (Higher-Order Abstract Syntax) [10] for example), and 2) the typing judgment in the source language is translated as a typing judgment in Dedukti. This means

$$\frac{}{[\,]\ \text{well-formed}}\ \text{DK EMPTY CTX} \qquad \frac{\Gamma \vdash A : s}{\Gamma, x : A\ \mathbf{wf}}\ \text{DK CTX SORT} \qquad \frac{\Gamma \vdash t \hookrightarrow u\ \mathbf{wf}}{\Gamma, t \hookrightarrow u\ \mathbf{wf}}\ \text{DK CTX} \hookrightarrow$$

$$\frac{\Gamma\ \text{well-formed}}{\Gamma \vdash \mathbf{Type} : \mathbf{Kind}}\ \text{CTX } \mathbf{Type\ Kind} \qquad \frac{\Gamma\ \text{well-formed} \qquad (x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{DK VAR}$$

$$\frac{\Gamma \vdash A : \mathbf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi x^A.\ B : s}\ \text{DK } \Pi$$

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

$$\frac{\Gamma \vdash t : \Pi x^A.\ B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\ t' : B[x \leftarrow t']}\ \text{DK APP} \qquad \frac{\Gamma \vdash A : s \qquad \Gamma \vdash t : A \qquad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}\ \text{DK} \equiv$$

Figure 2: $\lambda\Pi$ modulo theory typing system

that we can use the type checker of Dedukti to check directly if a term from the source language 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$\forall_{\beta\delta}$

STT$\forall_{\beta\delta}$ is an extension of Simple Type Theory 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[1] [5]. The STT$\forall_{\beta\delta}$ syntax is presented in Fig. 3. The type of propositions **Prop** and the type of functions $\to$, could be declared as type operators, of arity 0 and 2 respectively. Since they have a particular meaning for the typing judgment, we add them explicitly. Also, STT$\forall_{\beta\delta}$ allows to declare and define constants. Declaring constants is better for interoperability as discussed in section 1.2 but increases the number of axioms that need to be ultimately instantiated. The typing system and the proof system are presented in Fig. 4 and Fig. 5. Finally, we point out that we identify in STT$\forall_{\beta\delta}$ the terms t and t' if they are convertible up to β and δ (unfolding of constants).

STT$\forall_{\beta\delta}$ is sound and type checking is decidable. The proofs are provided in annex of this paper.

Theorem 1. *STT$\forall_{\beta\delta}$ is sound: the judgment $\emptyset \vdash \forall x^{\mathbf{Prop}}.\ x$ is not provable.*

Proof. We construct a set-theoretical model of STT$\forall_{\beta\delta}$. □

[1] Coquand's paper [5] shows also that omitting types annotations for polymorphic types would make the logic inconsistent

Type operator	p		
Type variable	X		
Monotype	A, B	$:\equiv$	$X \mid \mathbf{Prop} \mid A \to B \mid p\ A_1\ ...\ A_n$
Polytype	T	$:\equiv$	$A \mid \forall X.\ T$
Constant	cst		
Constant term	c	$:\equiv$	$cst \mid c\ A$
Term variable	x		
Monoterm	t, u	$:\equiv$	$x \mid c \mid \lambda x^A.\ t \mid t\ u \mid t \Rightarrow u \mid \forall x^A.\ t \mid \lambda X.\ t$
Polyterm	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	$\mathcal{P}$	$:\equiv$	$\Sigma; \Gamma; \Xi \vdash \tau$
MonoType well-formed			$\Sigma; \Gamma \vdash A$ **wf**
PolyType wellf-formed			$\Sigma; \Gamma \vdash T$ **wf**
Typing ctx well-formed			$\Sigma \vdash \Gamma$ **wf**
Constant ctx well-formed			Σ **wf**

Figure 3: STT$\forall_{\beta\delta}$ syntax

Theorem 2. *Type checking and proof checking in STT$\forall_{\beta\delta}$ are decidable.*

Proof. We only have to show that $\equiv_{\beta\delta}$ is decidable. This property follows from the fact that $\hookrightarrow_\beta \cup \hookrightarrow_\delta$ is a convergent term rewriting system. □

2.2.1 Equality in STT$\forall_{\beta\delta}$

In order to give further insights into STT$\forall_{\beta\delta}$, we give here an example that expresses polymorphic Leibniz equality denoted by $=_{\mathcal{L}}$ in STT$\forall_{\beta\delta}$. Its type will be $\forall X.\ X \to X \to \mathbf{Prop}$ and it can be implemented by the term $\lambda X.\ \lambda x^X.\ \lambda y^X.\ \forall P^{X \to \mathbf{Prop}}.\ P\ x \Rightarrow P\ y$. From this definition, it is possible to prove that $=_{\mathcal{L}}$ is reflexive, which is expressed by the statement $\forall X.\ \forall x^X.\ x =_{\mathcal{L}} x$, proved on the right.

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

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

The purpose of this section is to describe a shallow encoding of STT$\forall_{\beta\delta}$ in Dedukti. Thanks to HOAS [10], such a shallow encoding exists in Dedukti. In Figure 6, we present the signature used to encode terms from STT$\forall_{\beta\delta}$. The translation function is given in annexes.

For the types, we declare in Dedukti two symbols *type* and *ptype* that are used to encode monomorphic types and polymorphic types of STT$\forall_{\beta\delta}$. Therefore every type of STT$\forall_{\beta\delta}$ will be encoded as an object of Dedukti. That is why we use the symbol *term* to encode a STT$\forall_{\beta\delta}$

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

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

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

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

$$\frac{\Sigma;\Gamma, X \vdash T\ \mathbf{wf}}{\Sigma;\Gamma \vdash \forall X.\ T\ \mathbf{wf}}\ \text{S WF FORALL TY}$$

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

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

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

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

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

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

Figure 4: STT$\forall_{\beta\delta}$ typing system

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

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

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

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

Figure 5: STT$\forall_{\beta\delta}$ proof system

type to a Dedukti type. We add in the signature a symbol p to coerce a monomorphic type to a polymorphic type. Then we need symbols to represent type constructors of STT$\forall_{\beta\delta}$: The Dedukti's symbol *prop* encodes **Prop** while *arr* encodes $\to$. Each type constructor of arity n is encoded as a new Dedukti symbol of type $type \to \cdots \to type$ with $n+1$ occurrences of *type*. Finally, to encode $\forall$ at the type level, we use the Dedukti symbol $forallK_{type}$.

For the terms, since the encoding is shallow, we do not need symbols for abstractions and applications. In contrast, we need the two symbols $forall$ and *impl* that encode respectively the connectives $\forall$ and $\Rightarrow$. Then, we add the symbol $forallK_{prop}$ to encode polymorphic propositions. To encode a proposition into a Dedukti type, we use the symbol $proof$. Finally, rewrite rules transform a deep representation of STT$\forall_{\beta\delta}$ syntax to a shallow one, for instance the Dedukti rule $term\ (p\ (arr\ l\ r)) \hookrightarrow term\ l \to term\ r$ allows the Dedukti term $term\ (p\ (arr\ l\ r))$ to be the type of a Dedukti's abstraction.

2.3.1 A proof of reflexivity in Dedukti[STT$\forall_{\beta\delta}$]:

The translation of Leibniz equality in Dedukti[STT$\forall_{\beta\delta}$] is as follow. First, the type of $=_{\mathcal{L}}{}^2$ is translated as:

$$term\ (forallK_{type}\ (\lambda X.\ arr\ X\ (arr\ X\ prop))$$

then its definition is translated as

$$\lambda A.\ \lambda x^{term\ A}.\ \lambda y^{term\ A}.\ \forall P^{term\ (arr\ A\ prop)}.\ impl\ (P\ x)\ (P\ y)$$

Finally, the proof of `refl` is translated as

$$\lambda A.\ \lambda x^{term\ A}.\ \lambda P^{term\ (arr\ A\ prop)}.\ \lambda h^{proof\ (P\ x)}.\ h$$

that is of type

$$proof\ (forallK_{prop}\ (\lambda X.\ \forall x^{termX}.\ leibniz\ X\ x\ x))$$

[2] also written *leibniz* in its prenex form

$$
\begin{array}{rl}
\mathsf{type} : \mathbf{Type} & \mathsf{impl} : \mathsf{term}\ (\mathsf{p}\ (\mathsf{arr}\ \mathsf{prop}\ (\mathsf{arr}\ \mathsf{prop}\ \mathsf{prop}))) \\
\mathsf{arr} : \mathsf{type} \to \mathsf{type} \to \mathsf{type} & \mathsf{forallK}_{\mathsf{type}} : (\mathsf{type} \to \mathsf{ptype}) \to \mathsf{ptype} \\
\mathsf{prop} : \mathsf{type} & \mathsf{proof} : \mathsf{term}\ (\mathsf{p}\ \mathsf{prop}) \to \mathtt{Type} \\
\mathsf{ptype} : \mathbf{Type} & \mathsf{forall} : (\mathsf{t} :\ \mathsf{type}) \to\ \mathsf{term}\ (\mathsf{p}\ (\mathsf{arr}\ (\mathsf{arr}\ \mathsf{t}\ \mathsf{prop})\ \mathsf{prop})) \\
\mathsf{p} : \mathsf{type} \to \mathsf{ptype} & \mathsf{forallK}_{\mathsf{prop}} : (\mathsf{type} \to \mathsf{term}\ (\mathsf{p}\ \mathsf{prop})) \to \mathsf{term}\ (\mathsf{p}\ \mathsf{prop}) \\
\mathsf{term} : \mathsf{ptype} \to \mathbf{Type} &
\end{array}
$$

$$
\begin{array}{rl}
\mathsf{term}\ (\mathsf{p}(\mathsf{arr}\ \mathsf{l}\ \mathsf{r})) & \hookrightarrow \mathsf{term}\ (\mathsf{p}\ \mathsf{l}) \to \mathsf{term}\ (\mathsf{p}\ \mathsf{r}) \\
\mathsf{term}\ (\mathsf{forallK}_{\mathsf{type}}\ \mathsf{f}) & \hookrightarrow (\mathsf{x} : \mathsf{type}) \to \mathsf{term}\ (\mathsf{f}\ \mathsf{x}) \\
\mathsf{proof}\ (\mathsf{forall}\ \mathsf{t}\ \mathsf{f}) & \hookrightarrow (\mathsf{x} : \mathsf{term}\ (\mathsf{p}\ \mathsf{t})) \to \mathsf{proof}\ (\mathsf{f}\ \mathsf{x}) \\
\mathsf{proof}\ (\mathsf{impl}\ \mathsf{l}\ \mathsf{r}) & \hookrightarrow \mathsf{proof}\ \mathsf{l} \to \mathsf{proof}\ \mathsf{r} \\
\mathsf{proof}\ (\mathsf{forallK}_{\mathsf{prop}}\ \mathsf{f}) & \hookrightarrow (\mathsf{x} : \mathsf{type}) \to \mathsf{proof}\ (\mathsf{f}\ \mathsf{x})
\end{array}
$$

Figure 6: Signature for STT$\forall_{\beta\delta}$ in Dedukti

Type operators	p		
Type variables	X		
Types	A, B	$:\equiv$	X $\mid$ **Prop** $\mid A \to B \mid p\ A_1\ ...\ A_n$
Terms variables	x		
Terms	t,u	$:\equiv$	x $\mid$ c $\mid \lambda x^A.\ t \mid t\ u \mid t = u \mid c$
Typing Judgment	$\mathcal{T}$	$:\equiv$	$\Sigma; \Gamma \vdash \tau : T$
Proof Judgment	$\mathcal{P}$	$:\equiv$	$\Sigma; \Gamma; \Xi \vdash \tau$

Figure 7: OpenTheory syntax

which is the translation of

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

3 OpenTheory

HOL is a logic that is implemented in several systems with some minor differences. OpenTheory [8] is a tool that allows to share proofs between several implementations of HOL. Since we are targeting OpenTheory, we will mostly refer to the logic defined by OpenTheory. The logic behind OpenTheory comes from $\mathcal{Q}_0$ [1], a classical logic taking only equality as a primitive logical connective. Terms are those of Simply Typed Lambda Calculus with an equality symbol while the type system extends the one of the Simply Typed Lambda Calculus by declaring type operators and prenex polymorphism. The syntax and the proof system of OpenTheory can be found respectively in Fig. 7 and Fig. 8. The syntax being very similar to the one of STT$\forall_{\beta\delta}$, we have omitted the definitions of typing judgments.

$$\frac{}{t \vdash t}\ \text{OT ASSUME} \qquad \frac{\Gamma \vdash t = u}{\Gamma \vdash \lambda v.t = \lambda v.u}\ \text{OT ABSTHM}$$

$$\frac{\Gamma \vdash t_1 = u_1 \qquad \Delta \vdash t_2 = u_2}{\Gamma \cup \Delta \vdash t_1\ t_2 = u_1\ u_2}\ \text{OT APPTHM} \qquad \frac{}{t_1, \ldots, t_n \vdash u}\ \text{OT AXIOM}$$

$$\frac{c = t \in \Sigma}{\vdash c = t}\ \text{OT DELTA} \qquad \frac{}{\vdash (\lambda x.t)u = t[u/x]}\ \text{OT BETA}$$

$$\frac{\Gamma \vdash t \qquad \Delta \vdash u}{(\Gamma - u) \cup (\Delta - t) \vdash t = u}\ \text{OT DEDUCTANTISYM} \qquad \frac{\Gamma \vdash t \qquad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash u}\ \text{OT EQMP}$$

$$\frac{\Gamma \vdash t \qquad \Delta \vdash u}{(\Gamma - u) \cup \Delta \vdash t}\ \text{OT PROVEHYP} \qquad \frac{}{\vdash t = t}\ \text{OT REFL} \qquad \frac{\Gamma \vdash t}{\Gamma\sigma \vdash t\sigma}\ \text{OT SUBST}$$

$$\frac{\Gamma \vdash t = u}{\Gamma \vdash u = t}\ \text{OT SYM} \qquad \frac{\Gamma \vdash t_1 = t_2 \qquad \Delta \vdash t_2 = t_3}{\Gamma \cup \Delta \vdash t_1 = t_3}\ \text{OT TRANS}$$

Figure 8: OpenTheory proof system

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

One may notice that STT$\forall_{\beta\delta}$ and OpenTheory are quite similar. However, there are some differences that makes the translation from STT$\forall_{\beta\delta}$ and OpenTheory not so easy:

- 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 from the equality symbol, while in STT$\forall_{\beta\delta}$ they are defined from $\forall$ and $\Rightarrow$ connectives
- Prenex polymorphism in OpenTheory is implicit: All free type variables in OpenTheory are implicitly quantified while in STT$\forall_{\beta\delta}$ all quantifications are explicit

These differences lead to three different proof transformations:

- Encode the $\forall$ and $\Rightarrow$ connectives using the equality of OpenTheory
- Explicit each application of the conversion rule
- Finally, get rid of the type quantifier

OpenTheory is a classical logic while STT$\forall_{\beta\delta}$ is intuitionistic. This is not an issue here since intuitionistic logic is a fragment of classical logic.

3.2 From STT$\forall_{\beta\delta}$ to OpenTheory

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

A first idea to encode STT$\forall_{\beta\delta}$ proofs in OpenTheory would be to axiomatize 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 the use of implication. Since OpenTheory does not know what an implication is, such axioms would not be usable 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$\forall_{\beta\delta}$ are admissible. Such encoding is already known from $\mathcal{Q}_0$ [1]. This encoding is presented below and uses two other connectives that are $\top$ and $\wedge$ that can be defined as axiom in OpenTheory:

$$\top = \lambda x.\ x \qquad\qquad x \Rightarrow y = (x \wedge y) = x$$
$$x \wedge y = \lambda f.\ f\ x\ y = \lambda f.\ f\ \top\ \top \qquad\qquad \forall x.P = \lambda x.\ P = \lambda x.\ \top$$

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 later these connectives by the *true* connectives of HOL as long as these axioms can be proved regardless of their definition in HOL. These axioms are not too strong to satisfy because in HOL, extensionality of predicates[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 above. Below, we prove the admissibility of the S $\forall_I$ rule

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

using the derivation tree below, Γ is the translation of $\mathcal{C}$ in OpenTheory[4].

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

The right branch is closed thanks to the axiom defining $\forall$.

All the rules of STT$\forall_{\beta\delta}$ can be derived in a similar way. At the end of this translation, the syntax of the term is changed: $=$ becomes a new connective, while $\forall$ and $\Rightarrow$ become defined constants.

3.2.2 Eliminate β, δ reductions

In STT$\forall_{\beta\delta}$, the terms $\forall X.\ \forall x^X.\ x =_{\mathcal{L}} x$ and $\forall X.\ \forall x^X.\ \forall P.\ P\ x \to P\ x$ are convertible, but not in OpenTheory. The convertibility test in STT$\forall_{\beta\delta}$ will unfold the definition of $=_{\mathcal{L}}$ once, then it

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

[4] In OpenTheory, free variables such as x do not need to appear inside the context.

will apply twice a β-reduction. However, in OpenTheory, it is possible to prove

$$\Big(\curlywedge X.\ \forall x^X.\ x =_{\mathcal{L}} x\Big) = \Big(\curlywedge X.\ \forall x^X.\ \forall P.\ P\ x \to P\ x\Big)$$

The purpose of this section is to explain how it is possible to derive a proof of $t = t'$ in OpenTheory when $t \equiv_{\beta\delta} t'$ in STT$\forall_{\beta\delta}$. The decidability of type checking in STT$\forall_{\beta\delta}$ relies on the decidability of the conversion rule S Conv. Since the term rewriting system defined by $\hookrightarrow_\beta$ and $\hookrightarrow_\delta$ is convergent, we can decide whether $t \equiv_{\beta\delta} u$ by computing their normal forms t' and u', then checking they are equal up to α-conversion. OpenTheory has two rules to handle β and δ conversion:

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

Hence, one rewrite step will be translated as an equality. The same is true for a sequence of rewrite steps thanks to transitivity of equality. Therefore, the main difficulty is to show how to derive the OpenTheory judgment $t = u$ from the STT$\forall_{\beta\delta}$ judgment $t \hookrightarrow^*_{\beta\delta} u$.

In general, the OT beta and OT delta rules will be applied inside a term. Thus, we need to show that for any context C, the rule below is admissible:

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

the base case being either the rule OT beta or OT delta. In our setting, contexts can be defined by the following grammar:

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

Notice that our definition of contexts does not depend on the previous translation. However, to prove the admissibility of the rule ctxrule for the $\Rightarrow$ case for example, we will need to use its definition from equality.

Theorem 3. *For every context C, the rule* ctxrule *is admissible.*

Proof. This is done inductively on the structure of C. There are already two contextual rules in OpenTheory for equality to handle abstractions and applications. We need to derive the other contextual rules to handle STT$\forall_{\beta\delta}$ connectives that are: $\curlywedge, \Rightarrow$ and $\forall$. We show here the admissibility of the contextual rule for $\Rightarrow$ but the derivations for all the other rules are in annex.

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

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

Half of the proof is omitted here but the derivation tree is symmetric. This rule can be used to solve two context cases. In the case where $C \Rightarrow t$, we instantiate q and q' by t. Hence, the right premise is closed by the OpenTheory rule OT REFL. The case $t \Rightarrow C$ can be instantiated in a symmetric way. All the other cases can be derived in a similar way. □

3.2.3 Suppressing type quantifiers

OpenTheory implicitly quantifies over free types variables while in STT$\forall_{\beta\delta}$ this is done explicitly thanks to the $\forall$ on types. This implies that substitution in STT$\forall_{\beta\delta}$ is handled by the system while in OpenTheory, the user has to manage substitution to avoid capturing free type variables. For example, the following type in STT$\forall_{\beta\delta}$ $\forall Y.\ Y \to X[X := Y]$ is equal to $\forall Z.\ Z \to Y$ while in OpenTheory, the same type $Y \to X$ is equal to $Y \to Y$ using the OT SUBST rule with the substitution $X \mapsto Y$. This mechanism forces us to replace each bound variable by a fresh variable each time the bound variable is substituted. In STT$\forall_{\beta\delta}$, there are two rules that are concerned by this: S CST APP and S $\forall_E$. Renaming bound variables can be done easily using the SUBST rule of OpenTheory. For example, the rule S $\forall_E$

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

is translated as the OpenTheory proof

$$\cfrac{\cfrac{\mathcal{C} \vdash \tau \qquad Z\ fresh}{\mathcal{C} \vdash \tau[X := Z]}\ \text{SUBST}}{\mathcal{C} \vdash \tau[Z := A]}\ \text{SUBST}$$

The same thing can be done for the rule CST APP each time a constant is applied to a type inside the definition of a constant for example. The rule S $\forall_I$ is just removed because there is no need to introduce a quantifier anymore.

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

Going from STT$\forall_{\beta\delta}$ to Coq or Matita is easy since the Calculus of Inductive Constructions with universes can be seen as an extension of STT$\forall_{\beta\delta}$. Only three universes are needed for the translation: the impredicative universe *Prop* for **Prop**, $Type_1$ for monotypes and $Type_2$ for polytypes. The three *forall* constructions of STT$\forall_{\beta\delta}$, the arrow on types and the implication all translates to an instantiation of the product rule of the Calculus of Inductive of Constructions. Introduction rules can be implemented as abstractions while elimination rules as applications. Finally, type operators can be encoded as parameters of type : $Type_1 \to \cdots \to Type_1$. As an example, we show the result of our reflexivity proof from STT$\forall_{\beta\delta}$ to Coq[5]. Using Coq floating universes, we omit indices for universes. The equality $=_{\mathcal{L}}$ will be translated as

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

[5] $Type_0$ is also denoted Prop in Coq

	Dedukti[STT]	OpenTheory	Coq	Matita
size (mb)	1.5	41	0.6	0.6
translation time (s)	-	18	3	3
checking time (s)	0.1	13	6	2

Table 1: Arithmetic library translation

while the proof of reflexivity will be translated as the following definition

```
Definition refl_= : forall X:Type, forall x:X, x =_L x :=
  fun X:Type => fun x:X => fun h:(P x) => h.
```

5 The arithmetic library

We have implemented these transformations to an arithmetic library that comes from Matita [14]. From this library, we have extracted all the lemmas needed to prove the Fermat's little theorem (about 300 lemmas). In this library, we can find basic definitions of operators such as $+, \times$ but also the definition of a permutation over natural numbers or the definition of *big* operator such as Σ or Π. This library also proves basic results related to these definitions such as the commutativity of $+$ or basic results related to prime numbers. In table 1, we give some results related to the export of this library to OpenTheory, Coq and Matita.

These results show that the type checking time in OpenTheory is longer than in Dedukti, Coq or Matita. We suppose that this is mostly due to making the β and δ conversions explicit. In order to illustrate the usability of the translated library, we give below the translation of Fermat's little theorem in Coq:

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

The constants `prime`, `congruent` and `pred` come with a definition while the constants `exp`, `Not`, `O` and `S` are axiomatized and should be defined by the user. Our tool produces a *functor* that the user should instantiate whose parameters are the axiomatization of those notions. The user should instantiate it with reasonable definitions, proving the axioms. Then the theorem is ready to use. For example, the definition of `exp` has to satisfy the two following axioms:

```
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)).
```

The following definition (that comes from the standard library) satisfy those definitions:

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

For this arithmetic library, one has to define about 40 constants and prove about 80 axioms. All the constants definitions can be guessed from their name or from the axioms they have to satisfy, and hence the axioms are then easy to prove. This instantiation has been made in Coq [16].

6 Related Work

Cauderlier and Dubois already used Dedukti for interoperability in [3]. Their goal was to prove the sieve of Eratosthenes using HOL and Coq in combination. The main advantage of their work is that there is no need to export proofs outside the logical framework, instead everything is checked in Dedukti. However, mathematical objects in Dedukti, such as natural numbers, may have different representation, and therefore this approach may require theorems to transfer results about one representation to results about another representation.

In [9], Keller and Werner made a translation from HOL Light to Coq. Despite the fact that their source logic and their target logic is different from ours, they did not use any logical framework.

OpenTheory [8] in itself is an interoperability tool between the HOL family provers. However, OpenTheory is focused for systems that all implement a variant of Higher-Order Logic while this work aims to be more general.

Beluga [11] is an extension of LF that handles open terms thanks to contextual types. Beluga aims to be useful for interoperability since it is easier to write proof transformations in it.

The Foundational Proof Certificate project [4] aims at defining a generic methods for checking proofs. The approach seems more tuned towards self-contained proofs produced by, e.g., automated theorem provers, rather than libraries developed in proof assistants and rich logics developed in the rich logics of proof assistants.

7 Conclusion

In this paper, we showed how STT$\forall_{\beta\delta}$ is a simple logic that can be easily represented in the logical framework Dedukti and is powerful enough to express arithmetic proofs. We defined translations from STT$\forall_{\beta\delta}$ to other systems such as OpenTheory and implemented these translations from Dedukti. We applied it to an arithmetic library containing a proof of Fermat's little theorem. The differences between OpenTheory and STT$\forall_{\beta\delta}$ reveal three difficulties which we addressed in different phases of the translation. In contrast, we showed how the translation to Coq and Matita is easy since STT$\forall_{\beta\delta}$ can be seen as a subsystem of the Calculus of Inductive Constructions.

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 or Matita, 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. We are also interested to import more proofs in Dedukti[STT$\forall_{\beta\delta}$] that could then be exported.

Finally, we hope that this work is the beginning of a process that could lead to a standardization of libraries, starting with the arithmetic one (naming conventions, constants definitions or statement of important lemmas).

References

[1] Peter B. Andrews (1986): *An introduction to mathematical logic and type theory - to truth through proof*. Computer science and applied mathematics, Academic Press.

[2] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen & Enrico Tassi (2011): *The Matita Interactive Theorem Prover*. In Nikolaj Bjørner & Viorica Sofronie-Stokkermans, editors: *Automated*

Deduction - CADE-23 - 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 - August 5, 2011. Proceedings, *Lecture Notes in Computer Science* 6803, Springer, pp. 64–69. Available at https://doi.org/10.1007/978-3-642-22438-6_7.

[3] Raphaël Cauderlier & Catherine Dubois (2017): *FoCaLiZe and Dedukti to the Rescue for Proof Interoperability*. In Mauricio Ayala-Rincón & César A. Muñoz, editors: *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, *Lecture Notes in Computer Science* 10499, Springer, pp. 131–147. Available at https://doi.org/10.1007/978-3-319-66107-0_9.

[4] Zakaria Chihani, Dale Miller & Fabien Renaud (2013): *Foundational Proof Certificates in First-Order Logic*. In Maria Paola Bonacina, editor: *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, *Lecture Notes in Computer Science* 7898, Springer, pp. 162–177. Available at https://doi.org/10.1007/978-3-642-38574-2_11.

[5] Thierry Coquand (1986): *An Analysis of Girard's Paradox*. In: *Proceedings of the Symposium on Logic in Computer Science (LICS '86), Cambridge, Massachusetts, USA, June 16-18, 1986*, IEEE Computer Society, pp. 227–236.

[6] 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*, *Lecture Notes in Computer Science* 4583, Springer, pp. 102–117. Available at http://dx.doi.org/10.1007/978-3-540-73228-0_9.

[7] Robert Harper, Furio Honsell & Gordon D. Plotkin (1993): *A Framework for Defining Logics*. *J. ACM* 40(1), pp. 143–184. Available at http://doi.acm.org/10.1145/138027.138060.

[8] Joe Hurd (2011): *The OpenTheory Standard Theory Library*. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann & Rajeev Joshi, editors: *Third International Symposium on NASA Formal Methods (NFM 2011)*, *Lecture Notes in Computer Science* 6617, Springer, pp. 177–191. Available at https://doi.org/10.1007/3-540-60275-5_76.

[9] Chantal Keller & Benjamin Werner (2010): *Importing HOL Light into Coq*. In Matt Kaufmann & Lawrence C. Paulson, editors: *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, *Lecture Notes in Computer Science* 6172, Springer, pp. 307–322. Available at https://doi.org/10.1007/978-3-642-14052-5_22.

[10] F. Pfenning & C. Elliott (1988): *Higher-order Abstract Syntax*. *SIGPLAN Not.* 23(7), pp. 199–208. Available at http://doi.acm.org/10.1145/960116.54010.

[11] Brigitte Pientka (2008): *A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions*. In: *35th Annual ACM Symposium on Principles of Programming Languages (POPL'08)*, ACM, pp. 371–382. Available at https://doi.org/10.1145/1328438.1328483.

[12] Ronan Saillard (2015): *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique)*. Ph.D. thesis, Mines ParisTech, France. Available at https://tel.archives-ouvertes.fr/tel-01299180.

[13] The Coq Development Team (2017): *The Coq Proof Assistant, version 8.7.1*. Available at https://doi.org/10.5281/zenodo.1133970.

[14] The Matita development team (2018): *Arithmetic library*. Available at https://github.com/LPCIC/matita/tree/master/matita/matita/lib/arithmetics.

[15] François Thiré (2018): *Interoperability in Dedukti: From the Calculus of Inductive Constructions to an extension of HOL*. http://www.lsv.fr/~fthire/research/interop/index.php.

[16] François Thiré (2018): *Sharing a library between proof assistants: reaching out to the HOL family*. http://www.lsv.fr/~fthire/research/sttforall/index.php.