Dedukti: a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory

Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, Ronan Saillard

Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory

Ali Assaf[1], Guillaume Burel[2], Raphaël Cauderlier[3], David Delahaye[4], Gilles Dowek[5], Catherine Dubois[2], Frédéric Gilbert[6], Pierre Halmagrand[7], Olivier Hermant[8], and Ronan Saillard[8]

1 **Inria and École polytechnique,** ali.assaf@inria.fr
2 **ENSIIE,** {guillaume.burel,catherine.dubois}@ensiie.fr
3 **IRIF, University Paris Diderot, and CNRS,** raphael.cauderlier@irif.fr
4 **CNAM and Université de Montpellier,** david.delahaye@umontpellier.fr
5 **Inria and École Normale de Supérieure de Cachan,** gilles.dowek@ens-cachan.fr
6 **École des Ponts, Inria, and CEA,** frederic.gilbert@inria.fr
7 **CNAM and Inria,** pierre.halmagrand@inria.fr
8 **MINES ParisTech,** {olivier.hermant,ronan.saillard}@mines-paristech.fr

Abstract

Dedukti is a Logical Framework based on the λΠ-Calculus Modulo Theory. We show that many theories can be expressed in Dedukti: constructive and classical predicate logic, Simple type theory, programming languages, Pure type systems, the Calculus of inductive constructions with universes, etc. and that permits to used it to check large libraries of proofs developed in other proof systems: Zenon, iProver, FoCaLiZe, HOL Light, and Matita.

1 Introduction

Defining a theory, such as arithmetic, geometry, or set theory, in predicate logic [49] just requires to choose function and predicate symbols and axioms expressing the meaning of these symbols. Using this way a single logical framework to define all these theories has many advantages.

First, it requires less efforts, as the logical connectives, $\wedge$, $\vee$, $\forall$, etc. and their associated deduction rules are defined, in the framework, once and for all and need not be redefined for each theory. Similarly, the notions of proof, model, etc. are defined once and for all. And general theorems, such as the soundness and the completeness theorems, can be proved once and for all.

Another advantage of using such a logical framework is that this induces a partial order between theories. For instance, Zermelo-Fraenkel set theory with the axiom of choice (ZFC) is an extension of Zermelo-Fraenkel set theory (ZF), as it contains the same axioms, plus the axiom of choice. It is thus obvious that any theorem of ZF is provable in ZFC, and for each theorem of ZFC, we can ask the question of its provability in ZF. Several theorems of ZFC that are provable in ZF have been identified, and these theorems can be used in extensions of ZF that are inconsistent with the axiom of choice.

Finally, using such a common framework permits to combine, in a proof, lemmas proved in different theories: if $\mathcal{T}$ is a theory expressed in a language $\mathcal{L}$ and $\mathcal{T}'$ a theory expressed in a language $\mathcal{L}'$, if A is expressed in $\mathcal{L} \cap \mathcal{L}'$, $A \Rightarrow B$ is provable in $\mathcal{T}$, and A is provable in $\mathcal{T}'$, then B is provable in $\mathcal{T} \cup \mathcal{T}'$.

Despite these advantages, several logical systems have been defined, not as theories in predicate logic, but as independent systems: Simple type theory [29, 6], also known as Higher-order logic, is defined as an independent system—although it is also possible to express it as a theory in predicate logic. Similarly, Intuitionistic type theory [54], the

Calculus of constructions [31], the Calculus of inductive constructions [59], etc. are defined as independent systems. As a consequence, it is difficult to reuse a formal proof developed in an automated or interactive theorem prover based on one of these formalisms in another, without redeveloping it. It is also difficult to combine lemmas proved in different systems: the realm of formal proofs is today a tower of Babel, just like the realm of theories was, before the design of predicate logic.

The reason why these formalisms have not been defined as theories in predicate logic is that predicate logic, as a logical framework, has several limitations, that make it difficult to express some logical systems.

1. Predicate logic does not allow the use of bound variables, except those bound by the quantifiers $\forall$ and $\exists$. For instance, it is not possible to define, in predicate logic, a unary function symbol $\mapsto$ that would bind a variable in its argument.
2. Predicate logic ignores the propositions-as-types principle, according to which a proof π of a proposition A is a term of type A.
3. Predicate logic ignores the difference between deduction and computation. For example, when Peano arithmetic is presented in predicate logic, there is no natural way to compute the term 2×2 into 4. To prove the theorem $2 \times 2 = 4$, several deduction steps need to be used while a simple computation would have sufficed [61].
4. Unlike the notions of proof and model, it is not possible to define, once and for all, the notion of cut in predicate logic and to apply it to all theories expressed in predicate logic: a specific notion of cut must be defined for each theory.
5. Predicate logic is classical and not constructive. Constructive theories must be defined in another logical framework: constructive predicate logic.

This has justified the development of other logical frameworks, that address some of these problems. Problem 1 has been solved in extensions of predicate logic such as λ-Prolog [56] and Isabelle [60]. Problems 1 and 2 have been solved in an extension of predicate logic, called the Logical framework [47], also known as the λΠ-calculus, and the λ-calculus with dependent types. Problems 3 and 4 have been solved in an extension of predicate logic, called Deduction modulo theory [40]. Combining the λΠ-calculus and Deduction modulo theory yields the λΠ-calculus modulo theory [32], a variant of Martin-Löf's logical framework [58], which solves problems 1, 2, 3, and 4.

Problem 5 has been addressed [38, 48, 62] and will be discussed in Section 5.

The expressivity of the λΠ-calculus modulo theory has already been discussed [32]. We have shown that all functional Pure type systems [14], in particular the Calculus of constructions, could be expressed in this framework.

Our goal, in this paper, is twofold: first, we want to go further and show that other systems can be expressed in the λΠ-calculus modulo theory, in particular classical systems, logical systems containing a programming language as a subsystem, Simple type theory, and extensions of the Calculus of constructions with universes and inductive types.

Second, we want to demonstrate this expressivity, not just with adequacy theorems, but also by showing that large libraries of formal proofs coming from automated and interactive theorem provers can be translated to and checked in Dedukti, our implementation of the λΠ-calculus modulo theory. To do so, we translate to Dedukti proofs developed in various systems: the automated theorem proving systems Zenon modulo, iProverModulo, the system containing a programming language as a subsystem FoCaLiZe, and the interactive theorem provers HOL Light and Matita. Expressing formal proofs, coming from different systems, in a single logical framework is a first step towards reverse engineering these proofs, that is precisely analyzing the theories in which these theorems can be proved. It is also a

first step towards interoperability, that is building proofs assembling lemmas developed in different systems. Finally, it is also a way to audit formal proofs develop in these systems by checking them in a different, independent, system.

We show this way that the logical framework approach scales up and that it is mature enough to start imagining a single distributed library of formal proofs expressed in different theories, developed in different systems, but formulated in a single framework.

We present here a synthesis of the state of the art about the $\lambda\Pi$-calculus modulo theory, its implementation Dedukti, and the expression of theories in this framework. In Sections 4, 5, 6, 7, and 8, we shall present a number of examples of theories that have been expressed in Dedukti and of libraries of proofs expressed in these theories. Before that, we shall present the $\lambda\Pi$-calculus modulo theory, in Section 2, and the Dedukti system, in Section 3.

2 The $\lambda\Pi$-calculus modulo theory

We present the $\lambda\Pi$-calculus modulo theory in two steps. We first present, in Section 2.2, a very general system, not caring about the decidability of type-checking. Then, in Section 2.3, we discuss restrictions of this system that enforce the decidability of type-checking.

2.1 The $\lambda\Pi$-calculus

The $\lambda\Pi$-calculus is an extension of the simply typed λ-calculus with dependent types. In this calculus, types are just terms of a particular type $Type$, some symbols have the type $A \to Type$, and can be applied to a term of type A to build a type depending on this term. A type $Kind$ is introduced to type the terms $Type$, $A \to Type$, etc. and the arrow $A \to B$ is extended to a dependent product $\Pi x : A\ B$. Finally, besides the usual typing judgements, the typing rules use another kind of judgement expressing the well-formedness of a context. The typing rules are the following.

Well-formedness of the empty context

$$\frac{}{[\,] \text{ well-formed}}$$

Declaration of a type or type family variable

$$\frac{\Gamma \vdash A : Kind}{\Gamma, x : A \text{ well-formed}}$$

Declaration of an object variable

$$\frac{\Gamma \vdash A : Type}{\Gamma, x : A \text{ well-formed}}$$

$Type$

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash Type : Kind}$$

Variable

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash x : A}\ x : A \in \Gamma$$

Product (for kinds)

$$\frac{\Gamma \vdash A : Type \quad \Gamma, x : A \vdash B : Kind}{\Gamma \vdash \Pi x : A\ B : Kind}$$

Product (for types)

$$\frac{\Gamma \vdash A : Type \quad \Gamma, x : A \vdash B : Type}{\Gamma \vdash \Pi x : A\ B : Type}$$

Abstraction (for type families)

$$\frac{\Gamma \vdash A : Type \quad \Gamma, x : A \vdash B : Kind \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A\ t : \Pi x : A\ B}$$

Abstraction (for objects)

$$\frac{\Gamma \vdash A : Type \quad \Gamma, x : A \vdash B : Type \quad \Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A\ t : \Pi x : A\ B}$$

Application

$$\frac{\Gamma \vdash t : \Pi x : A\ B \quad \Gamma \vdash t' : A}{\Gamma \vdash (t\ t') : (t'/x)B}$$

Conversion

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A : Type \quad \Gamma \vdash B : Type}{\Gamma \vdash t : B}\ A \equiv_\beta B$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash A : Kind \quad \Gamma \vdash B : Kind}{\Gamma \vdash t : B}\ A \equiv_\beta B$$

2.2 The λΠ-calculus modulo theory

The λΠ-calculus modulo theory is an extension of the λΠ-calculus. In the λΠ-calculus modulo theory, the contexts contain variable declarations—like in the λΠ-calculus—and also rewrite rules. The conversion rule is extended in order to take these rewrite rules into account.

2.2.1 Local and global contexts

There are two notions of context in the λΠ-calculus modulo theory: *global* contexts and *local* ones. Global contexts contain variable declarations and rewrite rules, but local ones contain object variable declarations only, that is declaration of variables whose type has type $Type$ and not $Kind$. A third kind of judgement is introduced to express that Δ is a local context in Γ, and two rules to derive such judgements.

Locality of the empty context

$$\frac{\Gamma \text{ well-formed}}{\Gamma \vdash [\,] \text{ local}}$$

Declaration of an object variable in a local context

$$\frac{\Gamma \vdash \Delta \text{ local} \quad \Gamma, \Delta \vdash A : Type}{\Gamma \vdash \Delta, x : A \text{ local}}$$

The following lemma is proved with a simple induction on the structure of the derivation of the judgment "Δ local".

▶ **Lemma 1.** *If Δ is local in Γ, then Γ, Δ is a well-formed global context.*

Thus, defining a local context is just a way to express that some end segment of the context contains object variables only.

A rewrite rule in a global context Γ is a pair of terms $\langle l, r \rangle$, where l is a term of the form $f\ u_1\ ...\ u_n$ for some variable f, together with a context Δ local in Γ. We note it $l \longrightarrow^{\Delta} r$.

The notion of well-formed global context is extended to allow the declaration of rewrite rules

$$\frac{\Gamma \text{ well-formed} \quad \Gamma \vdash \Delta \text{ local}}{\Gamma, l \longrightarrow^{\Delta} r \text{ well-formed}}$$

Note that, at this stage, l and r can have different types or be not well-typed at all. Conditions will be added in Section 2.3.

2.2.2 Conversion

If $l \longrightarrow^{\Delta} r$ is a rewrite rule and σ is a substitution binding the variables of Δ, we say that the term σl *rewrites* to the term σr.

If Γ is a well-formed global context, then the rewriting relation generated by β-reduction and Γ, noted $\to_{\beta\Gamma}$, is the smallest relation, closed by context, such that if t rewrites to u, for some rule in Γ, or if t β-reduces to u, then $t \to_{\beta\Gamma} u$. The congruence $\equiv_{\beta\Gamma}$ is the reflexive-symmetric-transitive closure of the relation $\to_{\beta\Gamma}$.

The conversion rules of the $\lambda\Pi$-calculus modulo theory are the same as the conversion rules of the $\lambda\Pi$-calculus except that the congruence $\equiv_{\beta}$ is replaced by $\equiv_{\beta\Gamma}$

$$\frac{\Gamma \vdash A : Type \quad \Gamma \vdash B : Type \quad \Gamma \vdash t : A}{\Gamma \vdash t : B} A \equiv_{\beta\Gamma} B$$

$$\frac{\Gamma \vdash A : Kind \quad \Gamma \vdash B : Kind \quad \Gamma \vdash t : A}{\Gamma \vdash t : B} A \equiv_{\beta\Gamma} B$$

2.2.3 Subject reduction

Subject reduction expresses that reduction preserves the type of a term. This property does not hold in general in the $\lambda\Pi$-calculus modulo theory. However it does if we have the two following properties: *well-typedness* of rewrite rules and *product compatibility* [13]. Henceforth, unless stated otherwise, $\Gamma \vdash t : T$ stands for "the judgement $\Gamma \vdash t : T$ is derivable".

A rewrite rule is *well-typed* if it is type preserving.

▶ **Definition 2.** Let $l \longrightarrow^{\Delta_0} r$ be a rewrite rule in the context Γ_0 and Γ be a well-formed extension of Γ_0. We say that $l \longrightarrow^{\Delta_0} r$ is *well-typed* for Γ if for any substitution σ binding the variables of Δ_0

$$\text{if } \Gamma \vdash \sigma l : T, \text{ then } \Gamma \vdash \sigma r : T$$

▶ **Definition 3.** A well-formed global context Γ satisfies the *product compatibility* property if for any products $\Pi x : A_1\ B_1$ and $\Pi x : A_2\ B_2$ well-typed in Γ

$$\text{if } \Pi x : A_1\ B_1 \equiv_{\beta\Gamma} \Pi x : A_2\ B_2 \text{ then } A_1 \equiv_{\beta\Gamma} A_2 \text{ and } B_1 \equiv_{\beta\Gamma} B_2.$$

Note that this property immediately follows from the confluence of $\beta\Gamma$-reduction, but it is weaker.

From well-typedness of rewrite rules and product compatibility, we can prove subject reduction.

▶ **Lemma 4** (Subject Reduction, Theorem 2.6.22 in [65]). *Let Γ be a global context satisfying product compatibility and whose rewrite rules are well-typed. If $\Gamma \vdash t_1 : T$ and $t_1 \to_{\beta\Gamma} t_2$ then $\Gamma \vdash t_2 : T$.*

Using product compatibility we can also show that a term has, at most, one type modulo conversion.

▶ **Lemma 5** (Uniqueness of Types, Theorem 2.6.34 in [65]). *Let Γ be a global context satisfying the product compatibility property. If $\Gamma \vdash t : T_1$ and $\Gamma \vdash t : T_2$ then $T_1 \equiv_{\beta\Gamma} T_2$.*

2.2.4 Rules and rule schemes

In predicate logic, some theories are defined with a finite set of axioms, in which case these axioms can just be enumerated. Some others, such as arithmetic or set theory, are defined with an infinite but decidable set of axioms: an axiom scheme. In a specific proof however, only a finite number of such axioms can be used. Thus, provability of a proposition A in a theory $\mathcal{T}$ having axiom schemes can either be defined as the fact that the infinite sequent $\mathcal{T} \vdash A$ has a proof, or as the fact that there exists a finite subset Γ of $\mathcal{T}$, such that the finite sequent $\Gamma \vdash A$ has a proof.

In the same way, some theories in the λΠ-calculus modulo theory may be defined with an infinite set of symbols and an infinite set of rewrite rules, recognized by an algorithm. In this case, as the contexts are always finite in the λΠ-calculus modulo theory, the derivability of a typing relation $t : A$ must be defined as the existence of a finite context Γ, such that the judgment $\Gamma \vdash t : A$ is derivable.

2.3 Effective subsystems

In the calculus presented in Section 2.2, proof-checking is not always decidable because, when using the conversion rule to reason modulo the congruence,

$$\frac{\Gamma \vdash A : s \quad \Gamma \vdash B : s \quad \Gamma \vdash t : A}{\Gamma \vdash t : B} A \equiv_{\beta\Gamma} B$$

the conversion sequence justifying $A \equiv_{\beta\Gamma} B$ is not recorded in the conclusion. Thus, we need to restrict to *effective* subsystems, that is systems where the rewrite rules are restricted in such a way that the congruence $\equiv_{\beta\Gamma}$ is decidable. A natural subsystem is obtained by restricting the rewrite rules to form, together with the β-reduction rule, a terminating and confluent rewrite system. This way, the congruence can be decided by checking equality of normal forms.

As β-reduction does not terminate on untyped terms, termination can hold only for typed ones. Therefore, the subject reduction property is essential to prove the termination of the calculus. As we have seen in Section 2.2.3, subject reduction is a consequence of well-typedness of rewrite rules and product compatibility, which is itself a consequence of confluence. Thus confluence should be proved first, without assuming termination or restricting ourselves to typed terms.

The rest of this section discusses effective conditions for confluence and well-typedness.

2.3.1 Higher-order rewriting

Confluence of the rewrite relation together with β-reduction is not general enough. Indeed, such a confluence result may easily be lost when allowing λ-abstraction in the left-hand side of a rewrite rule, a useful feature—see, for instance, Section 8.3.

For example, let $\Delta = f : \mathit{Real} \to \mathit{Real}$, and consider the rewrite rule explaining how to differentiate the function $\lambda x : \mathit{Real}\ (\sin\ (f\ x))$

$$(D\ (\lambda x : \mathit{Real}\ (\sin\ (f\ x)))) \longrightarrow^{\Delta} (\times\ (D\ (\lambda x : \mathit{Real}\ (f\ x)))\ (\lambda x : \mathit{Real}\ (\cos\ (f\ x))))$$

where $\times$ is the pointwise product of functions. This rule overlaps with β-reduction and creates the critical peak

$$(D\ (\lambda x\ (\sin\ x))) \longleftarrow (D\ (\lambda x\ (\sin\ ((\lambda y\ y)\ x)))) \longrightarrow (\times\ (D\ (\lambda x\ ((\lambda y\ y)\ x)))\ (\lambda x\ (\cos\ ((\lambda y\ y)\ x))))$$

that cannot be joined. This rewrite system with β-reduction is therefore not confluent.

Still, $\equiv_{\beta\Gamma}$ is decidable, as some form of confluence holds. For this, we need a richer rewrite relation: rewriting modulo β-equivalence, also known as higher-order rewriting.

Higher-order rewriting requires higher-order matching: the term t matches the term l if there is a substitution σ such that σl is β-convertible to t. Higher-order matching is undecidable in general [36, 52], but it is decidable if we restrict to Miller's patterns [55]. Thus, we restrict the left-hand sides of the rewrite rules to be patterns.

▶ **Definition 6.** A β-normal term $f\ u_1\ ...\ u_n$ is a pattern with respect to Δ if f is a variable not declared in Δ, and the variables declared in Δ are applied to pairwise distinct bound variables.

For example, if f is declared in Δ, then the term $D\ (\lambda x\ (\sin\ (f\ x)))$ is a pattern, but neither $D\ (\lambda x\ (f\ (\sin\ x)))$ nor $D\ ((\lambda x\ (\sin\ x))\ y)$ are.

Now, if we consider the derivative rule as a higher-order rule, we have

$$D\ (\lambda x\ (\sin\ x)) \longrightarrow \times\ (D\ (\lambda x\ x))\ (\lambda x\ (\cos\ x))$$

thus the peak above is now joinable, and, more generally, confluence holds.

Given a pattern l and a term t there are different substitutions σ such that $\sigma l \equiv_{\beta\Gamma} t$ and some of them are not be well-typed. To get a precise and type-safe definition of higher-order-rewriting, we use the concepts of higher-order rewrite system [57]. Remark, however, that our notion of higher-order rewriting does not require the identification of η-equivalent terms as for higher-order rewrite systems. See [63] for more details. This also gives us powerful confluence criteria such as confluence for development-closed systems [67].

The reason why we are interested in the confluence of this extended higher-order rewrite relation is that, when combined with β-reduction, first- and higher-order rewriting yield the same congruence. Therefore, if higher-order rewriting is confluent and terminating, $\equiv_{\beta\Gamma}$ is decidable.

2.3.2 Typing rewrite rules

As we want reduction to preserve typing, we must check that the rewrite rules are well-typed in the sense of Definition 2. To do so, we introduce two new judgments: $\Gamma \vdash l \longrightarrow^{\Delta} r$ expressing that the rule $l \longrightarrow^{\Delta} r$ is well-typed in Γ and Γ *strongly well-formed*, expressing that all the rules declared in Γ are well-typed. This judgment is defined by the rules

$$\frac{}{[\,]\ \text{strongly well-formed}}$$

$$\frac{\Gamma\ \text{strongly well-formed} \quad \Gamma \vdash A : \mathit{Type}}{\Gamma, x : A\ \text{strongly well-formed}}$$

$$\frac{\Gamma\ \text{strongly well-formed} \quad \Gamma \vdash A : \mathit{Kind}}{\Gamma, x : A\ \text{strongly well-formed}}$$

$$\frac{\Gamma\ \text{strongly well-formed} \quad \Gamma \vdash l \longrightarrow^{\Delta} r}{\Gamma, l \longrightarrow^{\Delta} r\ \text{strongly well-formed}}$$

Defining an effective rule for the judgment $\Gamma \vdash l \longrightarrow^{\Delta} r$, that fits Definition 2, is more difficult. This definition quantifies over all possible substitutions, so is not effective. Thus, we need to under-approximate this notion and find rules that express sufficient conditions for well-typedness. A first attempt is to require l and r to be well-typed and to have the same type in Γ, Δ

$$\frac{l \text{ is a pattern} \quad \Gamma, \Delta \vdash l : T \quad \Gamma, \Delta \vdash r : T \quad dom(\Delta) \subseteq FV(l)}{\Gamma \vdash l \longrightarrow^{\Delta} r}$$

But this condition is too restrictive as shown by the following example.

Consider a context containing a variable A of type $Type$, a variable $Vector$ of type $nat \to Type$, a variable Nil of type $(Vector\ 0)$, and a variable $Cons$ of type $\Pi n : nat\ (A \to (Vector\ n) \to (Vector\ (S\ n)))$. Assume that we want to define the $Tail$ function of type $\Pi n : nat\ ((Vector\ (S\ n)) \to (Vector\ n))$.

A first possibility is to use the non left-linear rule

$$(Tail\ n\ (Cons\ n\ a\ l)) \longrightarrow l$$

but proving confluence of a non-linear rewrite system, not assuming termination—as underlined above, confluence comes first—, is difficult. Thus, we might prefer the linear rule

$$(Tail\ n\ (Cons\ m\ a\ l)) \longrightarrow l$$

This rule is well-typed in the sense of Definition 2, but its left-hand side is not well-typed: $(Cons\ m\ a\ l)$ has type $(Vector\ (S\ m))$ and $(Tail\ n)$ expects a term of type $(Vector\ (S\ n))$. So we need a weaker condition, in order to accept this rule [16, 65].
Typing the term $\sigma(Tail\ n\ (Cons\ m\ a\ l))$ generates the constraint

$$(Vector\ (S\ \sigma n)) \equiv (Vector\ (S\ \sigma m))$$

Using the injectivity of the functions $Vector$ and S, this constraint boils down to

$$\sigma n \equiv \sigma m$$

so all the substitutions σ, such that $\sigma(Tail\ n\ (Cons\ m\ a\ l))$ is well-typed, have the form $\eta\tau$ where η is an arbitrary substitution and $\tau = p/m, p/n$. The substitution τ is the most general typing substitution for l. Both the terms $\tau(Tail\ n\ (Cons\ m\ a\ l))$ and τl have type $(Vector\ p)$. So, for all η, $\eta\tau(Tail\ n\ (Cons\ m\ a\ l))$ and $\eta\tau l$ have the same type and the rule is well-typed.

Therefore, sufficient condition for the rule $l \longrightarrow r$ to be well-typed is that l has a most general typing substitution τ and τl and τr have the same type. This leads to the rule

$$\frac{l \text{ is a pattern} \quad \Gamma, \Delta \vdash \tau l : T \quad \Gamma, \Delta \vdash \tau r : T \quad dom(\Delta) \subseteq FV(l)}{\Gamma \vdash l \longrightarrow^{\Delta} r}$$

where τ is the most general typing substitution for l in Γ, Δ.

This rule subsumes the first one as, if l is well-typed, it has the identity as a most general typing substitution.

▶ **Lemma 7** (Theorem 3.5.8 in [65]). *Let Γ be a well-formed context satisfying the product compatibility. If $\Gamma \vdash l \longrightarrow^{\Delta} r$, then $(l \longrightarrow^{\Delta} r)$ is well-typed in Γ.*

Finally, we can prove the effectiveness theorem.

▶ **Theorem 8** (Effectiveness, Theorem 4.5.10 and Theorem 6.3.1 in [65]). *If Γ is a strongly well-formed context such that higher-order rewriting is confluent then Γ satisfies the subject reduction property. Moreover, if higher-order rewriting is terminating on well-typed terms, then typing is decidable.*

3 Dedukti

DEDUKTI (`http://dedukti.gforge.inria.fr/`) is a proof-checker for the λΠ-calculus modulo theory. This system is not designed to develop proofs, but to check proofs developed in other systems. In particular, it enjoys a minimalistic syntax. It gives a concrete syntax to the λΠ-calculus modulo theory, defined in Section 2.

The typing algorithm relies on the confluence and termination of the higher-order rewrite system together with β-reduction, to decide whether two terms are convertible.

The text with a grey background is executable DEDUKTI code. To avoid repetitions part of this code is omitted, the complete version can be found on the DEDUKTI website.

3.1 A proof-checker for the λΠ-calculus modulo theory

As in the λΠ-calculus modulo theory, there are two kinds of constructs available in DEDUKTI: variable declaration and rewrite rule declaration. The only available constant at the root of a file is `Type`, that corresponds to *Type* above. The sort *Kind* is not part of the input syntax and is used only internally.

One only needs to declare the variables and rewrite rules, that will take place in the *global context*. We can declare a variable `nat`, together with its type as follows

```
nat : Type.
```

The Π-types $\Pi x : A\ B$ are represented with a single-line arrow `x:A -> B`, which is thus a binder. When there is no dependency, `x` can be dropped, as in

```
0 : nat.
S : nat -> nat.
```

Rewrite rules, together with their local context Δ are declared in the form `[local context] l --> r`. The types of the variables in the local context are inferred automatically. For instance

```
def plus : nat -> nat -> nat.
[n] plus 0 n --> n.
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).
```

A head symbol may be defined through several rewrite rules, in this case, we allow their introduction all at once in the context, instead of one by one, as is done in Section 2.2 and just above. We express that the rules are defined all at once by writing just one dot at the end of the sequence.

```
[n] plus 0 n --> n
[n1, n2] plus (S n1) n2 --> S (plus n1 n2).
```

This has a direct impact on how the rules are typed. In the former case, the first rule can be used to type the second, while in the latter, it cannot.

Rewrite rules having the same head symbol need not be added sequentially, they can be separated by an arbitrary number of steps. In complex cases, the well-typing of a rewrite rule on a symbol `g` may depend on a rewrite rule on a symbol `f`, while further rewrite rules on `f` may depend on those previously declared on `g`.

Moreover, one can add as many rules as wanted for a given head symbol, as long as confluence is preserved. Checking confluence is out of the scope of DEDUKTI itself, and is

a separate concern. To support confluence checking by external tools, an export feature towards the old TPDB format [53], used in confluence competitions, is available. For instance, we can add the following rule to make `plus` associative.

```
[n1, n2, n3] plus n1 (plus n2 n3) --> plus (plus n1 n2) n3.
```

Defining a constant, standing for a complex expression, is identical to defining a rewrite rule in an empty rewrite context. For convenience, a special syntax is allowed

```
def two := S (S 0).
```

Finally, λ-abstraction is expressed with the double arrow `=>`

```
def K2 := x:nat => two.
```

Notice, that type annotation on the variable is optional [15], when it can be reconstructed by bidirectional typechecking [19, 28, 30]. In this example, it is mandatory.

The semantic difference between variables, constants and constructors is purely in the eyes of the user, Dedukti treats them in the same way.

It is also possible to split definitions in several files, that can be imported. There also exist meta-directives to the checker, for instance to ask the reduction of a term in head normal form. Refer to the user manual [33], for the details.

3.2 Static and definable symbols

We have seen in Section 2.3.2, that a key step for accepting the linear rule

```
tail n (Cons m a l) --> l
```

is to compute the most general typing substitution for its left-hand side, `tail n (Cons m a l)`. The existence of this substitution depends on the injectivity of the symbols `vector` and `S`. So, Dedukti needs to be aware that these symbols are injective with respect to the congruence $\equiv_{\beta\Gamma}$. A simple way to ensure this property is to forbid rewrite rules whose head symbol is `vector` or `S`. Such a symbol, whose appearance at the head of rewrite rules is forbidden, is called a *static* symbol and is introduced by a declaration containing only the symbol name and its type, such as `vector` in the declarations

```
A: Type.

vector: nat -> Type.
Nil : vector 0.
Cons : n : nat -> A -> vector n -> vector (S n).
```

Symbols such as `tail` which may appear at the head of rewrite rules are called *definable* and are declared using the keyword `def`

```
def tail: n:nat -> vector (S n) -> vector n.
```

Then, the rule

```
[n, m, a, l] tail n (Cons m a l) --> l.
```

is accepted.

3.3 Wildcards

Instead of introducing new names for variables which will be inferred by the unification step, the wildcard symbol _ can be used in the left-hand side as an unnamed pattern, matching any term, so we might write

```
[n, a, l] tail n (Cons _ a l) --> l.
```

or even

```
[l] tail _ (Cons _ _ l) --> l.
```

3.4 Guards

For the rare cases where the injectivity of a definable symbol would be needed to solve a unification problem, or when it would be preferable to use a non-normal term in the left-hand side of a rewrite rule, Dedukti implements a *guard* mechanism.

For this example, the user can add brackets—the guard—around the subterm whose shape is known by typing constraints.

```
[n, a, l] tail n (Cons {n} a l) --> l.
```

Dedukti will type-check the rule as if there were no brackets, but the rewrite rule declared will be the linear one, that is to say the rule obtained by replacing the guard by a fresh variable.

Dedukti cannot check in general the validity of the guard, namely that it indeed follows from typing constraints, since it is undecidable. Thus, in order to guarantee in any case a type-safe reduction, Dedukti checks, each time the rule is used, that the typing constraints are satisfied. If they are not, this means that the rewrite rule is not well-typed and Dedukti fails with an error message.

4 Constructive predicate logic and Deduction modulo theory

The first example of theory that can be embedded in Dedukti is constructive predicate logic. As the type system is derived from the $\lambda\Pi$-calculus, we shall see that those embeddings can be defined quite naturally, and that, with rewriting, we can endow them with a computational content and make them shallow.

4.1 Minimal predicate logic

Even without rewrite rules, the $\lambda\Pi$-calculus can express proofs of minimal many-sorted predicate logic, that is the fragment of constructive many-sorted predicate logic where the only connective is the implication $\Rightarrow$ and the only quantifiers are the universal quantifiers $\forall_s$, one for each sort.

▶ **Definition 9** (Language embedding). To each language $\mathcal{L}$ of predicate logic, we associate a context Σ containing

- for each sort s of $\mathcal{L}$, a variable `s`, of type `Type`,
- for each function symbol f of arity $\langle s_1, \ldots, s_n, s' \rangle$, a variable `f`, of type `s1 -> ... -> sn -> s'`,
- for each predicate symbol P of arity $\langle s_1, \ldots, s_n \rangle$, a variable `P`, of type `s1 -> ... -> sn -> Type`.

For instance

```
s : Type.
P : s -> Type.
```

▶ **Definition 10** (Term embedding). Let t be a term in the language $\mathcal{L}$, we express the term t as a λ-term as

- $|x| = \texttt{x}$,
- $|f(t_1, \ldots, t_n)| = (\texttt{f}\ |t_1|\ \ldots\ |t_n|)$.

The following lemma is proved with a simple induction on t.

▶ **Lemma 11.** *If t is a term of sort s in the language $\mathcal{L}$ and Γ a context containing for each variable x of sort s' free in t, a variable* `x`*, of type* `s'`*, then* $\Sigma, \Gamma \vdash |t| : \texttt{s}$.

▶ **Definition 12** (Proposition embedding). Let A be a proposition in the language $\mathcal{L}$, we express the proposition A as the following λ-term

- $\|P(t_1, \ldots, t_n)\| = (\texttt{P}\ |t_1|\ \ldots\ |t_n|)$,
- $\|A \Rightarrow B\| = \|A\|$ `->` $\|B\|$,
- $\|\forall_s x\ A\| =$ `x:s ->` $\|A\|$.

The following lemma is proved with a simple induction on A.

▶ **Lemma 13.** *If A is a proposition in the language $\mathcal{L}$ and Γ a context containing for each variable x of sort s free in A, a variable* `x`*, of type* `s`*, then* $\Sigma, \Gamma \vdash \|A\| :$ `Type`.

The following theorem is proved with a simple induction on proof structure and on the structure of the normal form of π.

▶ **Theorem 14** (Proof embedding). *Let $\mathcal{L}$ be a language and $A_1, \ldots, A_n \vdash B$ be a sequent in $\mathcal{L}$, Γ_1 a context containing for each variable x of sort s free in $A_1, \ldots, A_n \vdash B$, a variable* `x`*, of type* `s`*, and Γ_2 be a context containing, for each hypothesis A_i, a variable* $\texttt{a}_\texttt{i}$*, of type $\|A_i\|$. Then, the sequent $A_1, \ldots, A_n \vdash B$ has a proof in Natural deduction, if and only if there exists a λ-term π such that $\Sigma, \Gamma_1, \Gamma_2 \vdash \pi : \|B\|$, if and only if there exists a normal λ-term π such that $\Sigma, \Gamma_1, \Gamma_2 \vdash \pi : \|B\|$.*

For instance, the proposition $\forall_T x\ (P(x) \Rightarrow P(x))$ is embedded as `x:T -> (P x) -> (P x)` and it has the proof `x:T => a:(P x) => a`.

```
def Thm : x:s -> (P x) -> (P x) := x:s => a:(P x) => a.
```

Note that if B is a proposition, then the term $\|B\|$ is a type, in the sense that it has type `Type` and that it is the type of the proofs of B. But not all terms of type `Type` have the form $\|B\|$, for instance if s is a sort of the language, then `s` has type `Type` but there exists no proposition B such that `s` $= \|B\|$. So the concepts of proposition and type are not identified: propositions are types, but not all types are propositions.

4.2 Constructive predicate logic

The λΠ-calculus has been designed to express functional types but not Cartesian product types, disjoint union types, unit types, or empty types. Thus, the connectives and quantifiers $\top$, $\bot$, $\neg$, $\wedge$, $\vee$, and $\exists$ cannot be directly expressed as $\Rightarrow$ and $\forall$ in Section 4.1. One possibility is to extend the λΠ-calculus with Cartesian product types or inductive types. Another is

to use the rewrite rules of the $\lambda\Pi$-calculus modulo theory to define these connectives and quantifiers.

We first modify the expression of minimal predicate logic presented in Section 4.1. We introduce a variable `o`, of type `Type`

```
o : Type.
```

and instead of expressing a predicate symbol as a term of type $\texttt{s}_1\texttt{-> ... -> s}_n\texttt{ -> Type}$, we now express it as a term of type $\texttt{s}_1\texttt{ -> ... -> s}_n\texttt{ -> o}$

```
s : Type.
P : s -> o.
```

Thus, the atomic proposition $P(c)$ is expressed, in a first step, as the term `(P c)`, of type `o`. We can also introduce a variable `imp` of type `o -> o -> o`

```
imp : o -> o -> o.
```

in order to express the proposition $P(c) \Rightarrow P(c)$ as the term `(imp (P c) (P c))`, of type `o`. Then, to make a connection to Section 4.1, we introduce a variable `eps`—read "epsilon"—of type `o -> Type`

```
def eps : o -> Type.
```

and the proposition $P(c)$ can be expressed as the term `eps (P c)`, of type `Type`. The type o is thus a universe "*à la* Tarski" [54]. The proposition $P(c) \Rightarrow P(c)$ can then be expressed both as `eps (imp (P c) (P c))` and as `eps (P c)) -> (eps (P c)` and we add the rewrite rule

```
[x, y] eps (imp x y) --> (eps x) -> (eps y).
```

to identify these expressions.

This allows also to add other connectives and quantifiers, for instance the conjunction `and` as a variable of type `o -> o -> o`, together with the rewrite rule that express its meaning through the so called "second-order" expression of the conjunction

```
and: o -> o -> o.
[x, y] eps (and x y) --> z:o -> (eps x -> eps y -> eps z) -> eps z.
```

▶ **Definition 15** (Language embedding). To each language $\mathcal{L}$ of predicate logic, we associate a context Σ containing

- for each sort s of $\mathcal{L}$ a variable `s` of type `Type`,
- a variable `o` of type `Type`,
- for each function symbol f of arity $\langle s_1, \ldots, s_n, s' \rangle$, a variable `f`, of type $\texttt{s}_1\texttt{-> ... -> s}_n\texttt{-> s'}$,
- for each predicate symbol P of arity $\langle s_1, \ldots, s_n \rangle$, a variable `P`, of type $\texttt{s}_1\texttt{-> ... -> s}_n\texttt{-> o}$,
- variables `top` and `bot` of type `o`,
- a variable `not` of type `o -> o`,
- variables `imp`, `and`, and `or` of type `o -> o -> o`,
- for each sort s of $\mathcal{L}$ a variable `fa_s` and a variable `ex_s` of type `(s -> o) -> o`,
- a variable `eps` of type `o -> Type`.

The embedding of terms is defined as in Definition 10 and that of propositions comes in two steps. First, the translation |.| associates a term of type `o` to each proposition. Then the symbol `eps` is applied to this term to obtain a term of type `Type`.

▶ **Definition 16** (Proposition embedding)**.** Let A be a proposition in the language $\mathcal{L}$. We express the proposition A as a λ-term as

- $|P(t_1, \ldots, t_n)| = (\texttt{P}\ |t_1|\ \ldots\ |t_n|)$,
- $|\top| = \texttt{top}$, $|\bot| = \texttt{bot}$,
- $|\neg A| = \texttt{not}\ |A|$,
- $|A \Rightarrow B| = \texttt{imp}\ |A|\ |B|$, $|A \wedge B| = \texttt{and}\ |A|\ |B|$, $|A \vee B| = \texttt{or}\ |A|\ |B|$,
- $|\forall_s x\ A| = \texttt{fa\_s (x:s =>}\ |A|\texttt{)}$, $|\exists_s x\ A| = \texttt{ex\_s (x:s =>}\ |A|\texttt{)}$,

and finally

- $\|A\| = \texttt{eps}\ |A|$.

The following lemma is proved with a simple induction on A.

▶ **Lemma 17.** *If A is a proposition in the language $\mathcal{L}$ and Γ a context containing for each variable x of sort s free in A, a variable* `x`*, of type* `s`*, then $\Sigma, \Gamma \vdash |A| : \texttt{o}$ and $\Sigma, \Gamma \vdash \|A\| : \texttt{Type}$.*

Finally, we add the rewrite rules expressing the meaning of the connectives—`and` is presented above—and quantifiers. The rationale behind those rules is, that they turn the deep `o`-level encoding into a shallow `Type`-level encoding.

```
def top : o.
[ ] eps top --> z:o -> (eps z) -> (eps z).

def bot : o.
[ ] eps bot --> z:o -> (eps z).

def not : o -> o := x:o => imp x bot.

def or : o -> o -> o.
[x, y] eps (or x y)
        --> z:o -> (eps x -> eps z) -> (eps y -> eps z) -> eps z.

def fa_s : (s -> o) -> o.
[y] eps (fa_s y) --> x:s -> eps (y x).

def ex_s : (s -> o) -> o.
[y] eps (ex_s y) --> z:o -> (x:s -> eps (y x) -> (eps z)) -> eps z.
```

▶ **Theorem 18** (Embedding of proofs, [35])**.** *Let $\mathcal{L}$ be a language, Σ be the associated context, V be a set of variables and $A_1, \ldots, A_n, B$ be propositions in $\mathcal{L}$ such that the free variables of $A_1, \ldots, A_n, B$ are in V. Let Γ_1 be a context containing for each variable x of sort s in V, a variable* `x`*, of type* `s`*, and Γ_2 be a context containing, for each hypothesis A_i a variable* $\texttt{a}_\texttt{i}$ *of type $\|A_i\|$.*

Then, the sequent $A_1, \ldots, A_n \vdash B$ has a proof in Natural deduction if and only if there exists a λ-term π such that $\Sigma, \Gamma_1, \Gamma_2 \vdash \pi : \|B\|$.

Again, not all terms of type `Type` have the form $\|B\|$, only those convertible to a term of the form `(eps p)`, where `p` is a term of type `o`, do. For instance `o` has type `Type`, but there is no proposition B such that $\texttt{o} = \|B\|$. In contrast, every normal term of type `o` is equal to $|B|$ for some B. Thus, the type `o`, also called `Prop` or `bool` in many systems, is the type of propositions.

So, the concepts of proposition and type are not identified in this encoding: the type `o` of propositions is a subtype of the type of all types, `Type`, through the explicit embedding `eps`, of type `o -> Type`. This way `o` can have type `Type`, without introducing inconsistencies.

4.3 Deduction modulo theory

The embedding presented in Section 4.2 extends readily to Deduction modulo theory: a theory is just expressed by declaring its rewrite rules.

Moreover, some theories, such as arithmetic or set theory, contain a sort ι and a sort κ for the classes of elements of sort ι and a comprehension scheme

$$\forall x_1 \ \ldots \ \forall x_n \exists c \ \forall y \ (y \in c \Leftrightarrow A)$$

or its skolemized form

$$\forall x_1 \ \ldots \ \forall x_n \ \forall y \ (y \in f_{x_1,\ldots,x_n,A}(x_1,\ldots,x_n) \Leftrightarrow A)$$

In the $\lambda\Pi$-calculus modulo theory this sort can be omitted and replaced by the type `i -> o`.

For instance, Heyting arithmetic [41, 4] can be presented in Deduction modulo theory with nine rewrite rules defining addition, multiplication, equality, and numberness

```
nat : Type.

0 : nat.
S : nat -> nat.
def plus : nat -> nat -> nat.
def times : nat -> nat -> nat.
def equal : nat -> nat -> o.
N : nat -> o.
[y] plus 0 y --> y
[x, y] plus (S x) y --> S (plus x y).
[y] times 0 y --> 0
[x, y] times (S x) y --> plus (times x y) y.
[ ] equal 0 0 --> top
[x] equal (S x) 0 --> bot
[y] equal 0 (S y) --> bot
[x, y] equal (S x) (S y) --> equal x y.
[n] eps (N n) -->
  k:(nat->o) -> eps (k 0) ->
  eps ( fa_nat (y:nat => imp (N y) (imp (k y) (k (S y))) ) ) ->
  eps (k n).
```

The last rewrite rule, accounting for numberness, expresses that any `n`, of type `nat`, is a natural number if it verifies all predicates `k`, verified by 0 and closed by successor. Then we can prove the proposition $\forall x \ N(x) \Rightarrow (x + 0) = x$

```
def tt : eps top := z:o => p:eps z => p.

def k := x:nat => equal (plus x 0) x.

def z_r_neutral : eps ( fa_nat ( x => imp (N x) (k x)))
                 := x:nat => p:eps (N x) =>
                    p k tt (y:nat => q: eps (N y) => r:eps (k y) => r).
```

4.4 iProverModulo

In this section we present the Dedukti back-end of iProverModulo, more information can be found about iProverModulo in [21] and about the back-end in [22].

iProver [51] is a proof-search system for classical predicate logic, based on ordered resolution with selection [12]. An optional Instantiation-Generation rule [43] can also be used, but we shall use a version of iProver where this option has not been activated.

As usual, a literal is either an atomic proposition or the negation of an atomic proposition and a clause is a set of literals. If $C = \{L_1, \ldots, L_n\}$, we write $\ulcorner C \urcorner$ for the proposition $\forall x_1 \ \ldots \ \forall x_m \ (L_1 \vee (\cdots \vee (L_{n-1} \vee L_n) \ldots))$ where $x_1, \ldots, x_m$ are the variables of C. To prove a proposition A, its negation $\neg A$ is first transformed into a set of clauses $\{C_1, ..., C_n\}$ by an external tool. The proof then amounts to deriving the empty clause, using the usual resolution and factoring rules. The only difference between standard Resolution and ordered resolution with selection is that, in this second method, these rules are restricted to reduce the search space, while preserving completeness.

iProver has been extended to Deduction modulo theory, leading to iProverModulo [21]. Proposition rewrite rules are simulated using so-called one-way clauses [37] that are handled with just more restrictions on the Resolution rule. Term rewrite rules, in contrast, lead to introduce an additional Narrowing rule.

When the proposition $\neg A$ is transformed into a set of clauses $C_1, ..., C_n$ and the empty clause can be derived from $C_1, ..., C_n$ with the Resolution, Factoring, and Narrowing rules, a proof in Deduction modulo theory of the sequent $\ulcorner C_1 \urcorner, ..., \ulcorner C_n \urcorner \vdash \bot$ can be built, and this proof can then be transformed into a proof of the sequent $\vdash A$. The latter transformation is only possible in classical logic, but the proof of the sequent $\ulcorner C_1 \urcorner, ..., \ulcorner C_n \urcorner \vdash \bot$ itself is constructive. And indeed, since version 0.7-0.2, iProverModulo has an export functionality that builds a Dedukti proof of $\ulcorner C_1 \urcorner, ..., \ulcorner C_n \urcorner \vdash \bot$, every time it derives the empty clause from $C_1, ..., C_n$.

A clause C can be transformed into a proposition $\ulcorner C \urcorner$ and then to a Dedukti term $\|\ulcorner C \urcorner\|$, of type `Type`, as seen in Section 4.2. However, instead of using the binary disjunction $\vee$, we generalize it to a multiary disjunction and translate directly the clause $C = \{L_1, ..., L_n\}$ as the Dedukti term $[\![C]\!]$ of, type `Type`, as

$$[\![C]\!] = \texttt{x}_1\texttt{:i->...->x}_\texttt{m}\texttt{:i->(}\|L_1\|\texttt{->(eps bot))->...->(}\|L_n\|\texttt{->(eps bot))->(eps bot)}$$

where we use the sort `i` for terms. In particular, the translation of the empty clause is $[\![\varnothing]\!] = \texttt{eps bot} = \|\bot\|$.

Proofs built by iProverModulo are then expressed in Dedukti step by step. We first declare a variable $\texttt{d}_C$, of type $\|\ulcorner C \urcorner\|$ for each clause C in the set $C_1, ..., C_n$.

$$\texttt{d}_C\texttt{:eps} \ |\forall x_1 \ \ldots \forall x_m \ (L_1 \vee \cdots \vee (L_{n-1} \vee L_n) \ldots)|$$

Then, using these variables, we can define a term $\texttt{c}_C$, of type $[\![C]\!]$. For each new clause generated by the inference rules, we build a proof of the conclusion from the proof of the premises. For instance, for the Resolution rule

$$\frac{C \cup \{P\} \qquad D \cup \{\neg Q\}}{\sigma(C \cup D)}$$

where σ is the most general unifier of P and Q, we have a term `c` of type $[\![C \cup \{P\}]\!]$, and a term `d` of type $[\![D \cup \{\neg Q\}]\!]$. We can combine them to define a term of the resolvant type $[\![\sigma(C \cup D)]\!]$. See [22] for more details. As an example, the proof

$$\dfrac{(1) \qquad \dfrac{\dfrac{\{P(a,x), P(y,b), P(y,x)\}}{P(a,b)}\,\text{Factoring} \quad (1)}{\neg P(a,x)}\,\text{Resolution} \qquad \{\neg P(a,x), \neg P(y,b)\}}{\varnothing}\,\text{Resolution}$$

is a derivation of the empty clause from the clauses $C_1 = \{P(a,x), P(y,b), P(y,x)\}$ and $C_2 = \{\neg P(a,x), \neg P(y,b)\}$. It is translated in Dedukti as

```
i : Type.

P: i -> i -> o.
a : i.
b : i.
d1: eps (fa_i (x => fa_i (y => or (P a x) (or (P y b) (P y x))))).
d2: eps (fa_i (x => fa_i (y => or (not (P a x)) (not (P y b))))).
def c1 : x : i -> y : i -> (eps (P a x) -> eps bot) ->
                           (eps (P y b) -> eps bot) ->
                           (eps (P y x) -> eps bot) -> eps bot
       := x => y => l1 => l2 => l3 => z =>
          d1 x y z (l1' => l1 l1' z)
             (sb1 => sb1 z (l2' => l2 l2' z) (l3' => l3 l3' z)).
def c2 : x : i -> y : i ->
          ((eps (P a x) -> eps bot) -> eps bot) ->
          ((eps (P y b) -> eps bot) -> eps bot) -> eps bot
       := x => y => l1 => l2 => z =>
          d2 x y z (l1' => l1 l1' z) (l2' => l2 l2' z).
def c3 : (eps (P a b) -> eps bot) -> eps bot
       := l1 => c1 b a l1 l1 l1.
def c4 : x : i -> ((eps (P a x) -> eps bot) -> eps bot) -> eps bot
       := x => l1 => c3 (tp => c2 x a l1 (tnp => tnp tp)).
def c5 : eps bot := c3 (tp => c4 b (tnp => tnp tp)).
```

where `d1` (resp. `d2`) is the declaration of $\|\ulcorner C_1 \urcorner\|$ (resp. $\|\ulcorner C_2 \urcorner\|$); `c1` (resp. `c2`) its translation using a multiary disjunction $[\![C_1]\!]$ (resp. $[\![C_2]\!]$); and `c3`, `c4` and `c5` correspond to the clauses derived from C_1 and C_2 in the proof tree above.

iProverModulo can be used to produce Dedukti proofs for 3383 problems of the TPTP problem library v6.3.0 [66]. The generated Dedukti files weight a total of 38.1 MB, when gzipped.

5 Classical predicate logic

A logical framework should be able to express both constructive and classical theories. A way to mix constructive and classical arguments is to apply the adjectives "constructive" and "classical" not to proofs, but to connectives. So, we shall not say that the proposition $P \vee \neg P$ has a classical proof and no constructive one. Instead, we shall say that the proposition $P \vee_c \neg_c P$ has a proof and the proposition $P \vee \neg P$ does not, distinguishing the classical connectives $\vee_c$, $\neg_c$, etc. from the constructive ones $\vee$, $\neg$, etc.

Reaping the benefit of the work done on negative translations by Kolmogorov, Gödel, Gentzen, Kuroda, and others, we want to define the classical connectives from the constructive ones, adding double negations before or after the connectives. For instance, defining $A \vee_c B$ as $\neg\neg(A \vee B)$. This approach has been investigated by Hermant, Allali, Dowek, Prawitz, etc. [5, 38, 62]. In all these works the main problem is that the translation of an atomic proposition P needs to be $\neg\neg P$ and in this case there is no connective to which these negations can be incorporated. This has lead to various solutions: considering two different provability predicates [5], considering two entailment relations [38], incorporating the double negation to the predicate symbol at the root of P [62].

We consider here another solution that fits better our goal to express classical proofs in a logical framework: the introduction of a new connective.

5.1 Classical connectives and quantifiers

The usual syntax for predicate logic distinguishes terms and propositions

$$\begin{array}{rcl} t & ::= & x \mid f(t,\ldots,t) \\ A & = & P(t_1,\ldots,t_n) \mid \top \mid \bot \mid \neg A \mid A \Rightarrow A \mid A \wedge A \mid A \vee A \mid \forall x\ A \mid \exists x\ A \end{array}$$

The syntactic category of terms introduces only function symbols which are specific to the theory, while the second syntactic category of propositions introduces both predicate symbols, which are specific to the theory, and connectives which are not.

A clearer presentation completely separates the symbols of the theory and those of the logic introducing a third syntactic category for atoms

$$\begin{array}{rcl} t & ::= & x \mid f(t,\ldots,t) \\ a & ::= & P(t_1,\ldots,t_n) \\ A & = & a \mid \top \mid \bot \mid \neg A \mid A \Rightarrow A \mid A \wedge A \mid A \vee A \mid \forall x\ A \mid \exists x\ A \end{array}$$

When formalizing such a definition in a logical framework we must introduce a type for terms of each sort, a type for atoms, and a type for propositions, and we cannot just say that atoms are propositions. Instead, we must introduce an explicit embedding from atoms to propositions, a new connective, $\triangleright$, so the propositional category becomes

$$A \quad ::= \quad \triangleright a \mid \top \mid \bot \mid \neg A \mid A \Rightarrow A \mid A \wedge A \mid A \vee A \mid \forall x\ A \mid \exists x\ A$$

This way, the new connective $\triangleright$, in its classical version, may incorporate the double negation usually put on the atom itself. For instance, Kolmogorov translation yields the following definitions of the classical atomic, conjunction, and disjunction connectives

$$\begin{array}{rcl} \triangleright_{\mathrm{c}} A & = & \neg\neg \triangleright A \\ A \wedge_{\mathrm{c}} B & = & \neg\neg(A \wedge B) \\ A \vee_{\mathrm{c}} B & = & \neg\neg(A \vee B) \end{array}$$

Assuming the constructive connectives and quantifiers of Section 4.2, the full set of classical connectives and quantifiers can now be defined in Dedukti as

```
alpha : Type.
def atom : alpha -> o.
def atom_c : alpha -> o := p: alpha => not (not (atom p)).
def top_c : o := not (not top).
def bot_c : o := not (not bot).
def not_c : o -> o := x:o => (not (not (not x))).
def and_c : o -> o -> o := x:o => y:o => not (not (and x y)).
def or_c  : o -> o -> o := x:o => y:o => not (not (or x y)).
def imp_c : o -> o -> o := x:o => y:o => not (not (imp x y)).
def fa_s_c : (s -> o) -> o := x:(s -> o) => not (not (fa_s x)).
def ex_s_c : (s -> o) -> o := x:(s -> o) => not (not (ex_s x)).
```

The embedding of languages is as in Definitions 9 and 15, except that the predicate symbol now have a target type `alpha`, instead of `Type`, or `o`. The embedding of terms is defined as in Definition 10. We define the constructive and classical embeddings for propositions as follows.

▶ **Definition 19** (Proposition embedding)**.** Let A be a proposition in the language $\mathcal{L}$ we embed it constructively as a λ-term as

- $|P(t_1, \ldots, t_n)| =$ `atom (P` $|t_1|$ $\ldots$ $|t_n|$`)`,
- $|\top| =$ `top`, $|\bot| =$ `bot`,
- $|\neg A| =$ `not` $|A|$,
- $|A \Rightarrow B| =$ `imp` $|A|$ $|B|$, $|A \wedge B| =$ `and` $|A|$ $|B|$, $|A \vee B| =$ `or` $|A|$ $|B|$,
- $|\forall_s x\ A| =$ `fa_s (x:s =>` $|A|$`)`, $|\exists_s x\ A| =$ `ex_s (x:s =>` $|A|$`)`.

We embed A classically as a λ-term as

- $|P(t_1, \ldots, t_n)|_c =$ `atom_c (P` $|t_1|$ $\ldots$ $|t_n|$`)`,
- $|\top|_c =$ `top_c`, $|\bot|_c =$ `bot_c`,
- $|\neg A|_c =$ `not_c` $|A|_c$,
- $|A \Rightarrow B|_c =$ `imp_c` $|A|_c$ $|B|_c$, $|A \wedge B|_c =$ `and_c` $|A|_c$ $|B|_c$, $|A \vee B|_c =$ `or_c` $|A|_c$ $|B|_c$,
- $|\forall_s x\ A|_c =$ `fa_s_c (x:s =>` $|A|_c$ `)`, $|\exists_s x\ A|_c =$ `ex_s_c (x:s =>` $|A|_c$ `)`.

The following lemma is proved with a simple induction on A.

▶ **Lemma 20.** *If A is a proposition in the language $\mathcal{L}$ and Γ a context containing for each variable x of sort s free in A, a variable* `x`*, of type* `s`*, then $\Sigma, \Gamma \vdash |A| :$ `o` and $\Sigma, \Gamma \vdash |A|_c :$ `o`.*

Note that the normal form of $|A|_c$ is the result of the Kolmogorov translation of A, constructively embedded with $|\ |$. Thus, combining Kolmogorov's results about his translation with Theorem 18, we get the following theorem.

▶ **Theorem 21.** *A proposition A has a proof in constructive logic if and only if the type* `eps` $|A|$ *is inhabited. A proposition A has a proof in classical logic if and only if the type* `eps` $|A|_c$ *is inhabited.*

For instance, the proposition

$$\triangleright_c\, P \vee_c \neg_c \triangleright_c\, P$$

has the following proof, where a redex has been introduced

```
def lem : p : alpha -> eps (or_c (atom_c p) (not_c (atom_c p))) :=
 p => h0 : (eps (or (atom_c p) (not_c (atom_c p))) -> eps bot) =>
 (h1 : eps (not (atom_c p)) =>
  h0 (z =>
      h_left =>
      h_right =>
      h_right (h => h h1)))
 (h2 : eps (atom_c p) =>
  h0 (z =>
      h_left =>
      h_right =>
      h_left h2)).
```

Note that, since the symbols $\vee$ and $\vee_c$, $\triangleright$ and $\triangleright_c$, etc. have different meanings the propositions

$$\forall x \forall y \forall z\ (\triangleright\ x \in \{y, z\} \Leftrightarrow (\triangleright\ x = y \vee \triangleright\ x = z))$$
$$\forall_c x \forall_c y \forall_c z\ (\triangleright_c\ x \in \{y, z\} \Leftrightarrow_c (\triangleright_c\ x = y \vee_c \triangleright_c\ x = z))$$

also have different meanings. In a mixed logic containing both constructive and classical connectives, we can choose to take one, or the other, or both as axioms, leading to different theories.

5.2 Classical Deduction modulo theory

Like constructive logic, classical logic can be extended with a congruence defined by rewrite rules. The translation of a rewrite rule of classical Deduction modulo theory such as

$$\triangleright_{\mathrm{c}} x \in \{y, z\} \longrightarrow (\triangleright_{\mathrm{c}} x = y) \vee_{\mathrm{c}} (\triangleright_{\mathrm{c}} x = z)$$

could be expressed directly in Dedukti, but it would introduce a critical pair because $\triangleright_{\mathrm{c}} P$ also reduces to $\neg\neg \triangleright P$. To close this critical pair, we rather add the rule

$$\triangleright x \in \{y, z\} \longrightarrow (\triangleright_{\mathrm{c}} x = y) \vee (\triangleright_{\mathrm{c}} x = z)$$

that subsumes the first one as $\neg\neg \triangleright x \in \{y, z\}$ then rewrites to $\neg\neg((\triangleright_{\mathrm{c}} x = y) \vee (\triangleright_{\mathrm{c}} x = z))$.

Thus, to avoid critical pairs when translating a rewrite rule of classical Deduction modulo theory, we must remove the two head negations that would otherwise appear at the head of the left-hand and right-hand sides of the rewrite rule. Said otherwise, we turn the head connective of the left- and right-hand sides into a constructive one.

We had two versions of the pairing axiom above, one constructive and the other classical. In the same way, this rule is different from the translation of the rewrite rule of constructive Deduction modulo theory

$$\triangleright x \in \{y, z\} \longrightarrow (\triangleright x = y) \vee (\triangleright x = z)$$

5.3 Zenon

In this section we present Zenon 0.8.2 and Zenon Modulo 0.4.2, more information about Zenon can be found at [20], and about Zenon Modulo at [34, 23].

Zenon is a proof search system for classical predicate logic with equality, based on a tableaux method, that is a sequent calculus with one-sided sequents, where all propositions are on left-hand side. Zenon Modulo is an extension of Zenon that builds proofs in Deduction modulo theory. A proof of a proposition G in a context Γ is a tableaux proof of the sequent $\Gamma, \neg G \vdash$.

As we have seen, a proposition G has a classical proof, if and only if the type `(eps` $|G|_{\mathrm{c}}$`)` is inhabited in Dedukti. When proving a proposition G, Zenon Modulo produces a Dedukti proof-term, of type `(eps` $|G|_{\mathrm{c}}$`)` [27].

More precisely it first produces a Dedukti term of type `eps (not_c` $|G|_{\mathrm{c}}$`) -> eps bot.` This term can then be turned into a term of type `(eps` $|G|_{\mathrm{c}}$`)` by applying the constructive theorem $\neg\neg\neg A \Rightarrow \neg A$ twice, because $|G|_{\mathrm{c}}$ starts with a negation.

These proofs of `eps (not_c` $|G|_{\mathrm{c}}$`) -> eps bot` are generated by stating in Dedukti one lemma per tableaux rule. For instance, the tableaux rule

$$\frac{A \vee_{\mathrm{c}} B}{A \qquad B}\ \mathrm{Ror}$$

that corresponds to the sequent calculus rule

$$\frac{\Gamma, A \vdash \quad \Gamma, B \vdash}{\Gamma, A \vee_{\mathrm{c}} B \vdash}\ \vee\text{-left}$$

is reflected by the following Dedukti lemma

```
def Ror : A : o -> B : o ->
          (eps A -> eps bot) ->
```

```
          (eps B -> eps bot) ->
          eps (or_c A B) -> eps bot
   := A => B => HNA => HNB => HNNAB => HNNAB (HAB => HAB bot HNA HNB).
```

The translation of Zenon Modulo proofs to Dedukti has been tested using a benchmark of 9,994 B Method [3] proof obligations, representing 5.5 GiB of input files. The generated Dedukti files weigh a total of 595 MB, when gzipped.

Around fifty axioms of the B Method set theory were turned into rewrite rules. For instance, membership to the union between two sets is defined as follows

```
def mem : s -> s -> alpha.

def union : s -> s -> s.
[a, b, x] atom (mem x (union a b))
            --> (or (atom_c (mem x a)) (atom_c (mem x b))).
```

6 Simple type theory

6.1 Expressing simple type theory

There are two ways to express Simple type theory [29], also known as Higher-order logic, in the $\lambda\Pi$-calculus modulo theory. First, Simple type theory can be expressed in Deduction modulo theory [39]. Thus, as seen in Section 4.3, it can be expressed in the $\lambda\Pi$-calculus modulo theory. It can also be expressed as a Pure type system [44, 45], hence in the $\lambda\Pi$-calculus modulo theory [32], which leads to a similar result.

In Simple type theory, the sorts of the logic are the types of the simply-typed λ-calculus, with two basic types: ι for the objects and o for the propositions. The language contains the variables $\Rightarrow$, of type $o \to o \to o$, that represents implication and $\forall_A$, for each simple type A, that represent universal quantification. In order to avoid declaring an infinite number of symbols, we represent simple types as Dedukti terms of a type `type` and pass elements of this type, to index a single symbol $\forall$.

```
type : Type.
o : type.
i : type.
arrow : type -> type -> type.

def term : type -> Type.
imp : term o -> term o -> term o.
forall : a : type -> (term a -> term o) -> term o.
```

One should not confuse `type`, which is the type of objects representing types of the Simple type theory, with `Type`, which is the type of Dedukti types. We endow the arrow types of the `type` level, the behavior of arrow types of the `Type` level, by adding the rule

```
[a, b] term (arrow a b) --> term a -> term b.
```

▶ **Definition 22.** The terms of Simple type theory are embedded as Dedukti terms as follows

- $|x| =$ `x`,
- $|\Rightarrow| =$ `imp`,
- $|\forall_A| =$ `forall` $|A|$,

- $|M\ N| = |M|\ |N|$,
- $|\lambda x : A\ M| =$ `x: (term` $|A|$`) =>` $|M|$.

As in Section 4.2, we declare a variable `eps` that maps a proposition to the type of its proofs, and we define its meaning with the rewrite rules

```
def eps : term o -> Type.

[p, q] eps (imp p q) --> eps p -> eps q.
[a, p] eps (forall a p) --> x : term a -> eps (p x).
```

This signature—call it Σ_{STT}—defines the logic of minimal constructive Simple type theory. For any proposition A, we define $\|A\| =$ `eps` $|A|$, and for any context $\Gamma = A_1, \ldots, A_n$, we define $\|\Gamma\| = h_1 : \|A_1\|, \ldots, h_n : \|A_n\|$, where $h_1, ..., h_n$ are new variables. Then we can express all the proofs of Simple type theory.

▶ **Theorem 23** (Theorem 5.2.11 and Theorem 6.2.27 in [8]). *$\Gamma \vdash A$ is provable in Simple type theory, if and only if there is a λ-term π such that $\Sigma_{\text{STT}}, \|\Gamma\| \vdash \pi : \|A\|$.*

Moreover, the term π is a straightforward encoding of the original proof tree.

6.2 HOL Light proofs

The embedding of the previous Section 6.1 can be adapted to represent variations of Simple type theory, in particular classical ones like the system Q_0 [6], which is based on equality as a primitive connective and which is used in modern implementations of Simple type theory, like HOL Light [10].

The system HOLiDe—pronounced "holiday"—allows for the expression HOL Light proofs in Dedukti, building on the Open theory project [50]. HOLiDe supports all the features of HOL Light, including prenex polymorphism, constant definitions, and type definitions. It is able to express all of the 10MB OpenTheory standard theory library. This translation results in a Dedukti library of 21.5 MB, when gzipped. It could also be used to check in Dedukti proofs developed in other implementations of HOL, provided they can export proofs to the Open theory format.

7 Programming languages

Since rewriting is Turing-complete, we can also use the λΠ-calculus modulo theory as a programming language and embed programming languages into it. This allows to express proofs of programs in Dedukti. We target a shallow embedding in the logic, so that we can reason about them as functions. In particular we cannot speak about the number of lines of code of a program, or the number of execution steps, but we can speak about functional properties of these programs.

The embedding is defined by a rewrite system expressing the operational semantics of the source language.

In this section, we present three examples, the λ-calculus, the ς-calculus and ML and then FoCaLiZe that is a logic to reason about programs.

7.1 Lambda-calculus

For example, the untyped λ-calculus can be encoded by

- $|x| =$ `x`,

- $|t\ u| =$ `app` $|t|\ |u|$,
- $|\lambda x\ t| =$ `lam (x:term =>` $|t|$`)`.

in the signature

```
Term : Type.
lam : (Term -> Term) -> Term.
def app : Term -> Term -> Term.

[f,t] app (lam f) t --> f t.
```

Encoding programming languages in the $\lambda\Pi$-calculus modulo theory and translating their libraries is mandatory to check proofs of certified programs in Dedukti. It is especially adequate for checking partial correctness proofs—that implicitly assume termination of the program. As shallow embeddings preserve the programming language binding, typing, and reduction, we can abstract on the program reduction and only consider programs as mathematical functions.

Obviously, preserving the programming language reduction can lead to a non-terminating rewrite system, which is not effective in the sense of Section 2.3. However it is not often a problem in practice for the following reasons.

- The soundness of Dedukti, that is the fact that Dedukti accepts well-typed terms only, does not depend on termination but only on product compatibility—see Lemma 7.
- The only place where the type-checking algorithm uses reduction is the conversion rule, to check convertibility of two types; as long as we do not use types depending on non-terminating terms, Dedukti terminates.
- Even in presence of types depending on weakly-terminating terms, Dedukti's strategy—comparison of weak head normal forms—often terminates.

For instance, we can define a fixpoint operator in simply-typed λ-calculus

```
type : Type.
arrow : type -> type -> type.
def term : type -> Type.
[A,B] term (arrow A B) --> term A -> term B.

def fix : A : type -> B : type ->
          ((term A -> term B) -> term A -> term B) ->
          term A -> term B.
[A,B,F,a] fix A B F a --> F (fix A B F) a.
```

and prove properties of recursive terminating functions defined using this `fix` operator, such as the function `mod2` defined below.

```
nat : type.
def Nat := term nat.
0 : Nat.
S : Nat -> Nat.
def match_nat : Nat -> Nat -> (Nat -> Nat) -> Nat.
[n0,nS]   match_nat 0 n0 nS --> n0
[n,n0,nS] match_nat (S n) n0 nS --> nS n.

def mod2 := fix nat nat
                (m2 => n =>
                  match_nat n
```

```
                    0
                    (p => match_nat p
                       (S 0)
                       (q => m2 q))).

#CONV mod2 (S (S (S (S 0)))), 0.
```

The `#CONV` command in last line checks that the terms `mod2 (S (S (S (S 0))))` and `0` are convertible.

7.2 The ς-calculus

Abadi and Cardelli [2] have introduced a family of calculi for object-oriented programming. These calculi differ on their typing discipline and on whether they are purely functional or imperative. The simplest of them is Obj_1. It is object-based—classes are not primitive constructs but can be defined from objects—and its type system is similar to that of the simply-typed λ-calculus.

Types A are records of types $A ::= [l_i : A_i]_{i\in 1..n}$. Labels are distinct and their order is irrelevant. We translate them by first sorting the labels and then using association lists

```
label : Type.
type : Type.
typenil : type.
typecons : label -> type -> type -> type.
```

$|[l_i : A_i]_{i\in 1...n, l_1<...<l_n}| = \texttt{typecons}\ l_1\ |A_1|\ (\ldots \texttt{typecons}\ l_n\ |A_n|\ \texttt{typenil}\ldots)$

Objects are records of methods introduced by a special binder ς, binding the object itself inside the method's body. The primitive operations are method selection and method update:

$$\begin{array}{rcll} a, b, \ldots & ::= & x & \textit{(variable)} \\ & | & [l_i = \varsigma(x_i : A)a_i]_{i\in 1...n} & \textit{(object)} \\ & | & a.l & \textit{(method selection)} \\ & | & a.l \Leftarrow \varsigma(x : A)b & \textit{(method update)} \end{array}$$

We cannot define objects as lists in Dedukti, because a sublist of a well-typed object is not a well-typed object: the type annotation on each ς binder refers to the type of the full object. In order to define partially constructed objects as lists, we introduce the notion of preobject.

```
Preobj : type -> type -> Type.
```

The first type argument of the type `Preobj` is the type of the object that we are constructing. The second argument is the type of the part we have already constructed, a dependent type of lists parameterized by the corresponding portion of the type. When this second part reaches the first one, we get an object, so the type for objects is

```
def Obj (A : type) := Preobj A A.
```

The constructors of preobjects are

```
prenil : A : type -> Preobj A typenil.
precons : A : type ->
          B : type ->
          l : label ->
```

```
          C : type ->
          (Obj A -> Obj C) ->
          Preobj A B -> Preobj A (typecons l C B).
```

The two primitives on objects, selection and update, are the only functions that recurse on the list structure of preobjects. The update function on preobjects `preupdate` returns a preobject of the same type whereas the selection function on preobjects `preselect` returns a function defined on `Obj A`:

```
mem : label -> type -> type -> Type.
athead : l : label -> A : type -> B : type -> mem l A (typecons l A B).
intail : l : label -> A : type -> l' : label -> A' : type
              -> B : type -> mem l A B -> mem l A (typecons l' A' B).

def preselect : A : type ->
                B : type ->
                l : label ->
                C : type ->
                mem l C B ->
                Preobj A B ->
                (Obj A -> Obj C).
def preupdate : A : type ->
                B : type ->
                l : label ->
                C : type ->
                mem l C B ->
                Preobj A B ->
                (Obj A -> Obj C) ->
                Preobj A B.
```

Both functions are defined by structural induction on their argument of type `mem l C B` where `mem` is the inductive relation representing membership of the couple $(\texttt{l}, \texttt{C})$ to a type `B`.

From these, selection and update can be defined on objects:

```
def select (A : type) (l : label) (C : type)
           (p : mem l C A) (a : Obj A) : Obj C :=
   preselect A A l C p a a.
def update (A : type) := preupdate A A.
```

▶ **Definition 24** (Expression of the ς-calculus)**.** The encoding of ς-terms is then defined by

- $|x| = \texttt{x}$,
- $|[l_i = \varsigma(x : A)a_i]_{i\in 1\ldots n, l_1<\ldots<l_n}| = \texttt{precons}\ |A|\ |[l_i : A_i]_{i\in 2\ldots n}|\ \texttt{l}_1\ |A_1|\ \texttt{(x : Obj}\ |A|\ \texttt{=>}\ |a_1|\texttt{)}$ $\texttt{(... precons}\ |A|\ |[\ ]|\ \texttt{l}_\texttt{n}\ |A_n|\ \texttt{(x : Obj}\ |A|\ \texttt{=>}\ |a_n|\ \texttt{) (prenil}\ |A|\ \texttt{)...)}$,
- $|a.l_i| = \texttt{select}\ |A|\ \texttt{l}_\texttt{i}\ |A_i|\ \texttt{p}_\texttt{i}\ |a|$,
- $|a.l_i \Leftarrow \varsigma(x : A)c| = \texttt{update}\ |A|\ \texttt{l}_\texttt{i}\ |A_i|\ \texttt{p}_\texttt{i}\ |a|\ \texttt{(x : Obj}\ |A|\ \texttt{=>}\ |c|\texttt{)}$.

where $\texttt{p}_\texttt{i}$ is a proof that A contains $l_i : A_i$.

▶ **Lemma 25** (Shallow encoding, Theorems 13 and 18 in [26])**.** *This encoding preserves binding, typing, and the operational semantics of the simply-typed ς-calculus.*

This translation of the simply-typed ς-calculus in the $\lambda\Pi$-calculus modulo theories can be extended to subtyping, see [26].

7.3 ML

The translation of ML types has already been addressed in Section 6.1. The main features added by the ML programming language are call by value, recursion and algebraic datatypes. All these features are present in the λΠ-calculus modulo theory, but they are only available at toplevel and the notion of pattern matching in rewrite systems is slightly different. While, in functional languages, only values are matched and the order in which patterns are given is usually relevant, rewrite rules can be triggered on open terms and rewrite systems are assumed confluent.

Making use of tuples if necessary, we assume that every constructor has arity one. Pattern matching can be faithfully encoded without appealing to complex compilation techniques thanks to *destructors*. Destructors generalize over the if-then-else construct (the destructor associated with the constructor `true`) by binding the subterm t. Unlike the eliminators of Section 8.2, destructors are specific to a given constructor, which allows to handle nested patterns.

▶ **Definition 26.** Let `C` be a datatype constructor of type τ `->` τ', the destructor associated with `C` is the symbol $\texttt{destr}_\texttt{C}$, of type `R : type -> (eps` τ' `-> (eps` τ `-> eps R) -> eps R -> eps R)`, defined by the following rewrite rules:

$\texttt{destr}_\texttt{C}$ `R (C t) f d --> f t`
$\texttt{destr}_\texttt{C}$ `R (C' t') f d --> d` (for all other constructors `C'` for type τ')

▶ **Lemma 27** (Theorem 1 in [25])**.** *ML pattern matching can be expressed by destructors in a semantics-preserving way.*

Recursion can also be encoded without generating obvious non-termination on open terms; we introduce a global symbol `@`, of type `A : type -> B: type -> ((eps A -> eps B) -> eps A -> eps B)`, defined for each constructor `C` of type τ `->` τ' by the following rewrite rule

`@` τ' `R f (C t) --> f (C t)`

This symbol freezes evaluation until the argument starts with a constructor. It is inserted in the right-hand side of all recursive definitions to freeze recursive calls.

▶ **Lemma 28.** *If a ML term t evaluates to a ML value v, then its translation $|t|$ reduces to $|v|$.*

7.4 FoCaLiZe

Our main motivation for translating programming languages to the λΠ-calculus modulo theory is to check program certificates. The FoCaLiZe system [42] is an environment for the development of certified programs; it uses ML as an implementation language, classical predicate logic as a specification language, and provides static object-oriented features to ease modularity. Moreover, it delegates most proofs to Zenon Modulo, which provides Dedukti proofs, as seen in Section 5.3. Thus, it is a good candidate for an encoding to Dedukti.

Let us consider again the function `mod2`. It can be defined in FoCaLiZe by

```
type nat = | Zero | Succ (nat);;

let rec mod2 (n) =
```

```
  match n with
  | Zero -> Zero
  | Succ(m) ->
     match m with
       | Zero -> Succ(Zero)
       | Succ(k) -> mod2(k)
;;
```

A specification for `mod2` would typically include a theorem stating that `mod2` returns `Zero` on even numbers, where by "even number" we mean a number which is the double of another. This can be formulated from the function `twice` returning the double of a number:

```
let rec twice (n) =
  match n with
  | Zero -> Zero
  | Succ(m) -> Succ(Succ(twice(m)))
;;
```

Hence, the statement of our theorem would be `all n:nat, mod2(twice(n)) = Zero`. This theorem is proved by induction over natural numbers. The two inductive steps can be processed by Zenon Modulo.

```
theorem mod2_twice_Zero : mod2(twice(Zero)) = Zero
proof = by type nat definition of mod2, twice;;

theorem twice_Succ : all n : nat, twice(Succ(n)) = Succ(Succ(twice(n)))
proof = by type nat definition of twice;;

theorem mod2_SuccSucc : all n : nat, mod2(Succ(Succ(n))) = mod2(n)
proof = by type nat definition of mod2;;

theorem mod2_twice_Succ :
  all n : nat,
    mod2(twice(n)) = Zero ->
    mod2(twice(Succ(n))) = Zero
proof = by type nat property mod2_SuccSucc, twice_Succ;;
```

As we can see from this example, FoCaLiZe proofs are very succinct; they are mere a list of hints, used to compute a first-order problem, that is given to Zenon Modulo. In order to conclude the proof of the theorem, we need an induction principle. It can be axiomatized in FoCaLiZe

```
theorem nat_induction :
  all p : (nat -> bool),
    p(Zero) ->
    (all n : nat, p(n) -> p(Succ(n))) ->
    all n : nat, p(n)
proof = assumed;;
```

but this is not a first-order statement and Zenon Modulo is not able to instantiate it. However FoCaLiZe allows to write the instantiation directly in Dedukti

```
let mod2_twice_n_is_Zero (n : nat) : bool = (mod2(twice(n)) = Zero);;

theorem mod2_twice :
  all n : nat, mod2(twice(n)) = Zero
proof =
  dedukti proof
  {* nat_induction mod2_twice_n_is_Zero mod2_twice_Zero mod2_twice_Succ. *};;
```

FoCaLiZe version 0.9.2 features Dedukti code generation. More than 98% of FoCaLiZe standard library is checked in Dedukti. The size of the gzipped Dedukti translation weighs 1.89 MB.

8 The Calculus of inductive constructions with universes

8.1 Pure type systems

Pure type systems [14] are a general class of typed λ-calculi that include many important logical systems based on the propositions-as-types principle such as the Calculus of constructions. We can express functional Pure type systems in the $\lambda\Pi$-calculus modulo theory [32] using ideas similar to the expression of Simple type theory, see Section 6.1, and to the mechanisms of Tarski-style universes in Intuitionistic type theory [54]. We represent each type A of sort s as a term $|A|$, of type U_s, and each term M of type A as a term $|M|$, of type $\texttt{eps}_s\ |A|$ [32].

Consider a Pure type system specification $(\mathcal{S}, \mathcal{A}, \mathcal{R})$. For each sort $s \in \mathcal{S}$, we declare the type U_s whose inhabitants represent the types of sort s, and the dependent type $\texttt{eps}_s$ whose inhabitants represent the terms of the types of sort s. For conciseness, in this Section we use `e` instead of `eps`.

```
U_s : Type.
def e_s : U_s -> Type.
```

For each axiom $\langle s_1, s_2\rangle \in \mathcal{A}$, we declare the variable $\texttt{u}_{s_1}$ which is the term that represents the sort s_1 in s_2, and we give its interpretation as a type using a rewrite rule.

```
u_s1 : U_s2.
[ ] e_s2 u_s1 --> U_s1.
```

For each rule $\langle s_1, s_2, s_3\rangle \in \mathcal{R}$, we declare the variable π_{s_1,s_2} which is the term that represents dependent products such as $\Pi x : A\ B$ in s_3, and we give its interpretation as a type using a rewrite rule.

```
pi_s1s2 : a : U_s1 -> b : (e_s1 a -> U_s2) -> U_s3.
[a, b] e_s3 (pi_s1s2 a b) --> x : e_s1 a -> e_s2 (b x).
```

▶ **Definition 29.** The terms of the Pure type system are expressed as

- $|x| = \texttt{x}$,
- $|s| = \texttt{u\_s}$,
- $|M\ N| = |M|\ |N|$,
- $|\lambda x : A\ M| = \texttt{x : } \|A\| \texttt{ => } |M|$,
- $|\Pi x : A\ B| = \texttt{pi\_s1s2}\ |A|\ \texttt{(x: } \|A\| \texttt{ => } |B|\texttt{)}$ (where $\Gamma \vdash A : s_1$ and $\Gamma, x : A \vdash B : s_2$),
- $\|A\| = \texttt{e\_s}\ |A|$ (when $\Gamma \vdash A : s$),
- $\|s\| = \texttt{U\_s}$ (when $\nexists s'$ such that $\Gamma \vdash s : s'$),

- $\|[\,]\| = [\,]$,
- $\|\Gamma, x : A\| = \|\Gamma\|$, $\mathtt{x}{:}\|A\|$.

▶ **Lemma 30** (Preservation of computation, Proposition 1 in [32]). *If* $\Gamma \vdash M : A$ *and* $M \longrightarrow_\beta M'$, *then there exists* N' *such that* $|M| \longrightarrow_\beta N' \equiv_{\beta\Sigma} |M'|$.

▶ **Lemma 31** (Preservation of typing, Proposition 3 in [32]). *If* $\Gamma \vdash M : A$, *then* $\Sigma, \|\Gamma\| \vdash |M| : \|A\|$.

It is also conservative—that is, adequate—with respect to the original system.

▶ **Lemma 32** (Conservativity, adequacy, [9]). *If* $\Sigma, \|\Gamma\| \vdash M' : \|A\|$, *then there exists* M *such that* $\Gamma \vdash M : A$ *and* $|M| \equiv_{\beta\eta\Sigma} M'$.

Together, these results show that provability in the Pure type system is equivalent to provability in the embedding.

▶ **Theorem 33** ([9]). *There exists* M, *such that* $\Gamma \vdash M : A$ *if, and only if there exists* M, *such that* $\Sigma, \|\Gamma\| \vdash M : \|A\|$.

As an example, the Calculus of constructions can be defined as the Pure type system

$$\begin{array}{lcl} \mathcal{S} & ::= & \{Type, Kind\} \\ \mathcal{A} & ::= & \{(Type, Kind)\} \\ \mathcal{R} & ::= & \{(Type, Type, Type), (Type, Kind, Kind), \\ & & (Kind, Type, Type), (Kind, Kind, Kind)\} \end{array}$$

and it is expressed in Dedukti as

```
U_Type : Type.
U_Kind : Type.

def e_Type : U_Type -> Type.
def e_Kind : U_Kind -> Type.

u_Type : U_Kind.

pi_TypeType : a : U_Type -> b : (e_Type a -> U_Type) -> U_Type.
pi_TypeKind : a : U_Type -> b : (e_Type a -> U_Kind) -> U_Kind.
pi_KindType : a : U_Kind -> b : (e_Kind a -> U_Type) -> U_Type.
pi_KindKind : a : U_Kind -> b : (e_Kind a -> U_Kind) -> U_Kind.

[ ] e_Kind u_Type --> U_Type.

[a, b]
  e_Type (pi_TypeType a b) --> x : e_Type a -> e_Type (b x).
[a, b]
  e_Kind (pi_TypeKind a b) --> x : e_Type a -> e_Kind (b x).
[a, b]
  e_Type (pi_KindType a b) --> x : e_Kind a -> e_Type (b x).
[a, b]
  e_Kind (pi_KindKind a b) --> x : e_Kind a -> e_Kind (b x).
```

Consider the context $\Gamma = a : Type, b : Type, x : a, f : \Pi p : (a \to Type)\ (p\ x \to b)$, in which the term $f\ (\lambda y : a\ a)\ x$ is a proof of b. This proof is represented in Dedukti as

```
a : U_Type.
b : U_Type.
x : e_Type a.
f : (p : (e_Type a -> U_Type) -> e_Type (p x) -> e_Type b).

def example : e_Type b := f (y : e_Type a => a) x.
```

8.2 Inductive types

The Calculus of inductive constructions is an extension of the Calculus of constructions with inductive types.

The straightforward way to encode inductive types in the λΠ-calculus modulo theory is to use constructors and a primitive recursion operator as in Gödel's system T [46]. For example, for polymorphic lists, we declare a new variable `list`, such that `list A` is the type of lists of elements of `A` and two variables `nil` and `cons` for the constructors:

```
list : U_Type -> U_Type.
nil : A : U_Type -> e_Type (list A).
cons : A : U_Type -> e_Type A -> e_Type (list A) -> e_Type (list A).
```

We still need an elimination scheme to define functions by recursion on lists and to prove theorems by induction on lists. To this end, we declare a new variable `elim_list` for the primitive recursion operator:

```
List : U_Type -> Type.
[A] e_Type (list A) --> List A.

def elim_list : A : U_Type -> P : (e_Type (list A) -> U_Type)
  -> e_Type (P (nil A)) -> (x : e_Type A -> l2 : e_Type (list A)
  -> e_Type (P l2) -> e_Type (P (cons A x l2))) -> l : e_Type (list A)
  -> e_Type (P l).
```

The operator is parameterized by two types: the type `A` of the elements of the list we are eliminating and the return type `P`, which is the type of the object we are constructing by eliminating this list. It is a dependent type because it can depend on the list. The type of `elim_list` expresses that, if we provide a base case of type `P nil`, and a way to construct, for any element `x:A` and list `l2:(list A)` and object of type `P l2`, an object of type `P (cons A x l2)`, then we can construct an object of type `P l` for any list `l`. Finally, we express the computational behavior of the eliminator, that reduces when applied to a constructor.

```
[A, P, case_nil, case_cons]
  elim_list _ P case_nil case_cons (nil A) -->
  case_nil.
[A, P, case_nil, case_cons, x, l]
  elim_list _ P case_nil case_cons (cons A x l) -->
  case_cons x l (elim_list A P case_nil case_cons l).
```

Recursive functions and inductive proofs can then be expressed using this elimination operator. For instance, the append function is

```
def append : A : U_Type -> List A -> List A -> List A :=
  A : U_Type => l1 : List A => l2 : List A =>
  elim_list A (__ => list A) l2 (x => l3 => l3l2 => cons A x l3l2) l1.
```

We can extend this technique to other inductive types, and we can also adapt it to represent functions defined by pattern matching and guarded recursion—as opposed to ML general recursion—, like those of the Coq system [18]. Note that to formalize the full Calculus of inductive constructions, we need an infinite number of rules. As noticed in Section 2.2.4, each proof uses only a finite number of rules: those of the inductive types used in this proof.

8.3 Universes

Universes are the types of types in Intuitionistic type theory [54]. They are similar to sorts in Pure type systems. To fully encode the universes of Intuitionistic type theory, we need to take into account two more features. First, there is an infinite hierarchy of universes

$$U_0 : U_1 : U_2 : \cdots$$

Second, the hierarchy is cumulative

$$U_0 \subseteq U_1 \subseteq U_2 : \cdots$$

which is expressed by the typing rule

$$\frac{\Gamma \vdash M : U_i}{\Gamma \vdash M : U_{i+1}}$$

To deal with the infinite universe hierarchy, instead of declaring a different variable for each universe, we index the variable $U, \texttt{eps}, u, \pi$ by natural numbers.

```
nat : Type.
0 : nat.
S : nat -> nat.

U : nat -> Type.
def eps : i : nat -> U i -> Type.

u : i : nat -> U (S i).
[i] eps _ (u i) --> U i.

def pi : i : nat -> a : U i -> b : (eps i a -> U i) -> U i.
[i, a, b] eps _ (pi i a b) --> x : eps i a -> eps i (b x).
```

To express cumulativity, we cannot just identify the universe U_i with the universe U_{i+1}, because not all terms of type U_{i+1} have type U_i. In particular U_i itself does not. We therefore rely on explicit cast operators $\uparrow_i$ that lift types from a universe U_i to a higher universe U_{i+1}.

```
lift : i : nat -> U i -> U (S i).
[i, a] eps _ (lift i a) --> eps i a.
```

A side effect of introducing explicit casts is that some types can have multiple representations as terms. For example, if $\Gamma \vdash A : U_i$ and $\Gamma, x : A \vdash B : U_i$ then the type $\Pi x : A\ B$ has two different representations as a term in the universe U_{i+1}: `lift i (pi i` $|A|$ `(x =>` $|B|$`)` and `pi (S i) (lift i` $|A|$ `) (x => lift i` $|B|$`)`. This multiplicity can break the preservation of typing [7], so we add the following rewrite rule to ensure that types have a unique representation as terms:

```
[i, a, b] pi _ (lift i a) (x => lift {i} (b x)) -->
          lift i (pi i a (x => b x)).
```

These techniques can also be adapted to express the universes of the Calculus of inductive constructions, which in particular include an impredicative universe *Prop* [7].

8.4 Matita proofs

There are not many libraries of proofs expressed in the pure Calculus of constructions with universes. Indeed, the Coq library is expressed in a Calculus of constructions with universes, modules and universe polymorphism. Those two latter features have not yet been expressed in Dedukti, so the Coq library cannot be checked directly. Preliminary results in this direction are presented in [1].

A closely related library comes from Matita. It is expressed in a Calculus of constructions with universes, and proof irrelevance. Even if proof irrelevance has not yet been expressed in Dedukti, the system Krajono is able to translate all the files which come with the Matita tarball and which do not explicitly use proof irrelevance. Tests have been conducted on the `arithmetic` library of Matita, giving a successfully checked Dedukti library of 1.11 MB, when gzipped.

9 Contributions

The results presented in this paper build upon the work of many people.

The λΠ-calculus modulo theory appeared in a joint paper of Denis Cousineau and Gilles Dowek in 2007 [32], together with an expression of Pure type systems in it, and a partial correctness proof. The framework was further refined in [64] and then in [65]. In 2010, during his undergrad internship [35], Alexis Dorra showed that constructive predicate logic also could be expressed in the λΠ-calculus modulo theory.

The first implementation of Dedukti was designed by Mathieu Boespflug [17] between 2008 and 2011, during his thesis supervised by Gilles Dowek. A second implementation was designed in 2012 by Quentin Carbonneaux during his Master thesis [24], supervised by Olivier Hermant and Mathieu Boespflug. Finally, the current implementation has been designed by Ronan Saillard between 2012 and 2015 during his thesis [65], supervised by Pierre Jouvelot and Olivier Hermant.

A first expression of the Calculus of Inductive Constructions was proposed by Mathieu Boespflug and Guillaume Burel in 2011 [18]. An expression of Simple type theory and of the Calculus of Inductive Constructions was then proposed by Ali Assaf between 2012 and 2015 during his thesis [8] advised by Gilles Dowek and Guillaume Burel. This led to the translations of the proofs of HOL Light and Matita in Dedukti. A graphical front-end for the translation of the proofs of Holide has been proposed by Shuai Wang during his Master thesis in 2015 [68]. Guillaume Burel [22] showed in 2013 that iProverModulo could produce Dedukti proofs. During his thesis, started in 2013, advised by David Delahaye, Pierre Halmagrand showed that Zenon Modulo also could produce Dedukti proofs. Later Pierre Halmagrand, Frédéric Gilbert, and Raphaël Cauderlier, showed how classical proofs could be expressed in Dedukti without any axiom, introducing the classical connectives and quantifiers. During his thesis, started in 2013, advised by Catherine Dubois, Raphaël Cauderlier opened a new direction, with the expression of programs and properties of programs, in particular those expressed in FoCaLiZe.

10 Conclusion and future work

The encodings and the benchmarks discussed in this paper give an overview of the versatility of Dedukti to express a wide range of theories. They also demonstrate that this tool scales up well to very large libraries, recall the five libraries discussed in this paper:

The iProverModulo TPTP library	38.1 MB
The Zenon modulo Set Theory Library	595 MB
The Focalide library	1.89 MB
The Holide library	21.5 MB
The Matita arithmetic library	1.11 MB

Like the Dedukti system, all these translators and libraries are available on the Dedukti web site.

Future work include expressing more proofs in Dedukti, in particular proofs coming from Coq, PVS and from SMT solvers and build a larger proof library.

It also includes reverse engineering of proofs. Although HOL Light is a classical system, many proofs of the HOL Light library happen to be constructive. Many proofs in the Matita library do not require the full power of the Calculus of inductive constructions with universes and can be expressed in weaker theories.

This reverse engineering of proofs is a first step towards the possibility to use Dedukti to develop complex proofs by combining lemmas developed in several systems. A first investigation in this direction is presented in [11], but still requires to be generalized and automated.

More generally, we hope, with this project, to contribute to the shift of the general question "What is a good system to express mathematics?" to the more specific questions "Which definitions, axioms and rewrite rules are needed to prove which theorem?"

Acknowledgements

Many thanks to Jasmin Blanchette, Frédéric Blanqui, Cezary Kaliszyk, Claude Kircher, and Jean-Pierre Jouannaud for very helpful comments on a previous version of this paper.

References

1 J.-P. Jouannaud A. Assaf, G. Dowek and J. Liu. Untyped confluence in dependent type theories. 2016.

2 M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer New York, 1996.

3 J.-R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, 1996.

4 L. Allali. Algorithmic equality in Heyting arithmetic modulo. In *TYPES*, pages 1–17, 2007.

5 L. Allali and O. Hermant. Semantic A-translations and super-consistency entail classical cut elimination. In Ken McMillan, Aart Middeldorp, and Andreï Voronkov, editors, *LPAR*, volume 8312 of *LNCS ARCoSS*, pages 407–422. Springer, 2013.

6 P. B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof*. Academic Press Professional, Inc., San Diego, CA, USA, 1986.

7 A. Assaf. A calculus of constructions with explicit subtyping. In *Post-proceedings of the 20th International Conference on Types for Proofs and Programs (TYPES 2014)*, Leibniz International Proceedings in Informatics (LIPIcs), Paris, 2014. Schloss Dagstuhl.

8 A. Assaf. *A framework for defining computational higher-order logics*. PhD thesis, École polytechnique, 2015.

9 A. Assaf. Conservativity of embeddings in the lambda-Pi calculus modulo rewriting. In Thorsten Altenkirch, editor, *Proceedings of the 13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, Leibniz International Proceedings in Informatics (LIPIcs), Warsaw, 2015. Schloss Dagstuhl.

10 A. Assaf and G. Burel. Translating HOL to Dedukti. In *Proceedings of the Fourth Workshop on Proof Exchange for Theorem Proving (PxTP 2015)*, Electronic Proceedings in Theoretical Computer Science, Berlin, 2015.

11 A. Assaf and R. Cauderlier. Mixing HOL and Coq in Dedukti. In Kaliszyk, Cezary and Paskevich, Andrei, editor, Proceedings 4th Workshop on *Proof eXchange for Theorem Proving,* Berlin, Germany, August 2-3, 2015, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 89–96, Berlin, Germany, August 2015. Open Publishing Association.

12 L. Bachmair and H. Ganzinger. Resolution theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 19–99. Elsevier and MIT Press, 2001.

13 Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic lambda-cube, 1996.

14 H. Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

15 G. Barthe and M.H. Sørensen. Domain-free pure type systems. In *J. Funct. Program.*, pages 9–20. Springer, 1993.

16 F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.

17 M. Boespflug. *Conception d'un noyau de vérification de preuves pour le lambda-Pi-calcul modulo*. PhD thesis, École Polytechnique, 2011.

18 M. Boespflug and G. Burel. CoqInE: Translating the calculus of inductive constructions into the lambda-Pi-calculus modulo. In *Proof Exchange for Theorem Proving—Second International Workshop, PxTP*, page 44, 2012.

19 M. Boespflug, Q. Carbonneaux, and O. Hermant. The λΠ-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, *PxTP*, 2012.

20 R. Bonichon, D. Delahaye, and D. Doligez. Zenon: An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings*, pages 151–165, 2007.

21 G. Burel. Experimenting with deduction modulo. In Viorica Sofronie-Stokkermans and Nikolaj Bjørner, editors, *CADE*, volume 6803 of *Lecture Notes in Computer Science*, pages 162–176. Springer, 2011.

22 G. Burel. A shallow embedding of resolution and superposition proofs into the λΠ-calculus modulo. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP 2013*, volume 14 of *EPiC Series*, pages 43–57. EasyChair, 2013.

23 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, November 2015.

24 Q. Carbonneaux. Compilation jit des termes de preuve. Masters thesis, 2012.

25 R. Cauderlier and C. Dubois. ML pattern-matching, recursion, and rewriting: from FoCaLiZe to Dedukti. Submited to FSCD 2016.

26 R. Cauderlier and C. Dubois. Objects and subtyping in the λΠ-calculus modulo. In *Post-proceedings of the 20th International Conference on Types for Proofs and Programs (TYPES 2014)*, Leibniz International Proceedings in Informatics (LIPIcs), Paris, 2014. Schloss Dagstuhl.

27 R. Cauderlier and P. Halmagrand. Checking Zenon Modulo Proofs in Dedukti. In Kaliszyk, Cezary and Paskevich, Andrei, editor, Proceedings 4th Workshop on *Proof eXchange for Theorem Proving,* Berlin, Germany, August 2-3, 2015, volume 186 of *Electronic Proceedings in Theoretical Computer Science*, pages 57–73, Berlin, Germany, August 2015. Open Publishing Association.

28 A. Chlipala, L. Petersen, and R. Harper. Strict bidirectional type checking. In J. Gregory Morrisett and Manuel Fähndrich, editors, *TLDI*, pages 71–78. ACM, 2005.

29 A. Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

30 T. Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

31 T. Coquand and G. Huet. The calculus of constructions. *Information and computation*, 76(2):95–120, 1988.

32 D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In S. Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, pages 102–117. Springer-Verlag, 2007.

33 Deducteam. User manual for Dedukti. `http://dedukti.gforge.inria.fr/manual_v2_5.html`.

34 D. Delahaye, D. Doligez, F. Gilbert, P. Halmagrand, and O. Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *LNCS/ARCoSS*. Springer, 2013.

35 A. Dorra. Équivalence Curry-Howard entre le lambda-Pi-calcul et la logique intuitionniste. Undergrad research intership report, 2010.

36 G. Dowek. L'indécidabilité du filtrage du troisième ordre dans les calculs avec types dépendants ou constructeurs de types (the undecidability of third order pattern matching in calculi with dependent types or type constructors). *Comptes rendus à l'Académie des Sciences I*, 312(12):951–956, 1991. Erratum, ibid. I, 318, 1994, p. 873.

37 G. Dowek. Polarized resolution modulo. In Cristian S. Calude and Vladimiro Sassone, editors, *IFIP TCS*, volume 323 of *IFIP AICT*, pages 182–196. Springer, 2010.

38 G. Dowek. On the definition of the classical connectives and quantifiers. In *Why is this a Proof? Festschrift for Luiz Carlos Pereira.* College Publication, 2015.

39 G. Dowek, T. Hardin, and C. Kirchner. Hol-lambda-sigma: an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):21–45, 2001.

40 G. Dowek, T. Hardin, and C. Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.

41 G. Dowek and B. Werner. Arithmetic as a theory modulo. In J. Giesl, editor, *Term rewriting and applications*, pages 423–437. Springer-Verlag, 2005.

42 FoCaLiZe. `http://focalize.inria.fr`.

43 H. Ganzinger and K. Korovin. New directions in instantiation-based theorem proving. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings*, pages 55–64. IEEE Computer Society, 2003.

44 H. Geuvers. *Logics and Type systems.* PhD thesis, Nijmegen, 1993.

45 H. Geuvers. The calculus of constructions and higher order logic. In Ph. de Groote, editor, *The Curry-Howard isomorphism*, volume 8 of *Cahiers du Centre de logique*, pages 139–191. Université catholique de Louvain, 1995.

46 K. Gödel. Über eine bisher noch nicht benützte Erweiterung des finiten Standpunktes. *Dialectica*, pages 280–287, 1958.

47 R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

48 O. Hermant and F. Gilbert. Double negation translations as morphisms. Submitted to publication, 2015.

49 D. Hilbert and W. Ackermann. *Grundzüge der theoretischen Logik*. Springer-Verlag, 1928.

50 J. Hurd. The opentheory standard theory library. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.

51 K. Korovin. iProver – an instantiation-based theorem prover for first-order logic (system description). In Alessandro Armando and Peter Baumgartner, editors, *IJCAR*, volume 5195 of *Lecture Notes in Artificial Intelligence*, pages 292–298. Springer, 2008.

52 R. Loader. Higher order beta matching is undecidable. *Logic Journal of the IGPL*, 11(1):51–68, 2003.

53 C. Marché, A. Rubio, and H. Zantema. `https://www.lri.fr/~marche/tpdb/format.html`.

54 P. Martin-Löf. *Intuitionistic type theory, G. Sambin (Ed.)*. Bibliopolis, 1984.

55 D. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1:253–281, 1991.

56 D. Miller and G. Nadathur. An overview of λProlog. In *Proceedings of the 5th International Conference on Logic Prog ramming*, MIT Press, 1988.

57 T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349. IEEE Press, 1991.

58 B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin-Löf's type theory*. Oxford University Press, 1990.

59 C. Paulin-Mohring. Inductive definitions in the system coq rules and properties. In M. Bezem and J.-F. Groote, editors, *Typed Lambda Calculi and Applications*, pages 328–345. Springer-Verlag, 1993.

60 L.C. Paulson. Isabelle: the next seven hundred theorem provers. In E. Lusk and R. Overbeek, editors, *9th International Conf. on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, page 772–773. Springer, 1988.

61 H. Poincaré. *La science et l'hypothèse*. Flammarion, 1902.

62 D. Prawitz. Classical versus intuitionistic logic. In *Why is this a Proof? Festschrift for Luiz Carlos Pereira*. College Publication, 2015.

63 Saillard R. Rewriting modulo β in the λΠ-calculus modulo. In Iliano Cervesato and Kaustuv Chaudhuri, editors, *Proceedings Tenth International Workshop on Logical Frameworks and Meta Languages: Theory and Practice, LFMTP 2015, Berlin, Germany, 1 August 2015.*, volume 185 of *EPTCS*, pages 87–101, 2015.

64 R. Saillard. Towards explicit rewrite rules in the lambda-Pi calculus modulo. In *IWIL*, 2013.

65 R. Saillard. *Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, École des Mines, 2015.

66 G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

67 V. van Oostrom. Development closed critical pairs. In *Higher-Order Algebra, Logic, and Term Rewriting, Second International Workshop, HOA '95, Paderborn, Germany, September 21-22, 1995, Selected Papers*, pages 185–200, 1995.

68 S. Wang. Reverse engineering of higher order proofs. Master thesis, 2015.