ML Pattern-Matching, Recursion, and Rewriting: From FoCaLiZe to Dedukti

Raphaël Cauderlier, Catherine Dubois

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 - Cedric and Samovar

Abstract. The programming environment FoCaLiZe allows the user to specify, implement, and prove programs with the help of the theorem prover Zenon. In the actual version, those proofs are verified by Coq. In this paper we propose to extend the FoCaLiZe compiler by a backend to the Dedukti language in order to benefit from Zenon Modulo, an extension of Zenon for Deduction modulo. By doing so, FoCaLiZe can benefit from a technique for finding and verifying proofs more quickly. The paper focuses mainly on the process that overcomes the lack of local pattern-matching and recursive definitions in Dedukti.

1 Introduction

FoCaLiZe [15] is an environment for certified programming which 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 [3] which takes Coq problems as input and outputs proofs in Coq format for independent checking. Zenon has recently been improved to handle Deduction modulo [9], an efficient proof-search technique. However, the Deduction modulo version of Zenon, Zenon Modulo, outputs proofs for the Dedukti proof checker [17] instead of Coq [6].

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 translated to Dedukti [8, 1, 2].

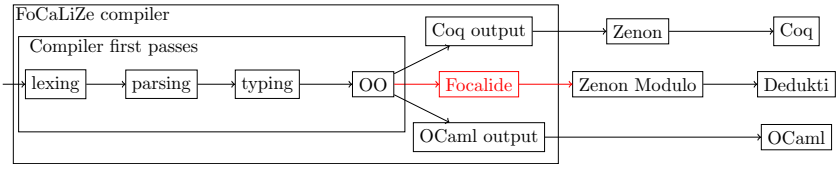

Fig. 1. FoCaLiZe Compilation Scheme

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

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 local pattern-matching and recursive definitions.

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. In Section 4, the backend to Dedukti is evaluated on benchmarks. Section 5 discusses related work and Section 6 concludes the paper by pointing some future work.

2 Presentation of the tools

2.1 Dedukti

Dedukti [17] 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 [8, 2, 1]. It is commonly used to check proofs coming from the Deduction modulo provers Iprover Modulo [4] and Zenon Modulo [6].

A Dedukti file consists of an interleaving of declarations (such as `0 : nat`) and rewrite rules (such as `[n] plus 0 n --> 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 following proof of $2+2 = 4$ (theorem `two_plus_two_is_four` below):

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

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

2.2 Zenon Modulo

Zenon [3] is a first-order theorem prover based on the tableaux method. It is able to produce proof terms which can be checked independently by Coq.

Zenon Modulo [9] is an extension of Zenon for Deduction modulo, an extension of first-order logic distinguishing computation from reasoning. Computation is defined by a rewrite system, it is part of the theory. Reasoning is defined by a usual deduction system (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 [15] and FoCaLiZe reference manual). More precisely we adress here the focalizec compiler that 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. A FoCaLiZe specification is 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. Statements belong to first-order logic. 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 [3].

FoCaLiZe developments are organized in program units called species. Species in FoCaLiZe define types together with functions and properties applying to them. At the beginning of a development, types are usually abstract. A species may inherit one or several species and specify a function or a property or implement them by respectively providing a definition or a proof. 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 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, inheritance and late binding are resolved at compile-time (OO on Figure 1), relying on a dependency calculus described in [15]. The process for compiling proofs towards Coq is achieved in two steps. First the statement is compiled with a hole for the proof script. The goal and the context are 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 brievly describe the

input language we have to consider and the main principles of the translation and then focus on the compilation of pattern-matching and recursive functions. A more detailed and formal description can be found in [7].

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. 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. 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. In particular patterns are linear and tried in the order they are written. A logical formula is a regular first order formula where an atomic formula is a Boolean expression. Its free variables correspond to the functions and constants introduced in the program.

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 [6].

3.3 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.

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

Each FoCaLiZe expression is translated into an expression 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. It 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. Serialization and flattening terminate and are linear; moreover they preserve the semantics of pattern-matching.

3.4 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 [10]. This mechanism is not available in Dedukti. Contrary to Coq, Dedukti does not require recursive functions to be proved terminating *a priori*. We can postpone termination proofs.

As we did in a previous translation of a programming language in Dedukti [5], we express the semantics of FoCaLiZe by a non-terminating Dedukti signature.

In FoCaLiZe, recursive functions can be defined by pattern-matching on algebraic types but also with regular conditional expressions. For example, if lists are not defined but axiomatized, we might define list equality as follows:

```
let rec list_equal (l1, l2) = (is_nil (l1) && is_nil (l2)) ||
     (~ is_nil(l1) && ~ is_nil(l2) && head(l1) = head(l2) &&
        list_equal(tail(l1), tail(l2)))
```

In Dedukti, defining a recursive function `f` by a rewrite rule of the form `[x] f x --> g (f (h x)) x.` is not a viable option because it breaks termination and no proof of statement involving `f` can be checked in finite time.

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.

For efficiency reasons, we approximate the semantics by only checking that the argument starts with a constructor. This is done by defining a combinator `CBV` of type `A:Type -> B:Type -> (A -> B) -> A -> B` which acts as application when its last argument start with a constructor but does not reduce otherwise. Its definition is extended when new datatypes are introduced by giving a rewrite rule for each constructor. Here is the definition for the algebraic type `nat` whose constructors are `O` and `S`:

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

Local recursion is then defined by introducing the fixpoint combinator `Fix` of type `A:Type -> B:Type -> ((A -> B) -> (A -> B)) -> A -> B` defined by the rewrite rule `[A, B, F, x] Fix A B F x --> CBV A B (F (Fix A B F)) x.` This does not trivially diverge as before because the term `Fix` in the right-hand side is only partially applied so it does not match the pattern `Fix A B F x`.

If `f` is a FoCaLiZe recursive function, 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.

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.

4 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 2 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 libraries uses pattern-matching and recursion extensively. 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 a list implementation of iterators using both recursion and pattern-matching.

Results[4] in Figures 2 and 3, 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.

[4] The files can be obtained from `http://deducteam.inria.fr/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. 2. Size (in bytes) comparison of Focalide with the Coq backend

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. 3. Time (in seconds) comparison of Focalide with the Coq backend

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 4; as we can see, the Coq backend becomes quickly unusable whereas Deduction modulo is so fast that it is even hard to measure it.

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

5 Related work

The closest related work is a translation from a fragment of Coq kernel to Dedukti [1]. 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. To handle recursion, filter functions playing the role of our `CBV` combinator are proposed. Because of dependent typing, they need to duplicate their arguments. Moreover, we define the `CBV` operator by ad-hock polymorphism whereas filter functions are unrelated to each other.

Compilation techniques for pattern-matching to enriched λ-calculi have been proposed, see e.g. [16, 13, 12]. We differ mainly in the treatment of matching failure.

A lot of work has also been done to compile programs (especially functional recursive definitions [11, 14]) 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 preserve the semantics of the programs so they can hardly be adapted for handling translations of correctness proofs.

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

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.

As 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.

Acknowledgements This work has been partially supported by the BWare project (ANR-12-INSE-0010) funded by the INS programme of the French National Research Agency (ANR).

References

1. Assaf, A.: A Framework for Defining Computational Higher-Order Logics. Ph.D. thesis, École Polytechnique (2015)
2. Assaf, A., Burel, G.: Translating HOL to Dedukti. In: Kaliszyk, C., Paskevich, A. (eds.) Proceedings Fourth Workshop on Proof eXchange for Theorem Proving. EPTCS, vol. 186, pp. 74–88. Berlin, Germany (2015)
3. Bonichon, R., Delahaye, D., Doligez, D.: Zenon: An Extensible Automated Theorem Prover Producing Checkable Proofs. In: Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007. LNCS/LNAI, vol. 4790, pp. 151–165. Springer (2007)
4. Burel, G.: A Shallow Embedding of Resolution and Superposition Proofs into the $\lambda\Pi$-Calculus Modulo. In: Blanchette, J.C., Urban, J. (eds.) PxTP 2013. 3rd International Workshop on Proof Exchange for Theorem Proving. EasyChair Proceedings in Computing, vol. 14, pp. 43–57. Lake Placid, USA (2013)
5. Cauderlier, R., Dubois, C.: 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), Schloss Dagstuhl, Paris (2014)
6. Cauderlier, R., Halmagrand, P.: Checking Zenon Modulo Proofs in Dedukti. In: Kaliszyk, C., Paskevich, A. (eds.) Proceedings 4th Workshop on Proof eXchange for Theorem Proving. EPTCS, vol. 186, pp. 57–73. Berlin, Germany (2015)
7. Cauderlier, R.: Object-Oriented Mechanisms for Interoperability between Proof Systems. Ph.D. thesis, Conservatoire National des Arts et Métiers, Paris (draft)
8. Cousineau, D., Dowek, G.: Embedding pure type systems in the lambda-pi-calculus modulo. In: Rocca, S.R.D. (ed.) Typed Lambda Calculi and Applications. pp. 102–117. Springer-Verlag (2007)
9. Delahaye, D., Doligez, D., Gilbert, F., Halmagrand, P., Hermant, O.: Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In: LPAR. LNCS/ARCoSS, vol. 8312, pp. 274–290. Springer (2013)
10. Dubois, C., Pessaux, F.: Termination proofs for recursive functions in focalize. In: Serrano, M., Hage, J. (eds.) Trends in Functional Programming - 16th International Symposium, TFP 2015, Sophia Antipolis, France, June 3-5, 2015. Revised Selected Papers. Lecture Notes in Computer Science, vol. 9547, pp. 136–156. Springer (2016)
11. Giesl, J., Raffelsieper, M., Schneider-Kamp, P., Swiderski, S., Thiemann, R.: Automated termination proofs for haskell by term rewriting. ACM Trans. Program. Lang. Syst. 33(2), 7:1–7:39 (2011)
12. Kahl, W.: Basic Pattern Matching Calculi: A Fresh View on Matching Failure. In: Kameyama, Y., Stuckey, P. (eds.) Functional and Logic Programming, Proceedings of FLOPS 2004. LNCS, vol. 2998, pp. 276–290. Springer (2004)

13. Klop, J.W., van Oostrom, V., de Vrijer, R.: 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
14. Lucas, S., Peña, R.: Rewriting Techniques for Analysing Termination and Complexity Bounds of SAFE Programs. In: LOPSTR'08. pp. 43–57 (2008)
15. Pessaux, F.: FoCaLiZe: Inside an F-IDE. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France. EPTCS, vol. 149, pp. 64–78 (2014)
16. Peyton Jones, S.L.: The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1987)
17. Saillard, R.: Type Checking in the Lambda-Pi-Calculus Modulo: Theory and Practice. Ph.D. thesis, MINES Paritech (2015)