Expressing Theories in the $\lambda\Pi$-Calculus Modulo Theory and in the Dedukti System

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

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

Abstract

The $\lambda\Pi$-calculus modulo theory, implemented in the Dedukti system, is a logical framework where many theories can be expressed: constructive and classical predicate logic, Simple type theory, programming languages, Pure type systems, the Calculus of inductive constructions with universes... This allows to use it to check large libraries of proofs coming from various proof-systems: Zenon, iProver, FoCaLiZe, HOL-Light, and Mattea.

1 Introduction

Defining a theory, such as arithmetic, geometry, or set theory, in predicate logic [48] just requires to chose function and predicate symbols and axioms, that express 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, $\land$, $\lor$, $\forall$... and their associated deduction rules are defined once and for all, in the framework and need not be redefined for each theory. Similarly, the notions of proof, model... 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 $T$ is a theory expressed in a language $L$ and $T'$ a theory expressed in a language $L'$, if $A$ is expressed in $L \cap L'$, $A \Rightarrow B$ is provable in $T$, and $A$ is provable in $T'$, then $B$ is provable in $T \cup T'$.

Despite these advantages, several logical systems have been defined, not as theories in predicate logic, but as independent systems: Simple type theory [28, 5], 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 [53], the Calculus of constructions [30], the Calculus of inductive constructions [58]... 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 modern logical systems.

1. Predicate logic does not allow the use of bound variables, except those bound by the
quantifiers ∃ and ∀. For instance, it is not possible to define, in predicate logic, a unary
function symbol →, 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 reasoning 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 × 2 = 4, several derivation steps need to
be used while a simple computation would have sufficed [60].

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
[55] and Isabelle [59]. Problems 1 and 2 have been solved in an extension of predicate logic,
called the Logical framework [46], 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 [39]. Combining the λΠ-calculus and Deduction modulo theory
yields the λΠ-calculus modulo theory [31], a variant of Martin-Löf’s logical framework [57],
which solves problems 1, 2, 3, and 4.

Problem 5 has been addressed in [37, 47, 61] and will be discussed in Section 5.

The expressivity of the λΠ-calculus modulo theory has already been discussed in [31],
where we have shown that all functional Pure type systems [13], 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 such as FoCALiZE, Sim-
ple 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 inter-
active theorem provers can be translated to and checked in Dedukti, our implementation of
the λΠ-calculus modulo theory. To do so, we shall translate to Dedukti proofs developed
in various systems: the automated theorem proving systems Zenon Modulo, iProver-
Modulo, the system containing a programming language as a subsystem, FoCALiZE, and
the interactive theorem provers HOL-Light and Matita.

We show this way that the logical framework approach scales up and that it is mature
enough, so that we can start imagining a single library of formal proofs expressed in different
theories, developed in different systems, but formulated in a single framework.
This paper presents a synthesis on the state of the art about the $\lambda\Pi$-calculus modulo theory, its implementation Dedukti, and the expression of theories in this framework. Most of the theorems stated in this paper have been proved in others. 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

Our presentation of the $\lambda\Pi$-calculus modulo theory comes in two steps. We shall first present, in Section 2.2, a very general system, not caring about the decidability of type-checking. Then, in Section 2.3, we shall 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 $\lambda$-calculus with dependent types. In this calculus, types are just terms of a particular type $Type$, some symbols have the type $A \rightarrow Type$, 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 $Type$, $A \rightarrow Type$... and the arrow $A \rightarrow B$ 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

\[
[\ ] \text{well-formed}
\]

Declaration of a type or type family variable

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

Declaration of an object variable

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

$Type$

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

$Variable$

\[
\frac{\Gamma \text{ well-formed}}{\Gamma \vdash x : A \quad x \in \Gamma}
\]

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)
\[
\frac{\Gamma ⊢ A : Type \quad \Gamma, x : A ⊢ B : Kind \quad \Gamma, x : A ⊢ t : B}{\Gamma ⊢ \lambda x : A \, t : \Pi x : A \, B}
\]
Abstraction (for objects)
\[
\frac{\Gamma ⊢ A : Type \quad \Gamma, x : A ⊢ B : Type \quad \Gamma, x : A ⊢ t : B}{\Gamma ⊢ \lambda x : A \, t : \Pi x : A \, B}
\]
Application
\[
\frac{\Gamma ⊢ t : \Pi x : A \, B \quad \Gamma ⊢ t' : A}{\Gamma ⊢ (t \, t') : (t'/x)B}
\]
Conversion
\[
\frac{\Gamma ⊢ t : A \quad \Gamma ⊢ A : Type \quad \Gamma ⊢ B : Type \quad A \equiv \beta B}{\Gamma ⊢ t : B}
\]
\[
\frac{\Gamma ⊢ t : A \quad \Gamma ⊢ A : Kind \quad \Gamma ⊢ B : Kind \quad A \equiv \beta B}{\Gamma ⊢ t : B}
\]
2.2.2 Conversion

If \( l \rightarrow^\Delta r \) is a rewrite rule and \( \sigma \) is a substitution binding the variables of \( \Delta \), we say that the term \( \sigma l \) rewrites to the term \( \sigma r \).

If \( \Gamma \) is a well-formed global context, then the rewriting relation generated by \( \beta \)-reduction and \( \Gamma \), noted \( \rightarrow^\beta \Gamma \), is the smallest relation, closed by context, such that if \( t \) rewrites to \( u \), for some rule in \( \Gamma \), or if \( t \) \( \beta \)-reduces to \( u \), then \( t \rightarrow^\beta \Gamma u \). The congruence \( \equiv^\beta \Gamma \) is the reflexive-symmetric-transitive closure of the relation \( \rightarrow^\beta \Gamma \).

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 \( \equiv^\beta \Gamma \).

\[\Gamma \vdash A : Type \quad \Gamma \vdash B : Type \quad \Gamma \vdash t : A \quad A \equiv^\beta \Gamma B\]

\[\Gamma \vdash A : Kind \quad \Gamma \vdash B : Kind \quad \Gamma \vdash t : A \quad A \equiv^\beta \Gamma B\]

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 holds if we have the two following properties: well-typedness of rewrite rules and product compatibility. Henceforth, unless stated otherwise, \( \Gamma \vdash t : T \) stands for “the judgement \( \Gamma \vdash t : T \) is derivable”.

▶ Definition 2. Let \( l \rightarrow^\Delta_0 r \) be a rewrite rule in the context \( \Gamma_0 \) and \( \Gamma \) be a well-formed extension of \( \Gamma_0 \). We say that \( l \rightarrow^\Delta_0 r \) is well-typed for \( \Gamma \) if for any substitution \( \sigma \) binding the variables of \( \Delta_0 \)

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

▶ Definition 3. A well-formed global context \( \Gamma \) 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 \( \Gamma \)

\[\text{if } \Pi x : A_1 B_1 \equiv^\beta \Gamma \Pi x : A_2 B_2 \text{ then } A_1 \equiv^\beta \Gamma A_2 \text{ and } B_1 \equiv^\beta \Gamma B_2.\]

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 [64]). Let \( \Gamma \) be a global context satisfying product compatibility and whose rewrite rules are well-typed. If \( \Gamma \vdash t_1 : T \) and \( t_1 \rightarrow^\beta \Gamma t_2 \) 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 [64]). Let \( \Gamma \) 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. But some others, such as arithmetic or set theory, require an infinite set of axioms, recognized by an algorithm: 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 $\Gamma$ of $\mathcal{T}$, such that the finite sequent $\Gamma \vdash A$ has a proof.

In the same way, some theories in the $\lambda\Pi$-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 $\lambda\Pi$-calculus modulo theory, the derivability of a typing relation $t : A$ must be defined as the existence of a finite context $\Gamma$, 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, $\Gamma \vdash A : s \Gamma \vdash B : s \Gamma \vdash t : A \implies A \equiv_{\beta} B$, the conversion sequence justifying $A \equiv_{\beta} B$ 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 $\equiv_{\beta}$ is decidable. A natural subsystem is obtained by restricting the rewrite rules to form, together with the $\beta$-reduction rule, a terminating and confluent rewrite system. This way, the congruence can be decided by checking equality of normal forms.

As $\beta$-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 $\beta$-reduction is not general enough. Indeed, such a confluence result may easily be lost when allowing $\lambda$-abstraction in the left-hand side of a rewrite rule, a useful feature—see, for instance, Section 8.3.

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

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

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

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

that cannot be joined. This rewrite system with $\beta$-reduction is therefore not confluent.

Still, $\equiv_{\beta}$ is decidable, as some form of confluence holds. For this, we need a richer rewrite relation: rewriting modulo $\beta$-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 $\sigma$ such that $\sigma t$ is $\beta$-convertible to $t$. Higher-order matching is
undecidable in general [35, 51], but it is decidable if we restrict to Miller’s patterns [54]. Thus, we restrict the left-hand sides of the rewrite rules to be patterns.

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

For example, if $f$ is declared in $\Delta$, 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

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

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

The reason why we are interested the confluence of this extended higher-order rewrite relation is that, when combined with $\beta$-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: $\Gamma \vdash l \rightarrow^\Delta r$ expressing that the rule $l \rightarrow^\Delta r$ is well-typed in $\Gamma$ and $\Gamma$ strongly well-formed, expressing that all the rules declared in $\Gamma$ are well-typed. This judgment is defined by the rules

- $\Gamma$ strongly well-formed
- $\Gamma \vdash A : Type$
- \[ \Gamma, x : A \text{ strongly well-formed} \]
- $\Gamma$ strongly well-formed
- $\Gamma \vdash A : Kind$
- \[ \Gamma, x : A \text{ strongly well-formed} \]
- $\Gamma$ strongly well-formed
- $\Gamma \vdash l \rightarrow^\Delta r$
- \[ \Gamma, l \rightarrow^\Delta r \text{ strongly well-formed} \]

Defining an effective rule for the judgment $\Gamma \vdash l \rightarrow^\Delta r$, 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 $\Gamma, \Delta$.

\[ l \text{ is a pattern } \Gamma, \Delta \vdash l : T \quad \Gamma, \Delta \vdash r : T \quad \text{dom}(\Delta) \subseteq \text{FV}(l) \]

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

A first possibility is to use the non left-linear rule
(Tail n (Cons n a l)) → l
but proving confluence of a non-linear rewrite system, not assuming termination—as under-
lined above, confluence comes first—, is difficult. Thus, we might prefer the linear rule
(Tail n (Cons m a l)) → 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.
Typing the term σ(Tail n (Cons m a l)) generates the constraint
(Vector (S σn)) = (Vector (S σm))
so all the substitutions σ, such that σ(Tail n (Cons m a l)) is well-typed, have the form
ητ where η is an arbitrary substitution and τ = p/m, p/n. The substitution τ is the most
general typing substitution for l. Both the terms τ(Tail n (Cons m a l)) and τl have type
(Vector p). So, for all η, ητ(Tail n (Cons m a l)) and ητl have the same type and the rule
is well-typed.

Therefore, sufficient condition for the rule l → 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
$l is a pattern \quad \Gamma, \Delta \vdash \tau l : T \quad \Gamma, \Delta \vdash \tau r : T \quad dom(\Delta) \subseteq FV(l)$
$\Gamma \vdash l \rightarrow^{\Delta} r$
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 [64]). Let Γ be a well-formed context satisfying the product
compatibility. If $\Gamma \vdash l \rightarrow^{\Delta} r$, then $(l \rightarrow^{\Delta} r)$ is well-typed in $\Gamma$.

Finally, we can prove the effectiveness theorem.

Theorem 8 (Effectiveness, Theorem 4.5.10 and Theorem 6.3.1 in [64]). 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 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 \(\lambda\Pi\)-calculus modulo theory

As in the \(\lambda\Pi\)-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 II-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 `\(\Delta\)` 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. In this case, we need to put a dot only at the end of the set of rewrite rules, as follows:

```
\[n\] 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 [52] 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, \(\lambda\)-abstraction is expressed with the double arrow `\(\Rightarrow\)`:

```
def K2 := x:nat => two.
```
Notice, that type annotation on the variable is optional [14], when it can be reconstructed by bidirectional typechecking [18, 27, 29]. 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 [32], 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

\[ \text{tail } n \ (\text{Cons } m \ a \ l) \rightarrow l \]

is to compute the most general typing substitution for its left-hand side, \(\text{tail } n \ (\text{Cons } m \ a \ l)\), and that the existence of this substitution depends on the injectivity of the symbols \(\text{vector}\) and \(S\). So, DEDUKTI needs to be aware that these symbols are injective with respect to the congruence \(\equiv_{\beta}\).

A simple way to ensure this property is to forbid rewrite rules whose head symbol is \(\text{vector}\) 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 \(\text{vector}\) in the declarations

\[
A : \text{Type}.
\]

\[
\text{vector} : \text{nat} \rightarrow \text{Type}.
\]

\[
\text{Nil} : \text{vector} \ 0.
\]

\[
\text{Cons} : n : \text{nat} \rightarrow A \rightarrow \text{vector} \ n \rightarrow \text{vector} \ (S \ n).
\]

Symbols such as \(\text{tail}\) which may appear at the head of rewrite rules are called definable and are declared using the keyword def

\[
\text{def tail} : n : \text{nat} \rightarrow \text{vector} \ (S \ n) \rightarrow \text{vector} \ n.
\]

Then, the rule

\[
[n, m, a, l] \text{tail } n \ (\text{Cons } m \ a \ l) \rightarrow l.
\]

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

\[
[n, a, l] \text{tail } n \ (\text{Cons } _ \ a \ l) \rightarrow l.
\]

or even

\[
[l] \text{tail } _ \ (\text{Cons } _ \ _ \ l) \rightarrow l.
\]
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.

\[ [\text{n, a, l}] \; \text{tail n (Cons \{n\} a l)} \rightarrow \text{l}. \]

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 not, it 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 series of examples of theories, that can be embedded in Dedukti, consist of constructive 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 \( \Sigma \) containing
- for each sort \( s \) of \( \mathcal{L} \), a variable \( s \), of type \( \text{Type} \),
- for each function symbol \( f \) of arity \( \langle s_1, \ldots, s_n, s' \rangle \), a variable \( f \), of type \( s_1 \rightarrow \ldots \rightarrow s_n \rightarrow s' \),
- for each predicate symbol \( P \) of arity \( \langle s_1, \ldots, s_n \rangle \), a variable \( P \), of type \( s_1 \rightarrow \ldots \rightarrow s_n \rightarrow \text{Type} \).

For instance

\[ s : \text{Type}. \]
\[ P : s \rightarrow \text{Type}. \]

▶ Definition 10 (Term embedding). Let \( t \) be a term in the language \( \mathcal{L} \), we express the term \( t \) as a \( \lambda \)-term as
- \( |x| = x \),
- \( |f(t_1, \ldots, t_n)| = (f |t_1| \ldots |t_n|) \).

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 \( \Gamma \) a context containing for each variable \( x \) of sort \( s' \) free in \( t \), a variable \( x \), of type \( s' \), then \( \Sigma, \Gamma \vdash |t| : s \).
Definition 12 (Proposition embedding). Let $A$ be a proposition in the language $L$, we express the proposition $A$ as a $\lambda$-term as

$$∥P(t_1, \ldots, t_n)∥ = (P \mid t_1 | \ldots | t_n),$$

$$∥A \Rightarrow B∥ = ∥A∥ \Rightarrow ∥B∥,$$

$$∥\forall x A∥ = x:s \Rightarrow ∥A∥.$$  

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

Lemma 13. If $A$ is a proposition in the language $L$ and $Γ$ a context containing for each variable $x$ of sort $s$ free in $A$, a variable $x$, of type $s$, then $Σ, Γ_1, Γ_2 \vdash ∥A∥ : Type$.

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 $L$ be a language and $A_1, \ldots, A_n \vdash B$ be a sequent in $L$, $Γ_1$ 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 $a_i$, 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 $Σ, Γ_1, Γ_2 \vdash π : ∥B∥$, if and only if there exists a normal $λ$-term $π$ such that $Σ, Γ_1, Γ_2 \vdash π : ∥B∥$.

For instance, the proposition $∀T x (P(x) ⇒ P(x))$ is embedded as $x:T \Rightarrow (P x) ⇒ (P x)$ 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 is 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 $\top, \bot, ¬, \land, \lor$, and $∃$ cannot be directly expressed as $⇒$ and $∀$ 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 $λΠ$-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 $s_1 \Rightarrow \ldots \Rightarrow s_n \Rightarrow Type$, we now express it as a term of type $s_1 \Rightarrow \ldots \Rightarrow s_n \Rightarrow o$

```
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 \((\text{imp } (P c) (P c))\), of type \( o \).

Then, to make a connection to Section 4.1, we introduce a variable \( \text{eps} \)—read “epsilon”—of type \( o \rightarrow \text{Type} \).

def eps : o -> Type.

And the proposition \( P(c) \Rightarrow P(c) \) can then be expressed both as \((\text{eps } (\text{imp } (P c) (P c)))\) and as \((\text{eps } (P c)) \rightarrow (\text{eps } (P c))\) and we add the rewrite rule

\[[x, y] \text{eps } (\text{imp } x y) --> (\text{eps } x) \rightarrow (\text{eps } y).\]

to identify these expressions.

This allows also to add other connectives and quantifiers, for instance the conjunction and as a variable of type \( o \rightarrow o \rightarrow o \), together with the rewrite rule

\[[x, y] \text{eps } (\text{and } x y) --> z:o -> (\text{eps } x \rightarrow \text{eps } y \rightarrow \text{eps } z) \rightarrow \text{eps } z.\]

\textbf{Definition 15 (Language embedding).} To each language \( \mathcal{L} \) of Predicate logic, we associate a context \( \Sigma \) containing

- for each sort \( s \) of \( \mathcal{L} \) a variable \( s \) of type \( \text{Type} \),
- a variable \( o \) of type \( \text{Type} \),
- for each function symbol \( f \) of arity \( \langle s_1, \ldots, s_n, s' \rangle \), a variable \( f : s_1 \rightarrow \cdots \rightarrow s_n \rightarrow s' \),
- for each predicate symbol \( P \) of arity \( \langle s_1, \ldots, s_n \rangle \), a variable \( P : s_1 \rightarrow \cdots \rightarrow s_n \rightarrow o \),
- variables \( \text{top} \) and \( \text{bot} \) of type \( o \),
- a variable \( \text{not} \) of type \( o \rightarrow o \),
- variables \( \text{imp} \), \( \text{and} \), and \( \text{or} \) of type \( o \rightarrow o \rightarrow o \),
- for each sort \( s \) of \( \mathcal{L} \) a variable \( \text{fa}_s \) and a variable \( \text{ex}_s \) of type \( (s \rightarrow o) \rightarrow o \),
- a variable \( \text{eps} \) of type \( o \rightarrow \text{Type} \).

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 \( \text{eps} \) is applied to this term to obtain a term of type \( \text{Type} \).

\textbf{Definition 16 (Proposition embedding).} Let \( A \) be a proposition in the language \( \mathcal{L} \), we express the proposition \( A \) as a \( \lambda \)-term as

- \(|P(t_1, \ldots, t_n)| = (P |t_1| \ldots |t_n|),\)
- \(|\top| = \text{top}, |\bot| = \text{bot},\)
- \(|\neg A| = \text{not } |A|,\)
- \(|A \Rightarrow B| = \text{imp } |A| |B|, |A \land B| = \text{and } |A| |B|, |A \lor B| = \text{or } |A| |B|,\)
- \(|\forall x A| = \text{fa}_s (x:s \Rightarrow |A|), |\exists x A| = \text{ex}_s (x:s \Rightarrow |A|),\)

and finally

\(|A| = \text{eps } |A| .\)

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

\textbf{Lemma 17.} If \( A \) is a proposition in the language \( \mathcal{L} \) and \( \Gamma \) 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 \( \Sigma, \Gamma \vdash |A| : \text{Type} \).
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, similarly to the second-order encoding of connectives and quantifiers.

```plaintext
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, [34]). Let \( \mathcal{L} \) be a language, \( \Sigma \) 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 \( \Gamma_1 \) be a context containing for each variable \( x \) of sort \( s \) in \( V \), a variable \( x \), of type \( s \), and \( \Gamma_2 \) be a context containing, for each hypothesis \( A_i \) a variable \( a_i \) 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 \( \lambda \)-term \( \pi \) such that \( \Sigma, \Gamma_1, \Gamma_2 \vdash \pi : \| B \| \).

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 \( \| B \| \), only those convertible to a term of the form \( (\text{eps } p) \), where \( p \) is a term of type o, do. For instance o has type Type, but there is no proposition \( B \) such that \( o = \| B \| \). 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: the type o of propositions is a subtype of the type of all types, Type, through the explicit embedding \( \text{eps} \), of type o \( \rightarrow \) Type. This way o can have type Type, without introducing any inconsistency.

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 \( \iota \) and a sort \( \kappa \) for the classes of elements of sort \( \iota \) and a comprehension scheme

\[ \forall x_1 \ldots \forall x_n \exists c \forall y \ (y \in c \iff A) \]

or its skolemized form

\[ \forall x_1 \ldots \forall x_n \forall y \ (y \in f_{x_1,\ldots, x_n}(x_1, \ldots, x_n) \iff A) \]
In the $\lambda\Pi$-calculus modulo theory this sort can be omitted and replaced by the type $i \to o$.

For instance, Heyting Arithmetic $[40, 3]$ can be presented in Deduction modulo theory with nine rules defining addition, multiplication, equality, and numberness

```
nat : Type.
0 : nat.
S : nat \to nat.
def plus : nat \to nat \to nat.
def times : nat \to nat \to nat.
def equal : nat \to nat \to o.
N : nat \to 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) \to eps (k 0) -->
  eps ( fa_nat (y:nat => imp (N y) (imp (k y) (k (S y)))) ) \to
  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 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 proofs

iProver $[50]$ is a proof-search system for classical predicate logic, based on Ordered resolution with selection $[12]$. An optional Instantiation-Generation rule $[42]$ 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 $\forall C$ for the proposition $\forall x_1 \ldots \forall x_m \, (L_1 \lor \cdots \lor (L_{n-1} \lor L_n) \ldots)$ 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 $\{ C_1, \ldots, C_n \}$ by an external tool. The proof then boils down 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 avoid redundancy, while preserving completeness.

iProver has been extended to Deduction modulo theory, leading to iProverModulo $[20]$. Proposition rewrite rules are simulated using so-called one-way clauses $[36]$, 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 \( C_1, \ldots, C_n \) and the empty clause can be derived from \( C_1, \ldots, C_n \) with the Resolution, Factoring, and Narrowing rules, a proof in Deduction modulo theory of the sequent \( \vdash C_1, \ldots, C_n \vdash \bot \) 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 \( \vdash C_1, \ldots, C_n \vdash \bot \) itself is constructive. And indeed, iProverModulo has an export functionality, that builds a Dedukti proof of \( \vdash C_1, \ldots, C_n \vdash \bot \), every time it derives the empty clause from \( C_1, \ldots, C_n \).

A clause \( C \) can be transformed into a proposition \( \lnot C \) and then to a Dedukti term \( \Gamma.C \), of type Type, as seen in Section 4.2. However, instead of using the binary disjunction \( \lor \), we generalize it to a multiary disjunction and translate directly the clause \( C = \{ \bot \} \) as constructive. And indeed, \( \Gamma.C \) has an export functionality, that builds a Dedukti term \( \Gamma.C \), of type Type, as

\[
\Gamma.C = (x_1 : i \to \ldots \to x_n : i \to (L_1 \to (\text{eps} \bot)) \to \ldots \to (L_n \to (\text{eps} \bot)) \to (\text{eps} \bot)
\]

where we use the sort \( i \) for terms. In particular, the translation of the empty clause is \( \emptyset = (\text{eps} \bot) \)

Proofs built by iProverModulo are then expressed in Dedukti step by step. We first declare a variable \( d_c \), of type \( \Gamma.C \) for each clause \( C \) in the set \( C_1, \ldots, C_n \).

\[
d_c : \text{eps} \ \forall x_1 \ldots \forall x_m \ (L_1 \lor \ldots \lor (L_{n-1} \lor L_n) \ldots)
\]

Then, using these variables, we can define a term \( c_c \), of type \( \Gamma.C \). 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

\[
\sigma(L_1 \lor \ldots \lor L_n \lor K_1 \lor \ldots \lor K_m)
\]

where \( \sigma \) is the most general unifier of \( P \) and \( Q \), we have a term \( c \), of type

\[
x_1 : i \to \ldots \to x_n : i \to (L_1 \to (\text{eps} \bot)) \to \ldots \to (L_1 \to (\text{eps} \bot)) \to (P \to (\text{eps} \bot)) \to (\text{eps} \bot) -> (L_n \to (\text{eps} \bot)) \to \ldots \to (L_n \to (\text{eps} \bot)) \to (\text{eps} \bot)
\]

and a term \( d \), of type

\[
y_1 : i \to \ldots \to y_n : i \to (K_1 \to (\text{eps} \bot)) \to \ldots \to (K_1 \to (\text{eps} \bot)) \to (\text{eps} \bot) -> (K_n \to (\text{eps} \bot)) \to \ldots \to (K_n \to (\text{eps} \bot)) \to (\text{eps} \bot)
\]

that we can combine to define a term of the resolvant type. See [21] for more details. For instance, the proof

\[
\frac{\{P(a, x), P(y, b), P(y, x)\}}{\text{Factoring}} \quad \frac{P(a, b) \ \ (1)}{\text{Resolution}} \quad \frac{\{\neg P(a, x), \neg P(y, b)\}}{\text{Resolution}}
\]

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

\[
i : \text{Type}.
\]

\[
P : i \to i \to o.
\]
The second example below, that proves $\exists x \ x + 1 = 2$ (d3), uses the Narrowing rule. It indeed applies c3 to S y, that triggers a rewrite rule on plus.

eq : i -> i -> o.
0 : i.
S : i -> i.
def plus : i -> i -> i.

[x] plus 0 x --> x.
[x, y] plus (S x) y --> S (plus x y).
def one : i := S 0.
def two : i := S one.
d'1 : eps(eq 0 0).
d'2 : eps(fa_i (x => fa_i (y => or (eq (S x) (S y)) (not (eq x y))))).
d'3 : eps(fa_i (x => not (eq (plus x one) two))).
def c'1 : (eps(eq 0 0) -> eps bot) -> eps bot := l1 => l1 d'1.
def c'2 : x : i -> y : i -> (eps (eq (S x) (S y)) -> eps bot) ->
    (eps (not (eq x y)) -> eps bot) -> eps bot
    := x => y => l1 => l2 => z => d'2 x y z (l1' => l1 11' z) (12' => 12 12' z).
def c'3 : x : i -> (eps (not (eq (plus x one) two)) -> eps bot) ->
    eps bot := x => l1 => l1 (d'3 x).
def c'4 : y : i -> (eps (not (eq (S (plus y one)) two)) -> eps bot) ->
    eps bot := y => l1 => c'3 (S y) l1.
def c'5 : y : i -> (eps (not (eq (plus y one) one)) -> eps bot) ->
    eps bot := y => l1 => c'2 (plus y one) one
    (tp => c'4 y (tpn => tnp tp)) l1.
def c'6 : (eps (not (eq one one)) -> eps bot) -> eps bot
    := l1 => c'5 0 l1.
def c'7 : (eps (not (eq 0 0)) -> eps bot) -> eps bot
Expressing Theories in the $\lambda\Pi$-Calculus Modulo Theory and in the Dedukti System

### 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 $P \lor \neg P$ has a classical proof and no constructive one. Instead, we shall say, that the proposition $P \lor_c \neg_c P$ has a proof and the proposition $P \lor \neg P$ does not, distinguishing the classical connectives $\lor_c, \neg_c...$ from the constructive ones $\lor, \neg...$

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 $A \lor_c B$ as $\neg \neg (A \lor B)$. This approach has been investigated in [4, 37, 61]. 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 incorporate these negations. This has lead to various solutions: considering two different provability predicates [4], considering two entailment relations [37], incorporating the double negation to the predicate symbol at the root of $P$ [61].

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

\[
t ::= x \mid f(t, \ldots, t)
\]

\[
A ::= P(t_1, \ldots, t_n) \mid \top \mid \bot \mid \neg A \mid A \Rightarrow A \mid A \land A \mid A \lor A \mid \forall x A \mid \exists x A
\]

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

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

\[
t ::= x \mid f(t, \ldots, t)
\]

\[
a ::= P(t_1, \ldots, t_n)
\]

\[
A ::= a \mid \top \mid \bot \mid \neg A \mid A \Rightarrow A \mid A \land A \mid A \lor A \mid \forall x A \mid \exists x A
\]

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

\[
A ::= \triangleright a \mid \top \mid \bot \mid \neg A \mid A \Rightarrow A \mid A \land A \mid A \lor A \mid \forall x A \mid \exists x A
\]
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

\[
\begin{align*}
\triangleright_c A &= \neg\neg A \\
A \land_c B &= \neg\neg(A \land B) \\
A \lor_c B &= \neg\neg(A \lor B)
\end{align*}
\]

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

```lean
alpha : Type.
def atom : alpha \to o.
def atom_c : alpha \to o := \lambda p: alpha \Rightarrow \textsf{not} (\textsf{not} (\textsf{atom} p)).
def top_c : o := \textsf{not} (\textsf{not} \bot).
def bot_c : o := \textsf{not} (\textsf{not} \top).
def not_c : o \to o := \lambda x:o \Rightarrow (\textsf{not} (\textsf{not} x)).
def and_c : o \to o \to o := \lambda x:o \Rightarrow y:o \Rightarrow (\textsf{not} (\textsf{not} (\textsf{and} x y))).
def or_c : o \to o \to o := \lambda x:o \Rightarrow y:o \Rightarrow (\textsf{not} (\textsf{not} (\textsf{or} x y))).
def imp_c : o \to o \to o := \lambda x:o \Rightarrow y:o \Rightarrow (\textsf{not} (\textsf{imply} x y)).
def fa_s_c : (s \to o) \to o := \lambda x:s \Rightarrow (\textsf{not} (\textsf{not} (\textsf{fa}_s x))).
def ex_s_c : (s \to o) \to o := \lambda x:s \Rightarrow (\textsf{not} (\textsf{not} (\textsf{ex}_s x))).
```

The embedding of languages is as in Definitions 9 and 15, except that the predicate
symbol now have a target type \( \text{alpha} \), instead of \( \text{Type} \), 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 \( \lambda \)-term as

\[
\begin{align*}
|P(t_1, \ldots, t_n)| &= \textsf{atom} \ (P \ |t_1| \ \ldots \ |t_n|), \\
|T| &= \textsf{top}, \ |\bot| = \textsf{bot}, \\
|\neg A| &= \textsf{not} \ |A|, \\
|A \Rightarrow B| &= \textsf{impl} \ |A| \ |B|, \ |A \land B| &= \textsf{and} \ |A| \ |B|, \ |A \lor B| &= \textsf{or} \ |A| \ |B|, \\
|\forall x \ A| &= \textsf{fa}_s \ (x : s \Rightarrow |A|), \ |\exists x \ A| = \textsf{ex}_s \ (x : s \Rightarrow |A|).
\end{align*}
\]

We embed \( A \) classically as a \( \lambda \)-term as

\[
\begin{align*}
|P(t_1, \ldots, t_n)|_c &= \textsf{atom}_c \ (P \ |t_1| \ \ldots \ |t_n|), \\
|T|_c &= \textsf{top}_c, \ |\bot|_c = \textsf{bot}_c, \\
|\neg A|_c &= \textsf{not}_c \ |A|, \\
|A \Rightarrow B|_c &= \textsf{impl}_c \ |A|_c \ |B|_c, \ |A \land B|_c = \textsf{and}_c \ |A|_c \ |B|_c, \ |A \lor B|_c = \textsf{or}_c \ |A|_c \ |B|_c, \\
|\forall x \ A|_c &= \textsf{fa}_s_c \ (x : s \Rightarrow |A|_c), \ |\exists x \ A|_c = \textsf{ex}_s_c \ (x : s \Rightarrow |A|_c).
\end{align*}
\]

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

▶ **Lemma 20.** If \( A \) is a proposition in the language \( \mathcal{L} \) and \( \Gamma \) 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
\( \text{eps} \ |A| \) is inhabited. A proposition \( A \) has a proof in classical logic, if and only if the type
\( \text{eps} \ |A|_c \) is inhabited.
For instance, the proposition

$$\vdash_c P \lor_c \neg_c P$$

has the following proof, where for convenience, a redex has been introduced

```lean
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, as the symbols $\lor$ and $\lor_c$ have different meanings, the propositions

$$\forall x \forall y \forall z (\vdash x \in \{y, z\} \leftrightarrow (\vdash x = y \lor \vdash x = z))$$

$$\forall_c x \forall_c y \forall_c z (\vdash_c x \in \{y, z\} \leftrightarrow (\vdash_c x = y \lor \vdash_c x = z))$$

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

The translation of a rewrite rule of classical Deduction modulo theory such as

$$\vdash_c x \in \{y, z\} \rightarrow (\vdash_c x = y) \lor (\vdash_c x = z)$$

could be expressed directly in Dedukti, but it would introduce a critical pair because $\vdash_c P$ also reduces to $\neg\neg_P$. To close this critical pair, we rather add the rule

$$\vdash x \in \{y, z\} \rightarrow (\vdash_c x = y) \lor (\vdash_c x = z)$$

that subsumes the first one. 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.

Note that, in the same way we have two versions of the pairing axiom above, this rule is different from the translation of the rewrite rule of constructive Deduction modulo theory

$$\vdash x \in \{y, z\} \rightarrow (\vdash x = y) \lor (\vdash x = z)$$

### 5.3 Zenon proofs

Zenon is a proof search system for classical predicate logic with equality, based on a tableau method [19], that is a sequent calculus with one-sided sequents, where all propositions are on left-hand side. Zenon Modulo [33, 22] is an extension of Zenon to Deduction modulo theory. A proof of a proposition $G$ in a context $\Gamma$ is a tableau 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 \( (\text{eps} \ G|c) \) is inhabited in DEDUKTI. When proving a proposition \( G \), ZENON MODULO produces a DEDUKTI proof-term, of type \( (\text{eps} \ G|c) \) [26].

More precisely it first produces a DEDUKTI term of type \( \text{eps} \ (\text{not}_c \ G|c) \rightarrow \text{eps} \ \text{bot} \). This term can then be turned into a term of type \( (\text{eps} \ G|c) \) by applying the constructive theorem \( \neg\neg\neg\ A \Rightarrow \neg\ A \) twice, because \( G|c \) starts with a negation.

These proofs of \( \text{eps} \ (\text{not}_c \ G|c) \rightarrow \text{eps} \ \text{bot} \) are generated by stating in DEDUKTI one lemma per tableau rule. For instance, the tableau rule

\[
\frac{A \lor_c B}{A} \quad B \quad \text{Ror}
\]

that corresponds to the sequent calculus rule

\[
\frac{\Gamma, A \vdash \Gamma, B \vdash \lor} {\Gamma, A \lor_c B \vdash \lor\text{-left}}
\]

is reflected by the following DEDUKTI lemma

\[
\text{def Ror : } A : o \rightarrow B : o \rightarrow \\
(\text{eps} \ A \rightarrow \text{eps} \ \text{bot}) \rightarrow \\
(\text{eps} \ B \rightarrow \text{eps} \ \text{bot}) \rightarrow \\
\text{eps} \ (\text{or}_c A \ B) \rightarrow \text{eps} \ \text{bot} \\
:= A \Rightarrow B \Rightarrow HNA \Rightarrow HNB \Rightarrow HNNAB \Rightarrow HNNAB \ (HAB \Rightarrow HAB \ \text{bot} \ HNA \ HNB).
\]

The translation of ZENON MODULO proofs to DEDUKTI has been tested using a benchmark of 9,994 B Method [2] proof obligations, representing 5.5 GiB of input files. The generated DEDUKTI files weigh a total of 595MB, 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

\[
\text{def mem : s \rightarrow s \rightarrow alpha.} \\
\text{def union : s \rightarrow s \rightarrow s.} \\
[a, b, x] \text{ atom (mem x (union a b))} \\
\rightarrow (\text{or} (\text{atom}_c (\text{mem x a})) (\text{atom}_c (\text{mem x b}))).
\]

### 6 Simple type theory

#### 6.1 Simple type theory

There are two ways to express Simple type theory [28], also known as Higher-order logic, in the \( \lambda\Pi \)-calculus modulo theory. First, Simple type theory can be expressed in Deduction modulo theory [38], 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 [43, 44], hence in the \( \lambda\Pi \)-calculus modulo theory [31], which leads to a similar result.

In Simple type theory, the sorts of the logic are the types of the simply-typed \( \lambda \)-calculus, with two basic types: \( \iota \) for the objects and \( o \) for the propositions. The language contains the variables \( \Rightarrow \), of type \( o \rightarrow o \rightarrow o \), 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 \text{type}. 

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 \texttt{type}, which is the type of objects representing types of the Simple type theory, with \texttt{Type}, which is the type of \texttt{Dedukti} types. We endow the arrow types of the \texttt{type} level, the behavior of arrow types of the \texttt{Type} level, by adding the rule

\[
[a, b] \text{term} (\text{arrow} a b) \rightarrow \text{term} a \rightarrow \text{term} b.
\]

\begin{definition}
The terms of Simple type theory are embedded as \texttt{Dedukti} terms as follows
\begin{itemize}
  \item $[x] = x$,
  \item $[\Rightarrow] = \text{imp}$,
  \item $[\forall A] = \text{forall} \ [A]$,
  \item $[M, N] = [M] [N]$,
  \item $[\lambda x : A \ M] = x : (\text{term} \ A) \Rightarrow [M]$.
\end{itemize}

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

\begin{verbatim}
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).
\end{verbatim}

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

\begin{theorem}
(\text{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 $\lambda$-term $\pi$ such that $\Sigma_{STT}, \|\Gamma\| \vdash \pi : \|A\|$.
\end{theorem}

Moreover, the term $\pi$ is a straightforward encoding of the original proof tree.

\subsection{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 $Q_0$ [5], 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 allows for the expression HOL-Light proofs in \texttt{Dedukti}, building on the Open theory project [49]. HOLiDE supports all the features of HOL-Light, including ML-polymorphism, constant definitions, and type definitions. It is able to express all of the OpenTheory standard theory library. The translation of this 10 MB library results in a \texttt{Dedukti} library of 21.5 MB, when gzipped.
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.

7.1 Lambda-calculus

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

| x | = x,
| t u | = app |t| |u|,
| λx t | = lam (x:term => |t|).

in the signature

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

Encoding programming languages in the λΠ-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 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—usually terminates.

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

```
type : Type.
arrrrow : 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) ->
and prove properties of recursive terminating functions defined using this \texttt{fix} operator, such as the function \texttt{mod2} defined below.

\begin{verbatim}

nat : type.
def Nat := term nat.
O : Nat.
S : Nat -> Nat.
def match_nat : Nat -> Nat -> (Nat -> Nat) -> Nat.
[n0,nS] match_nat O 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 O)))) , O.
\end{verbatim}

The \texttt{#CONV} command in last line checks that the terms \texttt{mod2 (S (S (S (S O))))} and \texttt{0} are convertible.

We have successfully used \texttt{Dedukti} to express both object-oriented and functional programs via the \texttt{ς}-calculus and ML.

### 7.2 The \texttt{ς}-calculus

Abadi and Cardelli [1] 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 \texttt{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 \texttt{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

\begin{verbatim}

label : Type.
type : Type.
typenil : type.
typecons : label -> type -> type -> type.
\end{verbatim}

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

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:

\[
\begin{align*}
a, b, & \ldots ::= x \quad \text{(variable)} \\
& | [l_i = ς(x_i : A) a_i]_{i \in 1...n} \quad \text{(object)} \\
& | a. l \quad \text{(method selection)} \\
& | a. l \leftarrow ς(x : A) b \quad \text{(method update)}
\end{align*}
\]
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 binder refers to the type of the full object. In order to define partially constructed objects as lists, we introduce the notion of preobject.

\[
\text{Preobj} : \text{type} \rightarrow \text{type} \rightarrow \text{Type}.
\]

The first type argument of the type \text{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

\[
\text{def Obj} (A : \text{type}) := \text{Preobj} A A.
\]

The constructors of preobjects are

\[
\begin{align*}
\text{prenil} : & \ A : \text{type} \rightarrow \text{Preobj} A \text{typenil}. \\
\text{precons} : & \ A : \text{type} \\
& \quad \ B : \text{type} \\
& \quad \ l : \text{label} \\
& \quad \ C : \text{type} \\
& \quad \ (\text{Obj} A \rightarrow \text{Obj} C) \\
& \quad \ \text{Preobj} A B \rightarrow \text{Preobj} A (\text{typecons} l C B).
\end{align*}
\]

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 \text{preupdate} returns a preobject of the same type whereas the selection function on preobjects \text{preselect} returns a function defined on \text{Obj} A:

\[
\begin{align*}
\text{mem} : & \ \text{label} \rightarrow \text{type} \rightarrow \text{type} \rightarrow \text{Type}.
\text{athead} : & \ l : \text{label} \rightarrow A : \text{type} \rightarrow B : \text{type} \rightarrow \text{mem} l A (\text{typecons} l A B).
\text{intail} : & \ l : \text{label} \rightarrow A : \text{type} \rightarrow l' : \text{label} \rightarrow A' : \text{type} \\
& \quad \rightarrow B : \text{type} \rightarrow \text{mem} l A B \rightarrow \text{mem} l A (\text{typecons} l' A' B).
\end{align*}
\]

\[
\begin{align*}
\text{def preselect} : & \ A : \text{type} \\
& \quad \ B : \text{type} \\
& \quad \ l : \text{label} \\
& \quad \ C : \text{type} \\
& \quad \ \text{mem} l C B \\
& \quad \ \text{Preobj} A B \\
& \quad \ (\text{Obj} A \rightarrow \text{Obj} C).
\end{align*}
\]

\[
\begin{align*}
\text{def preupdate} : & \ A : \text{type} \\
& \quad \ B : \text{type} \\
& \quad \ l : \text{label} \\
& \quad \ C : \text{type} \\
& \quad \ \text{mem} l C B \\
& \quad \ \text{Preobj} A B \\
& \quad \ (\text{Obj} A \rightarrow \text{Obj} C) \\
& \quad \ \text{Preobj} A B.
\end{align*}
\]

Both functions are defined by structural induction on their argument of type \text{mem} l C B where \text{mem} is the inductive relation representing membership of the couple \((l,\text{C})\) in 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) :=
preselect A A l C p a a.
def update (A : type) :=
preupdate A A.

Definition 24 (Expression of the $\varsigma$-calculus). The encoding of $\varsigma$-terms is then defined by

$|x| = x,$

$|[l_i = \varsigma(x : A)a_i]|_{i \in 1...n, l_i < ... < l_n} =
\text{precons } |A| |[| |]_{i \in 2...n} |A_1| (x : \text{Obj } |A| \Rightarrow |a_i|)
(\ldots \text{precons } |A| |[| |]_n |A_n| (x : \text{Obj } |A| \Rightarrow |a_n|) (\text{prenil } |A|) \ldots),$

$|a.l_i| = \text{select } |A| |A_1| p_1 |a|,$

$|a.l_i := \varsigma(x : A)c| = \text{update } |A| |A_1| p_1 |a| (x : \text{Obj } |A| \Rightarrow |c|).$

where $p_1$ is a proof that $A$ contains $l_i : A_i$.

Lemma 25 (Shallow encoding, Theorems 13 and 18 in [25]). This encoding preserves binding, typing, and the operational semantics of the simply-typed $\varsigma$-calculus.

This translation of the simply-typed $\varsigma$-calculus in the $\lambda\Pi$-calculus modulo theories can be extended to subtyping, see [25].

### 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. Both features are present in the $\lambda\Pi$-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 $\text{true}$) by binding the subterm $t$.

Definition 26. Let $C$ be a datatype constructor of type $\tau \rightarrow \tau'$, the destructor associated with $C$ is the symbol $\text{destr}_C$, of type $\tau : \text{type} \rightarrow (\text{eps } \tau' \rightarrow (\text{eps } \tau \rightarrow \text{eps } \tau' \rightarrow \text{eps } \tau' \rightarrow \text{eps } \tau') \rightarrow \text{eps } \tau')$, defined by the following rewrite rules:

$\text{destr}_C \tau f d \rightarrow f t$

$\text{destr}_C C' t' f d \rightarrow d$ (for all other constructors $C'$ for type $\tau'$)

Lemma 27 (Theorem 1 in [24]). 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 $\Phi$, of type $\tau : \text{type} \rightarrow \text{eps } \tau \rightarrow (\text{eps } \tau \rightarrow \text{eps } \tau \rightarrow \text{eps } \tau \rightarrow \text{eps } \tau')$, defined for each constructor $C$ of type $\tau \rightarrow \tau'$ by the following rewrite rule

$\Phi \tau' \tau f (C t) \rightarrow f (C t)$

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 \(\lambda\Pi\)-calculus modulo theory is to check program certificates. The FoCaLiZe system [41] 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 \(\text{mod2}\). It can be defined in FoCaLiZe by

```ml
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)

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

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

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

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

let rec 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

\[\begin{align*}
\text{theorem nat_induction :} \\
\quad \forall p : (\text{nat} \to \text{bool}), \\
\quad \quad p(\text{Zero}) \to \\
\quad \quad \quad \forall n : \text{nat}, p(n) \to p(\text{Succ}(n)) \to \\
\quad \quad \quad \quad \forall n : \text{nat}, p(n) \\
\quad \text{proof} = \text{assumed};;
\end{align*}\]

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

\[\begin{align*}
\text{let mod2_twice_n_is_Zero \ (n : \text{nat}) : \text{bool} = (\text{mod2(twice(n))} = \text{Zero});;}
\end{align*}\]

\[\begin{align*}
\text{theorem mod2_twice :} \\
\quad \forall n : \text{nat}, \text{mod2(twice(n))} = \text{Zero} \\
\quad \text{proof} = \\
\quad \quad \text{dedukti proof} \\
\quad \quad \quad \{^{*} \text{nat_induction mod2_twice_n_is_Zero mod2_twice_Zero mod2_twice_Succ.} ^{*}\};;
\end{align*}\]

More than 98% of the FoCALiZE 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 [13] 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 [31] using ideas similar to the expression of Simple type theory, see Section 6.1, and to the mechanisms of universes à la Tarski in Intuitionistic type theory [53]. 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 \(\text{eps}_s|A|\) [31].

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

\[\begin{align*}
U_s & : \text{Type.} \\
\text{def } e_s & : U_s \to \text{Type.}
\end{align*}\]

For each axiom \((s_1, s_2) \in A\), we declare the variable \(u_{s_1}\) 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.

\[\begin{align*}
u_{s1} & : U_{s2}. \\
[ ] & e_{s2} u_{s1} \to U_{s1}.
\end{align*}\]

For each rule \((s_1, s_2, s_3) \in R\), we declare the variable \(\pi_{s_1,s_2}\) 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.
\[ \pi_{s_1s_2} : a : U_{s_1} \rightarrow b : (e_{s_1} a \rightarrow U_{s_2}) \rightarrow U_{s_3}. \]
\[ [a, b] e_{s_3} (\pi_{s_1s_2} a b) \rightarrow x : e_{s_1} a \rightarrow e_{s_2} (b x). \]

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

- \(|x| = x,\)
- \(|s| = u_s,\)
- \(|M N| = |M| \cdot |N|,\)
- \(|\lambda x : A M| = x : |A| \rightarrow |M|,\)
- \(|\Pi x : A B| = \pi_{s_1s_2} |A| (x : |A| \rightarrow |B|)\) (where \(\Gamma \vdash A : s_1\) and \(\Gamma, x : A \vdash B : s_2\)),
- \(|A| = e_s |A|\) (when \(\Gamma \vdash A : s\)),
- \(|s| = u_s\) (when \(\exists s'\) such that \(\Gamma \vdash s : s'\)),
- \(|[ ]| = [],\)
- \(|\Gamma, x : A| = |\Gamma|, x : |A||.\)

*Lemma 30* (Preservation of computation, Proposition 1 in [31]). If \(\Gamma \vdash M : A\) and \(M \rightarrow_\beta M'\), then there exists \(N'\) such that \(|M| \rightarrow_\beta N' \equiv_\beta |M'|.\)

*Lemma 31* (Preservation of typing, Proposition 3 in [31]). 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 \(\Sigma, |\Gamma|| \vdash |A||,\) then there exists \(M\) such that \(\Gamma \vdash M : A\) and \(|M| \equiv_{\beta\eta} M'.\)

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

\[
S ::= \{\text{Type, Kind}\} \\
A ::= \{(\text{Type, Kind})\} \\
R ::= \{(\text{Type, Type, Type}), (\text{Type, Kind, Kind}), \\
(K\text{ind, Type, Type}), (\text{Kind, Kind, Kind})\}
\]

and it is expressed in **Dedukti** as

\[
\begin{align*}
U_{\text{Type}} & : \text{Type}. \\
U_{\text{Kind}} & : \text{Type}. \\
def e_{\text{Type}} & : U_{\text{Type}} \rightarrow \text{Type}. \\
def e_{\text{Kind}} & : U_{\text{Kind}} \rightarrow \text{Type}. \\
u_{\text{Type}} & : U_{\text{Kind}}. \\
\pi_{\text{TypeType}} & : a : U_{\text{Type}} \rightarrow b : (e_{\text{Type}} a \rightarrow U_{\text{Type}}) \rightarrow U_{\text{Type}}. \\
\pi_{\text{TypeKind}} & : a : U_{\text{Type}} \rightarrow b : (e_{\text{Type}} a \rightarrow U_{\text{Kind}}) \rightarrow U_{\text{Kind}}. \\
\pi_{\text{KindType}} & : a : U_{\text{Kind}} \rightarrow b : (e_{\text{Kind}} a \rightarrow U_{\text{Type}}) \rightarrow U_{\text{Type}}. \\
\pi_{\text{KindKind}} & : a : U_{\text{Kind}} \rightarrow b : (e_{\text{Kind}} a \rightarrow U_{\text{Kind}}) \rightarrow U_{\text{Kind}}.
\end{align*}
\]
Expressing Theories in the $\lambda\Pi$-Calculus Modulo Theory and in the Dedukti System

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

\[
\begin{align*}
a & : U\_Type. \\
b & : U\_Type. \\
x & : e\_Type a. \\
f & : (p : (e\_Type a \to U\_Type) \to e\_Type (p x) \to e\_Type b). \\
def example : e\_Type b := f (y : e\_Type a => a) x.
\end{align*}
\]

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 $\lambda\Pi$-calculus modulo theory is to use constructors and a primitive recursion operator as in Gödel’s system T [45]. 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:

\[
\begin{align*}
\text{list} & : U\_Type \to U\_Type. \\
\text{nil} & : A : U\_Type \to e\_Type (\text{list} A). \\
\text{cons} & : A : U\_Type \to e\_Type A \to e\_Type (\text{list} A) \to e\_Type (\text{list} A).
\end{align*}
\]

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:

\[
\begin{align*}
\text{List} & : U\_Type \to Type. \\
[A] e\_Type (\text{list} A) \to \text{List} A. \\
def \text{elim}_\text{list} & : A : U\_Type \to P : (e\_Type (\text{list} A) \to U\_Type) \\
& \to e\_Type (P (\text{nil} A)) \to (x : e\_Type A \to \text{List} A) \\
& \to e\_Type (P \text{12}) \to e\_Type (P (\text{cons} A \times 12))) \to l : e\_Type (\text{list} A) \\
& \to e\_Type (P \text{1}).
\end{align*}
\]

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 1 for any list 1. Finally, we express the computational behavior of the eliminator, that reduces when applied to a constructor.
Recursive functions and inductive proofs can then be expressed using this elimination operator. For instance, the append function is

```lean
def append : A : U_Type → List A → List A → List A :=
  A : U_Type => l1 : List A => l2 : List A =>
  elim_list A (λ _ => list A) l2 (x => l3 => 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 [17]. 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 [53]. 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

\[
\frac{}{\Gamma \vdash M : U_i} \qquad \frac{\Gamma \vdash M : U_i}{\Gamma \vdash M : U_{i+1}}
\]

To deal with the infinite universe hierarchy, instead of declaring a different variable for each universe, we index the variable \( U, \varepsilon, u, \pi \) by natural numbers.

```lean
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 $U_{i+1}$.

\[
\text{lift} : i : \text{nat} \rightarrow U_i \rightarrow U_{i+1}.
\]

\[
[i, a] \varepsilon (\text{lift} i a) \rightarrow \varepsilon i a.
\]

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 $U_{i+1}$: $\text{lift} i (\pi i |A| (x \Rightarrow |B|))$ and $\pi (S i) (\text{lift} i |A|) (x \Rightarrow \text{lift} i |B|)$. 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:

\[
[i, a, b] \pi (\text{lift} i a) (x \Rightarrow \text{lift} i (b x)) \rightarrow \\
\text{lift} i (\pi i a (x \Rightarrow b x)).
\]

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

### 8.4 Matita proofs

There are not many libraries of proofs expressed in the Calculus of constructions with universes. The Coq library is expressed in a Calculus of constructions with universes, modules and universe polymorphism. But universe polymorphism have not yet been expressed in Dedukti.

The Matita [6] library is expressed in a Calculus of constructions with universes, and proof irrelevance. Proof irrelevance has not yet been expressed in Dedukti, but 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 1.11MB, when gzipped.

### 9 Who did what?

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

The $\lambda\Pi$-calculus modulo theory appeared in a joint paper of Denis Cousineau and Gilles Dowek in 2007 [31], together with an expression of Pure Type Systems in it, and a partial correctness proof. The framework was further refined in [63] and then in [64]. In 2010, during his undergrad internship [34], Alexis Dorra showed that constructive predicate logic also could be expressed in the $\lambda\Pi$-calculus modulo theory.

The first implementation of Dedukti was designed by Mathieu Boespflug [16] between 2008 and 2011, during his thesis advised by Gilles Dowek. A second implementation was designed in 2012 by Quentin Carbonneaux during his Master thesis [23], advised by Olivier Hermant and Mathieu Boespflug. Finally, the current implementation has been designed by Ronan Saillard between 2012 and 2015 during his thesis [64], advised 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 [17]. 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 [67]. Guillaume Burel [21] 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. 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.

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?”

References

Expressing Theories in the $\lambda\Pi$-Calculus Modulo Theory and in the Dedukti System


18 M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$-calculus modulo as a universal proof language. In David Pichardie and Tjark Weber, editors, PxTP, 2012.


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


FoCaLiZe. [http://focalize.inria.fr](http://focalize.inria.fr).


60 H. Poincaré. La science et l’hypothèse. Flammarion, 1902.


