ML pattern-matching, recursion, and rewriting: from FoCaLiZe to Dedukti

Raphaël Cauderlier[1] and Catherine Dubois[2]

[1] Inria - Saclay and Cnam - Cedric
[2] ENSIIE - Samovar

Abstract. The programming environment FoCaLiZe allows the user to specify, implement, and prove programs. It produces as output OCaml executable programs along with proof hints that help the first-order theorem prover Zenon to find proofs. In the actual version, those proofs found by Zenon are verified by Coq. In this paper we propose to extend the FoCaLiZe compiler by a backend to the Dedukti language – a proof checker for Deduction modulo – in order to benefit from Zenon Modulo, an extension of Zenon for Deduction modulo. It produces proof terms for Dedukti. By doing so, FoCaLiZe can benefit from a technique for finding and verifying proofs more quickly and also for finding more concise proofs. The paper focusses mainly on the process that overcomes the lack of local pattern-matching and recursive definitions in Dedukti.

1 Introduction

FoCaLiZe [20] is an environment for certified programming. It allows the user to specify, implement, and prove. For implementation, FoCaLiZe provides an ML like functional language. FoCaLiZe proofs are delegated to the first-order theorem prover Zenon [4] which takes Coq problems as input and outputs proofs in Coq format for independent checking. Zenon has recently been improved to handle Deduction modulo [11], an efficient proof-search technique [6]. However, the Deduction modulo version of Zenon, Zenon Modulo, outputs proofs for the Dedukti proof checker [22] instead of Coq [8].

In order to benefit from the advantages of Deduction modulo in FoCaLiZe, we extend the FoCaLiZe compiler by a backend to Dedukti called Focalide[3] (see Figure 1). This work is also a first step in the direction of interoperability between FoCaLiZe and other proof languages for which translation tools to Dedukti exist such as the proof assistants of the HOL family [2].

This new compilation backend to Dedukti is based on the existing backend to Coq. While the compilation of types and logical formulae is a straightforward adaptation, the translation of FoCaLiZe terms to Dedukti is not trivial because Dedukti lacks two mechanisms found in functional languages and in Coq: local pattern-matching and recursive definitions.

[3] This work is available at `http://deducteam.gforge.inria.fr/focalide`

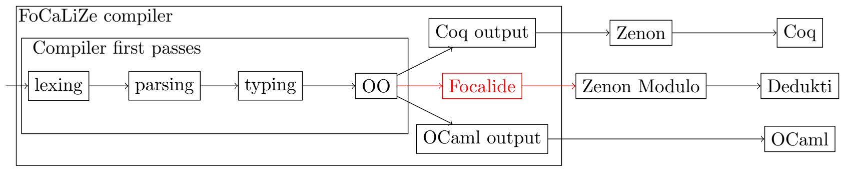

Fig. 1. FoCaLiZe Compilation Scheme

In the following, Section 2 contains a short presentation of Dedukti, Zenon Modulo, and FoCaLiZe. Then Section 3 presents the main features of the compilation to Dedukti. Section 4 focuses on compilation of pattern-matching and Section 5 on the compilation of recursive functions. In Section 6, the backend to Dedukti is evaluated on benchmarks. Section 7 discusses related work and Section 8 concludes the paper by pointing some future work.

2 Presentation of the tools

2.1 Dedukti

Dedukti [22] is a type checker for the $\lambda\Pi$-calculus modulo, an extension of a pure dependent type system, the $\lambda\Pi$-calculus, with rewriting. Through the Curry-Howard correspondence, Dedukti can be used as a proof-checker for a wide variety of logics [9, 2, 1]. It is commonly used to check proofs coming from Deduction modulo provers Iprover Modulo [5] and Zenon Modulo [8].

As a simple example, Figure 2 presents a possible definition of integers in Dedukti.

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

int : Type
def make : nat -> nat -> int.
[m,n] make (S m) (S n) --> make m n.
```

Fig. 2. Definition of integers in Dedukti

A Dedukti file consists of an interleaving of declarations (such as `0 : nat`) and rewrite rules (such as `[m,n] make (S m) (S n) --> make m n.`).

In Figure 2, the type `int` is defined from the type `nat` of Peano natural numbers. An integer is built from two natural numbers by taking their difference. The rewrite rule `make (S m) (S n) --> make m n` is used to normalize any closed term of type `int` to one of the following forms: `make 0 0`, `make (S m) 0`, or `make 0 (S n)`.

Declarations and rewrite rules are type checked modulo the previously defined rewrite rules. This mechanism can be used to perform proof by reflection, an example is given by the theorem `two_plus_two_is_four` of Figure 3.

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

equal : nat -> nat -> Type.
refl : n : nat -> equal n n.

def two_plus_two_is_four :
  equal (plus (S (S 0)) (S (S 0))) (S (S (S (S 0)))).
[] two_plus_two_is_four --> refl (S (S (S (S 0)))).
```

Fig. 3. A proof by reflection of $2 + 2 = 4$ in Dedukti

For correctness, Dedukti requires this rewrite system to be confluent; moreover, Dedukti does not guarantee to terminate when the rewrite system is not terminating.

2.2 Zenon Modulo

Zenon [4] is a first-order theorem prover based on the tableaux method. It is able to produce proof terms which can be checked independently by the Coq proof assistant.

Zenon Modulo [10] is an extension of Zenon for Deduction modulo, an extension of Predicate Logic distinguishing computation from reasoning. Computation is defined by a rewrite system such as the symmetric definition of addition given in Figure 3, it is part of the theory. Reasoning is defined by a usual deduction system for Predicate Logic (Sequent Calculus in the case of Zenon Modulo) for which syntactic comparison is replaced by the congruence induced by the rewrite system. Computation steps are left implicit in the resulting proof which has to be checked in Dedukti.

Zenon (resp. Zenon Modulo) accepts input problems in Coq (resp. Dedukti) format so that it can be seen as a term synthesizer: its input is a typing context and a type to inhabit, its output is an inhabitant of this type. This is the mode of operation used when interacting with FoCaLiZe because it limits ambiguities and changes in naming schemes induced by translation tools between languages.

2.3 FoCaLiZe and its compilation process

This subsection presents briefly FoCaLiZe and its compilation process (for details please see [20] and FoCaLiZe reference manual). We address more precisely the focalizec compiler which produces OCaml and Coq code.

The FoCaLiZe (`http://focalize.inria.fr`) environment provides a set of tools to formally specify and implement functions and logical statements together with their proofs. The FoCaLiZe language has an object oriented flavor allowing (multiple) inheritance, late binding and redefinition. These characteristics are very helpful to reuse specifications, implementations and proofs.

A FoCaLiZe specification can be seen as a set of algebraic properties describing relations between input and output of the functions implemented in a FoCaLiZe program. For implementing, FoCaLiZe offers a pure functional programming language close to ML, featuring a polymorphic type system, recursive functions, data types and pattern-matching. Proofs are written in a declarative style and can be considered as a bunch of hints that the automatic prover Zenon uses to produce proofs that can be verified by Coq for more confidence [4].

Program units in FoCaLiZe define types together with functions and properties applying to them. At the beginning of a development, types are usually abstract, they are precised later in the development. More precisely a unit may specify a function or a property or implement them by respectively providing a definition or a proof. A defined function must match its signature and similarly a proof should prove its statement. Statements belong to first-order typed logic.

A FoCaLiZe source program is analyzed and translated into OCaml sources for execution and Coq sources for certification. The compilation process between both target languages is shared as much as possible. The architecture of the FoCaLiZe compiler is shown in Figure 1. The FoCaLiZe compiler integrates a type checker in order to early detect type errors and emit comprehensive error messages. Inheritance and late binding are resolved at compile-time (OO on Figure 1), relying on a dependency calculus described in [20]. To deal with late binding, code generation relies on λ-lifting. The process for compiling proofs towards Coq is achieved in 2 steps: a first one compiles the statement with holes instead of the proof script. The goal together with the context is also transmitted to Zenon. Then when the proof has been found, the hole is filled with the proof output by Zenon.

3 From FoCaLiZe to Focalide

As said previously, Focalide is adapted from the Coq backend. In particular it benefits from the early compilation steps. In this section, we describe the input language we have to consider and the main principles of the translation to Dedukti.

3.1 Input language

Focalide input language is simpler than FoCaLiZe, in particular because the initial compilation steps get rid of object oriented features (see Figure 1). So for generating code to Dedukti, we can consider that a program is a list of type definitions, well-typed function definitions and proved theorems. Figure 4 describes the syntax of the language taken as input by Focalide. A type definition defines a type *à la ML*, in particular it can be the definition of an algebraic datatype in which value constructors are listed together with their type. In these definitions, c denotes a constant, C a value constructor (from an algebraic datatype), *ident* a possibly qualified name, x a variable. When applied, a function must receive all its parameters. So partial application must be named. FoCaLiZe supports the usual patterns found in functional languages such as OCaml or Haskell: a pattern can either be a variable, a wildcard, a literal constant of some built-in-type (such as a literal integer or a literal character), a named pattern, or a constructor applied to as many sub-patterns than the constructor arity. Moreover, patterns have to be linear. An atomic formula is a boolean expression, that is the reason why the logical expressions grammar embed expressions.

Pre-defined types:	$i ::= \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \mid \texttt{string}$
Types:	$\tau ::= \alpha \mid i \mid a(\tau_1, \dots, \tau_k) \mid \tau \to \tau$
Type definitions:	$\texttt{type}\ a\ (\alpha_1, \dots, \alpha_k)\ =$ $\mid C_1(\tau_{1,1}, \dots, \tau_{1,k_1}) \mid \dots \mid C_n(\tau_{n,1}, \dots, \tau_{n,k_n})$
Constants:	$c ::= () \mid \texttt{true} \mid \texttt{false} \mid$ number literal $\mid$ string literal
Expressions:	$e ::= c \mid ident \mid \textbf{let}\ [\textbf{rec}]\ x\ :=\ e\ \textbf{in}\ e \mid \lambda(x_1, \dots, x_n).\ e$ $\mid e(e_1, \dots, e_n) \mid C(e_1, \dots, e_n) \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \mid e = e$ $\mid \textbf{match}\ e\ \textbf{with} \mid p \Rightarrow\ e\ \dots \mid p \Rightarrow\ e$
Patterns:	$p ::= x \mid \_ \mid c \mid p\ \textbf{as}\ x \mid C(p, \dots, p)$
Logical formulas:	$\varphi ::= e \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \forall x : \tau.\varphi \mid \exists x : \tau.\varphi$

Fig. 4. Focalide input syntax

3.2 Translation

Basic types such as `int` are mapped to their counterpart in the target proof checker. However there is no standard library in Dedukti, so we defined the Dedukti counterpart for the different FoCaLiZe basic types. It means defining the type and its basic operations together with the proofs of some basic properties.

The compilation of types is straightforward. It is also quite immediate for most of the expressions, except for pattern-matching expressions and recursive functions because Dedukti, contrary to Coq, lacks these two mechanisms. Thus we have to use other Dedukti constructions to embed their semantics. The compilation of pattern-matching expressions and recursive functions is detailed in next sections. Other constructs of the language such as abstractions and applications are directly mapped to the same construct in Dedukti.

The statement of a theorem is compiled in the input format required by Zenon Modulo, which is here Dedukti itself [8].

4 Compilation of pattern-matching

Pattern-matching is a useful feature in FoCaLiZe which is also present in Dedukti. However pattern-matching in Dedukti is only available at toplevel (rewrite rules cannot be introduced locally) and both semantics are different. FoCaLiZe semantics of pattern-matching is the one of functional languages: only values are matched and the first branch that applies is used. In Dedukti however, reduction can be triggered on open terms and the order in which the rules are applied should not matter since the rewrite system is supposed to be confluent.

To solve these issues, we define new symbols called *destructors*, using toplevel rewrite rules and apply them locally.

4.1 Semantics of pattern-matching in FoCaLiZe

The semantics of pattern-matching in FoCaLiZe is not surprising: to evaluate the expression **match** a **with** $\mid p_1 \Rightarrow\ e_1\ \dots\ \mid p_n \Rightarrow\ e_n$, the matched expression a is first reduced to a value v and then the expression reduces to e_i where i is the smallest index such that v matches p_i. Formally, it can be defined by the reduction rules of Figure 5.

- **match** a **with** $\mid p_1 \Rightarrow\ e_1\ \dots\ \mid p_n \Rightarrow\ e_n \rightsquigarrow$ **match** b **with** $\mid p_1 \Rightarrow\ e_1\ \dots\ \mid p_n \Rightarrow\ e_n$ when $a \rightsquigarrow b$
- **match** v **with** $\emptyset \rightsquigarrow$ **ERROR**
- **match** v **with** $\mid p_1 \Rightarrow\ e_1\ \dots\ \mid p_n \Rightarrow\ e_n \rightsquigarrow$ **match** v **with** $\mid p_2 \Rightarrow\ e_2\ \dots\ \mid p_n \Rightarrow\ e_n$ when p_1 and v are not unifiable
- **match** v **with** $\mid p_1 \Rightarrow\ e_1\ \dots\ \mid p_n \Rightarrow\ e_n \rightsquigarrow \sigma(e_1)$ where $\sigma = mgu(v, p_1)$ is the most general unifier of v and p_1.

Fig. 5. Call-by-value operational semantics of pattern-matching in FoCaLiZe

We denote by $\equiv$ the congruence generated by $\rightsquigarrow$ and we say that two expressions a and b are semantically equivalent when $a \equiv b$.

4.2 Translating pattern-matching to destructors

If C is a constructor of arity n for some datatype, the *destructor* associated with C is the expression $\lambda a, b, c.$ **match** a **with** $\mid C(x_1, \dots, x_n) \Rightarrow\ b\, x_1\ \dots\ x_n \mid \_ \Rightarrow\ c$. We say that a pattern-matching has *the shape of a destructor* if it is a fully applied destructor.

In this subsection, we show how to translate FoCaLiZe expressions to its fragment where each pattern-matching has the shape of a destructor. This shape is easy to translate to Dedukti because we only need to define the destructor associated with each constructor.

The transformation to this fragment is done in two steps: we first *serialize* pattern-matching so that each pattern-matching has exactly two branches and the second pattern is a wildcard, and we then *flatten* patterns so that the only remaining patterns are constructors applied to variables.

```
$\mathcal{S}(x) := x$
$\mathcal{S}(f(a_1, \ldots, a_n)) := f(\mathcal{S}(a_1), \ldots, \mathcal{S}(a_n))$
$\ldots$
$\mathcal{S}(\textbf{match } a \textbf{ with} \mid p_1 \Rightarrow e_1 \ldots \mid p_n \Rightarrow e_n) := \textbf{let } x := a \textbf{ in}$
  $\textbf{match } x \textbf{ with}$
  $\mid p_1 \Rightarrow \mathcal{S}(e_1)$
  $\mid \_ \Rightarrow \textbf{match } x \textbf{ with}$
    $\mid p_2 \Rightarrow \mathcal{S}(e_2)$
    $\mid \_ \Rightarrow \ldots \textbf{match } x \textbf{ with}$
      $\mid p_n \Rightarrow \mathcal{S}(e_n)$
      $\mid \_ \Rightarrow \textbf{ERROR}$            *where $x$ is a fresh variable*
```

Fig. 6. Definition of the function $\mathcal{S}$ for serialization of pattern-matching

Serialization of pattern-matching The serialization is the program transformation process by which pattern-matching with an arbitrary number of branches n becomes a sequence of pattern-matching with only two branches (and all second patterns are wildcards). Moreover, in order to avoid useless duplication of the matched term, we further restrict the allowed pattern-matchings to the case where the matched term is a variable. We define this program transformation formally by a function $\mathcal{S}$ which is defined in Figure 6.

This transformation is linear and preserves semantics.

Flattening of pattern-matching Now that pattern-matching has been serialized, we still need to compile pattern nesting; it means we want to restrict the shape of patterns to just constructors applied to variables. We introduce another program transformation $\mathcal{F}$ defined in Figure 7.

To prove the termination of $\mathcal{F}$, we define the notion of first-pattern size for an expression e as follows: if e is a pattern-matching then its first-pattern size is the size of its first pattern, and it is 0 otherwise. The lexical ordering $e_1 \leq e_2 \Leftrightarrow (\mathsf{size}_{1^{\mathsf{st}}\mathsf{pat}}(e_1), \mathsf{size}(e_1)) \leq_{\mathsf{lex}} (\mathsf{size}_{1^{\mathsf{st}}\mathsf{pat}}(e_2), \mathsf{size}(e_2))$ is well-founded and strictly decreasing at each recursive call of $\mathcal{F}$ so $\mathcal{F}$ terminates.

Flattening $\mathcal{F}$ preserves the semantics of pattern-matching:

Theorem 1. *For any expression e and any substitution θ from variables to values, the expressions $\theta(\mathcal{F}(e))$ and $\theta(e)$ are semantically equivalent.*

```
$\mathcal{F}(x) := x$
$\mathcal{F}(f(a_1, \ldots, a_n)) := f(\mathcal{F}(a_1), \ldots, \mathcal{F}(a_n))$
$\ldots$
$\mathcal{F}(\textbf{match } x \textbf{ with} \mid y \Rightarrow e \mid \_ \Rightarrow d) := \textbf{let } y := x \textbf{ in } \mathcal{F}(e)$
$\mathcal{F}(\textbf{match } x \textbf{ with} \mid \_ \Rightarrow e \mid \_ \Rightarrow d) := \mathcal{F}(e)$
$\mathcal{F}(\textbf{match } x \textbf{ with} \mid c \Rightarrow e \mid \_ \Rightarrow d) := \textbf{if } x = c \textbf{ then } \mathcal{F}(e) \textbf{ else } \mathcal{F}(d)$
$\mathcal{F}(\textbf{match } x \textbf{ with} \mid p \textbf{ as } y \Rightarrow e \mid \_ \Rightarrow d) :=$
    $\mathcal{F}(\textbf{match } x \textbf{ with} \mid p \Rightarrow \textbf{let } y := x \textbf{ in } e \mid \_ \Rightarrow d)$
$\mathcal{F}(\textbf{match } x \textbf{ with} \mid C(p_1, p_2, \ldots, p_n) \Rightarrow e \mid \_ \Rightarrow d) :=$
    $\textbf{let } f() := \mathcal{F}(d) \textbf{ in}$
    $\textbf{match } x \textbf{ with}$
    $\mid C(x_1, \ldots, x_n) \Rightarrow$
      $\mathcal{F}(\textbf{match } x_1 \textbf{ with}$
        $\mid p_1 \Rightarrow \mathcal{F}(\textbf{match } x_2 \textbf{ with}$
          $\mid p_2 \Rightarrow \ldots \mathcal{F}(\textbf{match } x_n \textbf{ with}$
            $\mid p_n \Rightarrow \mathcal{F}(e)$
            $\mid \_ \Rightarrow f()) \ldots$
          $\mid \_ \Rightarrow f())$
        $\mid \_ \Rightarrow f())$
    $\mid \_ \Rightarrow f()$
*where the variables $x_i$ and the function symbol $f$ are fresh*
```

Fig. 7. Definition of the function $\mathcal{F}$ for flattening of pattern-matching

Proof (sketch). The proof is done by induction on the structure of the function $\mathcal{F}$. The only non-trivial case is the case of nested patterns for which we have to distinguish cases: $\theta(x)$ is either a $C(v_1, \ldots, v_n)$ or not, in the former case it unifies or not with $C(p_1, \ldots, p_n)$. Pattern linearity is used to compute the unifier: $mgu(C(v_1, \ldots, v_n), C(p_1, \ldots, p_n)) = mgu(v_1, p_1) \circ \ldots \circ mgu(v_n, p_n)$.

All these results are very generic in the sense that we did not use any specific features of FoCaLiZe or Dedukti but only defined a linear program transformation procedure which simplifies ML pattern-matchings.

4.3 Destructors in Dedukti

Destructors are easy to write in Dedukti but a fully formal presentation of the implementation would be unnecessarily obscure. We will only consider the simple case of Peano natural numbers which can be written in FoCaLiZe as `type nat = | O | S(nat)`. Two destructors are introduced, `Destr_O` and `Destr_S`. `Destr_O R a b c` represents the ML term **match** a **with** $\mid 0 \Rightarrow b \mid \_ \Rightarrow c$ and `Destr_S R a b c` represent the ML term **match** a **with** $\mid S(n) \Rightarrow b(n) \mid \_ \Rightarrow c$. Each destructor is defined by two rewrite rules, following the definition given in 4.2.

The produced Dedukti code for `nat` is:

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

def Destr_0 : R : Type -> nat -> R -> R -> R.
[R,b,c]     Destr_0 R 0     b c --> b
[R,n,b,c] Destr_0 R (S n) b c --> c.

def Destr_S : R : Type -> nat -> (nat -> R) -> R -> R.
[R,b,c]     Destr_S R 0     b c --> c
[R,n,b,c] Destr_S R (S n) b c --> b n.
```

5 Compilation of recursive functions

Recursion is a powerful but subtle feature in FoCaLiZe. When certifying recursive functions, we reach the limits of Zenon and Zenon Modulo because the rewrite rules corresponding to recursive definitions have to be used with parsimony otherwise Zenon Modulo could diverge.

In FoCaLiZe backend to Coq, termination of recursive functions is achieved thanks to the high-level `Function` mechanism [12]. This mechanism is not available in Dedukti and it does not allow proof by reflection as it disables reduction of recursive functions.

Contrary to Coq, Dedukti does not require recursive functions to be proved terminating *a priori* but we can use functions whose termination is hard to prove (such as Goodstein sequence, whose termination is not provable in first-order arithmetic) or still unknown (such as the hailstone sequence whose termination is the Collatz conjecture).

As we did in a previous translation of a programming language in Dedukti [7], we express the semantics of FoCaLiZe by a non-terminating Dedukti signature.

5.1 Examples

As a starter, we will consider a few examples of idiomatic FoCaLiZe recursive functions in order to illustrate several possible styles allowed by FoCaLiZe. We want to treat these different styles uniformly. Furthermore, this set of examples highlight some important features that our translation of recursive functions will have to implement.

Factorial For better efficiency of the generated OCaml code, FoCaLiZe provides the type `int` for **integers** and not the type of natural numbers which would have a better inductive structure.

For this reason, when defining recursive functions on numbers in FoCaLiZe, we have to take care of negative values. Below, in the case of the factorial function, we choose to map any negative integer to the value 1.

```
let rec fact (n) = if n < 2 then 1 else n * fact (n - 1)
```

This example shows that it is not reasonable to limit recursion in FoCaLiZe to the case of recursive calls on syntactical subterms (such as the `Fixpoint` construct of Coq). Moreover, this illustrates a case where we can not inspect the type definition of the decreasing argument of the recursive function since the definition of the type `int` is not visible to Focalide (it is defined directly in Dedukti using the definition of Figure 2).

Equality of lists Our second example is about structural induction on the datatype of (polymorphic) lists. In a style typical of functional languages, we can define equality for lists by pattern-matching as in Figure 8.

```
type list ('a) = | Nil | Cons ('a, list ('a));;

let rec equal (l1, l2) =
  match l1 with
  | Nil -> (match l2 with
     | Nil -> true
     | Cons (_, _) -> false)
  | Cons (h1, t1) -> (match l2 with
     | Nil -> false
     | Cons (h2, t2) -> (h1 = h2) && equal (t1, t2));;
```

Fig. 8. Constructor-based equality of lists

In FoCaLiZe however, object-oriented mechanisms also invite us to a more abstract and axiomatic view of lists which we illustrate in Figure 9.

Like in the case of the factorial function, the recursive calls are not done on subterms but on results of other functions (`tail` here). The main advantage of this version is that we do not commit to a fixed representation of lists; there can be several implementations of lists sharing the same definition of equality. However, this comes at a price: it is not possible to prove the termination of `equal` without further assumptions because we can not prove that the tail of a list is strictly smaller than the list itself.

5.2 Very naive translation

For the rest of this section, we consider a function `f` of type $A \to B$ defined in FoCaLiZe by the recursive equation

```
let rec f (x) = g(f(h(x)), x)
```

In Dedukti, reduction under λ-abstraction is allowed so defining `f` by the rewrite rule[4]

[4] `x : A => t` is Dedukti syntax for the abstraction $\lambda x : A.t$

```
species List (A is Setoid) =
  inherit Setoid;
  signature nil : Self;
  signature cons : A -> Self -> Self;
  signature is_nil : Self -> bool;
  signature head : Self -> A;
  signature tail : Self -> Self;

  property surjective_pairing : all l : Self,
    ~ (is_nil(l)) <-> l = (cons(head(l), tail(l)));

  property head_proj : all a : A, all l : Self,
    head (cons (a, l)) = a;

  property tail_proj : all a : A, all l : Self,
    tail (cons (a, l)) = l;

  property is_nil_nil : is_nil(nil);

  let rec equal (l1, l2) = (is_nil (l1) && is_nil (l2)) ||
    (~~ is_nil(l1) && ~~ is_nil(l2) &&
     A!equal(head(l1), head(l2)) && equal(tail(l1), tail(l2)));
 end;;
```

Fig. 9. Projection-based equality of lists

```
[ ] f --> x : A => g (f (h x)) x.
```

leads to a diverging rewrite system and no proof of statement involving `f` can be checked in finite time. Hence we cannot define `f` by rewriting `f` itself; we need to allow reduction of `f` only when applied to arguments. However, the rewrite rule

```
[x] f x --> g (f (h x)) x.
```

is not better because `f (h x)` is an instance of the pattern `f x`.

5.3 Call-by-value application combinator

What makes recursive definitions (sometimes) terminate in FoCaLiZe is the use of the **call-by-value** semantics. The idea is that we have to reduce any argument of `f` to a value before unfolding the recursive definition.

A solution is to define a combinator `CBV` of type `A : Type -> B : Type -> (A -> B) -> A -> B` which acts as application on values but does not reduce on most non-values. Its definition is extended when new datatypes are introduced.

Here is the definition of `CBV` when A is the type `int`:

```
[m,n,B,f] CBV int B f (make m n) --> f (make m n).
```

For algebraic datatypes, `CBV` can be defined by giving a rewrite rule for each constructor. Here is the definition for the algebraic type `nat`:

```
[B,f] CBV nat B f O --> f O.
[B,f,n] CBV nat B f (S n) --> f (S n).
```

Thanks to this combinator, we can translate the definition of `f` by the following scheme:

```
[x] f x --> g (CBV A B f (h x)) x.
```

This rewrite system does not trivially diverge as before because the term `f` in the right-hand side is unapplied so it does not match the pattern `f x`.

We get the following reduction behaviour: `f` alone does not reduce, `f v` (where `v` is a value) is fully reduced, and `f x` (where `x` is a variable or a non-value term) is unfolded once.

We do not need to insert a `CBV` combinator at each application node in the translation (which would be linear but very verbose so it would impact Zenon Modulo and Dedukti) but only at recursive calls, this can be done globally by defining the fixpoint combinator `Fix` of type `A : Type -> B : Type -> ((A -> B) -> (A -> B)) -> A -> B` defined by the following rewrite rule:

```
[A, B, F, x] Fix A B F x --> CBV A B (F (Fix A B F)) x.
```

The local recursive definition **let rec** $f\ x\ :=\ t$ **in** u (where f has type $A \to B$) is translated by $u'\{f\backslash\texttt{Fix}\ A'\ B'\ (f\texttt{=>}x\texttt{=>}\ t')\}$ where u', A', B', and t' are the respective translations of u, A, B, and t.

Theorem 2 (Semantics preservation). *For any well-typed FoCaLiZe term t, if t evaluates to v with respect to FoCaLiZe semantics, then its translation to Dedukti normalizes to the translation of v.*

Proof (sketch). By induction on the evaluation on the FoCaLiZe side, reduction of pattern-matching relies on Theorem 1, other cases are trivial.

5.4 Efficiency and limitations

The size of the code produced by Focalide is linear wrt. the input, the operational semantics of FoCaLiZe is preserved and each reduction step in the input language corresponds to a bounded number of rewriting steps in Dedukti, so the execution time for the translated program is only increased by a linear factor.

Our treatment of recursive definitions generalizes directly to mutual recursion but we have not implemented this generalization.

Moreover, the understanding of datatypes by Zenon Modulo is still incomplete; it is able to perform computation using the rewrite rules defining destructors but it is not yet able to reason about datatypes by induction or even case distinction; nor is it able to prove injectivity and distinctness of constructors. These properties still need to be proved directly in Dedukti until Focalide is able to automatically generate them from the datatype definition.

5.5 Termination

The rewrite system defining `Fix` is not terminating and this is intended since we want to be able to reason modulo evaluation of any program, including non-terminating ones.

Moreover, the `CBV` combinator is only an approximation of the call-by-value strategy which is intentionally incomplete for efficiency reasons. In the pathological case where the function `h` reduces to a term starting with a constructor, the rewrite rule

```
[x] f x --> g (CBV A B f (h x)) x.
```

is still diverging, even if the original code was terminating with respect to the call-by-value semantics.

6 Experimental results

We have evaluated Focalide by running it on different available FoCaLiZe developments. When proofs required features which are not yet implemented in Focalide, we commented the problematic lines and ran both backends on the same input files; the coverage column of Figure 10 indicates the percentage of remaining lines.

FoCaLiZe ships with three libraries: the standard library (stdlib) which defines a hierarchy of species for setoids, cartesian products, disjoint unions, orderings and lattices, the external library (extlib) which defines mathematical structures (algebraic structures and polynomials) and the user contributions (contribs) which are a set of concrete applications. Unfortunately, none of these library uses pattern-matching and recursion extensively so the fact that Focalide gives comparable or better results than the old backend is reassuring but does not tell much about the validity of our approach.

The other developments are more interesting in this respect; they consist of a test suite for termination proofs of recursive functions (term-proof), a pedagogical example of FoCaLiZe features with several examples of functions defined by pattern-matching (ejcp) and a specification of Java-like iterators together with an implementation of iterators by lists using both recursion and pattern-matching (iterators).

The results[5], shown in Figure 10 and Figure 11, show that on FoCaLiZe problems the user gets a good speed-up by using Zenon Modulo and Dedukti instead of Zenon and Coq. Proof-checking is way faster because Dedukti is a mere type-checker which features almost no inference whereas FoCaLiZe asks Coq to infer type arguments of polymorphic functions; this also explain why generated Dedukti files are bigger than the corresponding Coq files. Moreover, each time Coq checks a file coming from FoCaLiZe, it has to load a significant part of its standard library which often takes the majority of the checking time (about a second per file). In the end, finding a proof and checking it is usually faster when using Focalide.

Library	FoCaLiZe	Coverage	Coq	Dedukti
stdlib	163335	99.42%	1314934	4814011
extlib	158697	100%	162499	283939
contribs	126803	99.54%	966197	2557024
term-proof	24958	99.62%	227136	247559
ejcp	13979	95.16%	28095	239881
iterators	80312	88.33%	414282	972051

Fig. 10. Size (in bytes) comparison of Focalide with the old backend on available FoCaLiZe developments

Library	Zenon	ZMod	Coq	Dedukti	Zenon + Coq	ZMod + Dedukti
stdlib	11.73	32.87	17.41	1.46	29.14	34.33
extlib	9.48	26.50	19.45	1.64	28.93	28.14
contribs	5.38	9.96	26.92	1.17	32.30	11.13
term-proof	1.10	0.55	24.54	0.02	25.64	0.57
ejcp	0.44	0.86	11.13	0.06	11.57	0.92
iterators	2.58	3.85	6.59	0.27	9.17	4.12

Fig. 11. Time (in seconds) comparison of Focalide with the old backend on available FoCaLiZe developments

These files have been developed prior to Focalide so they do not yet benefit from Deduction modulo as much as they could. The Coq backend going through Zenon is not very efficient on proofs requiring computation because all reduction steps are registered as proof steps in Zenon leading to huge proofs which take a lot of time for Zenon to find and for Coq to check. For example, if we define a polymorphic datatype `type wrap ('a) = | Wrap ('a)`, we can define the isomorphism `f : 'a -> wrap('a)` by `let f (x) = Wrap(x)` and its inverse `g : wrap('a) -> 'a` by `let g(y) = match y with | Wrap (x) -> x`. The time taken for our tools to deal with the proof of $(\mathtt{g} \circ \mathtt{f})^n(x) = x$ for n from 10 to 19 is given in Figure 12; as we can see, the Coq backend becomes quickly unusable whereas Deduction modulo is so fast that it is even hard to measure it.

7 Related work

The closest related work is a translation from Coq to Dedukti and compilation techniques from ML to enriched λ-calculi.

– In the context of Coqine, a translator of a fragment of Coq kernel to Dedukti, Assaf [1] has proposed several techniques to compile recursive functions and pattern-matching in Dedukti. Pattern-matching is limited in Coq kernel to flat patterns so it is possible to define a single `match` symbol for each inductive type, which simplifies greatly the compilation of pattern-matching to Dedukti and avoids the use of dynamic error handling.

[5] The files can be obtained from `http://deducteam.inria.fr/focalide`

Value of n	Zenon	Coq	Zenon Modulo	Dedukti
10	31.48	4.63	0.04	0.00
11	63.05	11.04	0.04	0.00
12	99.55	7.55	0.05	0.00
13	197.80	10.97	0.04	0.00
14	348.87	1020.67	0.04	0.00
15	492.72	1087.13	0.04	0.00
16	724.46	$>$ 2h	0.04	0.00
17	1111.10	1433.76	0.04	0.00
18	1589.10	$>$ 2h	0.07	0.00
19	2310.48	$>$ 2h	0.04	0.00

Fig. 12. Time comparison (in seconds) for computation-based proofs

However, it does not seem possible to define a single `fix` symbol without breaking strong normalization of the rewrite system so, as in our work, each fixpoint has to be named and recursive unfolding has to be limited to expressions starting with a constructor. Assaf distinguishes two ways to achieve this; we can either wrap each constructor as proposed in [3] or use a combinator similar to `CBV` (called a filter function in [1]). Because of dependent typing, function arguments have to be duplicated when using the latter solution so it is unclear which solution (wrapping constructors or duplicating arguments) is the best in the context of Coqine. In our case, the input type system does not feature dependent types so this duplication of argument is unnecessary.

The main difference between our encoding of recursion and the one implemented in Coqine is that we define the `CBV` operator by ad-hoc polymorphism whereas Coqine filter functions are unrelated to each other. Thanks to ad-hoc polymorphism, we can deal with recursion on abstract types such as the example of Figure 9.

– The semantics of functional languages often rely on λ-calculus. Pattern-matching is a common feature in these languages so proving the correctness of a compiler for a functional language usually requires to define a translation function from pattern-matching to λ-calculus. This has been achieved by enriching the λ-calculus with simple forms of pattern-matching. These enriched λ-calculi are then used as intermediate compilation languages between the rich functional language and the low-level λ-calculus.

  - In [21], Peyton-Jones and Wadler extend the λ-calculus with an abstraction over pattern and internalize the list of patterns using a $[]$ operator. Matching failure is represented by the constant **FAIL** which is left-neutral for $|$ and non-exhaustiveness is represented by the constant **ERROR**. We avoid the introduction of constants **FAIL** and $[]$ for tracking matching failure and so we avoid the appearance of some alien terms such as **FAIL**$+2$. In our work, failure is replaced by the default behaviour of destructors. However, we still rely on a dynamic error mechanism to test exhaustiveness of pattern coverage whereas this property can be checked statically and even reduced to type-checking [16].

- In [17], Oostrom, Klop, and Vrijer generalize the enriched λ-calculus of Peyton-Jones and Wadler; they define another extension of the λ-calculus, the λ-calculus with patterns, generalizing the shape of λ-terms allowed to build abstractions from variables to terms verifying the *Rigid Pattern Condition*. However, they restrict their attention to uniform patterns, in the sense that the order of the branches of pattern-matching should not matter, which we find too restrictive in the context of the compilation of functional languages in general and FoCaLiZe in particular.
- More recently, Kahl introduced [15] the Pattern-Matching Calculus, focusing on the notion of matchings (patterns, possibly fed with arguments) constituting a grammatical class distinct from terms. Like Peyton-Jones and Wadler, $[]$ and **FAIL** are part of the calculus but matching success is easier to detect and alien terms are harder to produce.

Following [21], we could add optimization steps to replace destructors by eliminators (called case-expressions in [21]) which are considered more efficient and would limit the use of dynamic errors, in particular in the common case where, like in our first example of equality over lists, the only pattern-matchings used in the source file are eliminators. However, we believe that keeping destructors is the best choice when the last pattern of the matching is a variable or a wildcard, in which case we do not emit any **ERROR**.
In our context, the efficiency of the produced Dedukti code is not fundamental because definitions using pattern-matching in FoCaLiZe are usually simple. The main reasons for this is that complex pattern-matchings are hard to specify and that Zenon support for pattern-matching is limited. Hence we avoid complex compilation techniques such as decision trees [19]. By doing so, we obtain a light translation, close to the compilation to Coq and predictable by the programmer.

A lot of work has also been done to compile programs (especially functional recursive definitions [14, 13, 18]) to rewrite systems. The focus has often been on termination preserving translations to prove termination of recursive functions using termination checkers for term rewrite systems. However, these translations do not try to preserve the semantics of the programs so they can hardly be adapted for handling translations of correctness proofs.

8 Conclusion

We have extended the compiler of FoCaLiZe to a new output language: Dedukti. Contrary to previously existing FoCaLiZe outputs OCaml and Coq, Dedukti is not a functional programming language but an extension of a dependently-typed λ-calculus with rewriting so pattern-matching and recursion are not trivial to compile to Dedukti.

However, we have shown that ML pattern-matching can easily and efficiently be translated to Dedukti using destructors. We plan to further optimize the compilation of pattern-matching, in particular to limit the use of dynamic error handling. For recursion, however, efficiency comes at a cost in term of normalization because we can not fully enforce the use of the call-by-value strategy without loosing linearity. Instead of trying to fix our encoding we would like to delegate termination checking to external tools; this is a

Our approach is general enough to be adapted to other functional languages because FoCaLiZe language for implementing functions is an ML language without specific features. FoCaLiZe originality comes from its object-oriented mechanisms which are invisible to Focalide because they are statically resolved in an earlier compilation step. Moreover, it can also easily be adapted to other rewriting formalisms, especially untyped and polymorphic rewrite engines because features specific to Dedukti (such as higher-order rewriting or dependent typing) are not used.

We have tested Focalide on existing FoCaLiZe libraries and have found it a decent alternative to the Coq backend whose adoption can enhance the usability of FoCaLiZe to a new class of proofs based on computation.

Focalide can also be a bridge of interoperability with other proof systems, Dedukti is used as the target language of a large variety of systems in the hope of exchanging proofs; we want to experiment the import and export of proofs between logical systems by using FoCaLiZe and Focalide as an interoperability platform.

References

1. Ali Assaf. *A Framework for Defining Computational Higher-Order Logics.* PhD thesis, École Polytechnique, 2015.
2. Ali Assaf and Guillaume Burel. Translating HOL to Dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, Proceedings Fourth Workshop on *Proof eXchange for Theorem Proving*, volume 186 of *EPTCS*, pages 74–88, Berlin, Germany, August 2015.
3. Mathieu Boespflug and Guillaume Burel. CoqInE : Translating the calculus of inductive constructions into the $\lambda\Pi$-calculus modulo. In *Proceedings of the Second International Workshop on Proof Exchange for Theorem Proving*, 2012.
4. Richard Bonichon, David Delahaye, and Damien Doligez. Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007*, volume 4790 of *LNCS/LNAI*, pages 151–165. Springer, 2007.
5. Guillaume Burel. A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP 2013. 3rd International Workshop on Proof Exchange for Theorem Proving*, volume 14 of *EasyChair Proceedings in Computing*, pages 43–57, Lake Placid, USA, June 2013.
6. Guillaume Bury, David Delahaye, Damien Doligez, Pierre Halmagrand, and Olivier 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.

7. Raphaël Cauderlier and Catherine Dubois. Objects and subtyping in the $\lambda\Pi$-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.
8. Raphaël Cauderlier and Pierre Halmagrand. Checking Zenon Modulo Proofs in Dedukti. In Cezary Kaliszyk and Andrei Paskevich, editors, Proceedings 4th Workshop on *Proof eXchange for Theorem Proving*, volume 186 of *EPTCS*, pages 57–73, Berlin, Germany, August 2015.
9. Denis Cousineau and Gilles 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.
10. David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand, and Olivier Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In *LPAR*, volume 8312 of *LNCS/ARCoSS*, pages 274–290. Springer, dec 2013.
11. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem Proving Modulo. *Journal of Automated Reasoning (JAR)*, 31, 2003.
12. Catherine Dubois and François Pessaux. Termination Proofs for Recursive Functions in FoCaLiZe. In *Trends in Functional Programming 15th International Symposium (TFP 2015), Revised Selected Papers*, LNCS, Sophia-Antipolis, France, June 2015. To appear.
13. Thomas Genet, Barbara Kordy, and Amaury Vansyngel. Vers un outil de vérification formelle légère pour OCaml. In Frédéric Dadeau and Pascale Le Gall, editors, *AFADL 2015*, pages 28–33, Bordeaux, France, May 2015.
14. Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7:1–7:39, February 2011.
15. Wolfram Kahl. Basic Pattern Matching Calculi: A Fresh View on Matching Failure. In Yukiyoshi Kameyama and Peter Stuckey, editors, *Functional and Logic Programming, Proceedings of FLOPS 2004*, volume 2998 of *LNCS*, pages 276–290. Springer, 2004.
16. Delia Kesner, Laurence Puel, and Val Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 1996.
17. Jan Willem Klop, Vincent van Oostrom, and Roel de Vrijer. Lambda calculus with patterns. *Theoretical Computer Science*, 398(1–3):16–31, 2008. Calculi, Types and Applications: Essays in honour of M. Coppo, M. Dezani-Ciancaglini and S. Ronchi Della Rocca.
18. Salvador Lucas and Ricardo Peña. Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In *LOPSTR'08*, pages 43–57, Valencia, Spain, July 2008.
19. Luc Maranget. Compiling Pattern Matching to Good Decision Trees. In *Workshop on the Language ML*. ACM Press, September 2008.
20. François Pessaux. FoCaLiZe: Inside an F-IDE. In Catherine Dubois, Dimitra Giannakopoulou, and Dominique Méry, editors, *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, volume 149 of *EPTCS*, pages 64–78, 2014.
21. Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.
22. Ronan Saillard. *Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice*. PhD thesis, MINES Paritech, 2015.