Termination Proofs for Recursive Functions in FoCaLiZe

Extended Abstract

Catherine Dubois[1] and François Pessaux[2]

[1] ENSIIE – CEDRIC, catherine.dubois@ensiie.fr
[2] ENSTA ParisTech – U2IS, francois.pessaux@ensta.fr

Abstract. FoCaLiZe is a development environment allowing the writing of specifications, implementations and correctness proofs. It generates both OCaml (executable) and Coq code (for verification needs). This paper extends the language and the compiler to handle termination proofs relying on well-founded relations or measures. We propose an approach where the user's burden is lighten as much as possible, leaving glue code to the compiler. Proofs are written in the usual way in FoCaLiZe, using the declarative proof language and the automatic theorem prover Zenon. When compiling to Coq we rely on the Coq construct `Function`.

Keywords: Formal proof, functional programming, FoCaLiZe, Coq, recursion, termination

1 Introduction

The FoCaLiZe environment [1] (formally FoCaL [8]) allows one to incrementally build programs or library components with a high level of confidence and quality. FoCaLiZe units may contain specifications, implementations and proofs that the implementations satisfy their specifications. These ones are first-order like formulas while implementations are given as a set of functions in a syntax close to OCaml's one. Proofs are done using hints to the automatic prover Zenon [2] [6] in a declarative style. Inheritance and parametrization allow the programmer to reuse specifications, implementations and proofs. FoCaLiZe units are translated into OCaml executable code and verified by the Coq proof assistant.

When specifying properties that the result of a function should follow, we assume that the function does compute a result. This hypothesis is trivial for functions such as identity or square; others may require the programmer to restrict their domain (e.g. division) or lead to extra proof obligations to show that the -recursive- function terminates. The problem of termination is known as undecidable, however it is tractable in many cases. We rely on classical techniques also used in PVS, Coq or Isabelle consisting in showing that the arguments of each recursive call in the function are strictly lower than the arguments of the initial call according to a measure or a well-founded ordering. Some tools try to automatically verify termination, they may fail with the answer *Don't know*. In

FoCaLiZe, we adopt for termination checking, a solution in line with the general proof discipline which consists in guiding Zenon in its proof search by giving some hints. So the programmer will indicate the well-founded relation or the measure the proof will use, the recursive argument and provide the proof that the argument is decreasing and the proof that the relation is a well-founded one when necessary. FoCaLiZe provides some helps and computes the statements of the required proofs. Furthermore we do want as much as possible to do the proofs with Zenon. However Zenon relies on first order and thus cannot cope with higher order statements, such as the ones that could be required to prove a certain relation is well-founded. However in many practical cases, the relation is a usual one (e.g. the usual order on natural numbers) or a lexicographic one obtained by combining some standard orders. So with a toolbox offering some standard orders, Zenon will be able to perform the required proofs. As said previously, all the proofs must be checked by Coq. In this context, the FoCaLiZe compiler translates a FoCaLiZe function into a Coq function that is required to be total. Thus when the recursion is structural, the function is translated into a Coq `Fixpoint`, and we benefit from the syntactic termination verification made by Coq. When the function is recursive but implements a general recursion, we translate it into a general recursive Coq function, using the `Function` construct [4]. The latter requires to determine the relation and asks for proofs that the argument is decreasing and when necessary a proof that the relation is well-founded. However it is not a syntactic translation because `Function` requires a relation on the tuple of the arguments of the function while the FoCaLiZe programmer has worked with the argument that really decreases and the well-founded relation on it only. So the compiler has to re-build the relation and the proofs required by `Function` and Coq from the ones provided by the FoCaLiZe programmer. Our approach strongly distinguishes these two views: the user/programmer view and the internal view. The compiler does the glue because it is not the burden of the programmer to fit to a scheme imposed by the certification process (Coq verification). Furthermore we believe that the user view allows to target different compilation schemes or certification environments (e.g. Isabelle).

The rest of the paper is organized as follows. Section 2 presents very briefly the FoCaLiZe environment, in particular its proof language. Section 3 is devoted to the definition of recursive functions whose termination proof requires a well-founded relation: both the user view and the internal view are illustrated on an example. Section 4 follows the same roadmap but for functions that can rely on a measure to prove termination[3]. Section 5 explains the current limitations and proposes some work in progress and perpectives. Many work exist on termination proof, so in Section 6 we discuss some related work.

2 Some FoCaLiZe Precisions

The basic brick of a FoCaLiZe development is the *species*, a grouping structure embedding *methods* which may be an internal datatype, *properties* (to be proved

[3] In the full paper, the compilation scheme will be explained in more details

later), *theorems*, *signatures* (declarations of functions to be defined later) and *definitions*. Once a species has all its signatures defined and properties proved, it can be submitted to an abstraction process turning it into an abstract datatype (a *collection* in the FoCaLiZe terminology) only showing its signatures and properties. Collections can then be used to parameterize species, bringing their own material.

The code generation model extensively uses a dependency calculus to handle late-binding and parametrization and λ-lift both types and methods to allow code consistency and sharing [13, 12]. The dependency calculus has been extended to take into account termination proofs which are not different from other proofs. Roughly speaking, a method m depending on the declaration (type) of a method n is said having a *decl-dependency* on n. In the definition of m, n then gets λ-lifted to circumvent its missing definition or final redefinition. If m depends on the definition of n, then it has a *def-dependency*. In this case, no λ-lifting is done, and the real definition of m is used in n. Dependencies on species parameters methods exist and can be considered as decl-dependencies.

Proofs are written in the FoCaLiZe Proof Language, providing a hierarchical decomposition into intermediate steps [11]. Each step states hypotheses, one goal and a proof of this latter. Each proof can either invoke Zenon to unfold definitions, use previous outer steps, properties, induction or can be a sub-proof.

```
theorem t : all a b c : bool, a -> (a -> b) -> (b -> c) -> c
proof =
  <1>1 assume a b c : bool,
       hypothesis h1: a, hypothesis h2: a -> b, hypothesis h3: b -> c,
       prove c
    <2>1 prove b by hypothesis h1, h2
    <2>2 qed by step <2>1 hypothesis h3
  <1>2 qed by step <1>1
```

The previous proof has two outer steps `<1>1` and `<1>2`. Step `<1>1` introduces hypotheses `h1`, `h2`, `h3` and the subgoal `c`. It is proved by a 2-step subproof. Step `<2>1` uses `h1` and `h2` to prove `b`. Step `<2>2` uses `<2>1` and `h3` in order to prove `c`. Step `<1>2` ends the whole proof.

A "backdoor" mechanism is however available, allowing to directly inline Coq scripts in proofs when Zenon does not suffice (e.g. higher-order) or to bind already existing Coq notions. This solution requires from the user a good knowledge of Coq, of the compiler transformations and makes the proofs not portable. It is mostly reserved for the standard library.

3 Well-Founded Relation

The essence of terminating recursion is that there are no infinite chains of nested recursive calls. This intuition is commonly mapped to the mathematical idea of a well-founded relation and we stick to this view which is also the Coq approach. More precisely, Coq uses accessibility to define well-founded relations. Accessibility describes those elements from which one cannot start an infinite descending chain. A relation on T is well-founded when all elements of type T are accessible.

In this section we illustrate our approach with the simple example of the function `div` that computes the quotient in the Euclidien division of two integers. The function is made total in order to only focus on its termination.

```
let rec div (a, b) =
  if a <= 0 || b <= 0 then 0
  else ( if (a < b) then 0 else 1 + div ((a - b), b) )
termination proof = order pos_int_order on a ... ;
```

3.1 User View

From the user's point of view, despite `div` has two arguments, only the first one `a` is of interest for termination. The well-founded relation used here, `pos_int_order` (of type `int` → `int` → `bool`) is the usual ordering on positive integers provided by the standard library. The well-foundedness obligation of this relation, is stated by `well_wrapper (pos_int_order)`, which relies on the FoCaLiZe standard library's definition of `well_wrapper` as:

`(fun f => well_founded (fun x y => Is_true (f x y)))`

and will be easily proved thanks to the theorem `pos_int_order_wf` of the library.

Notice that `well_founded` here is a Coq predicate, defined in the Coq standard library. This exemplifies that a FoCaLiZe specification can mix definitions and properties defined with the FoCaLiZe language together with Coq exported definitions and theorems (and also OCaml definitions but it is not the case in this work).

The function `div` having only one recursive call, the only decreasing proof obligation will be:

$\forall a : int, \forall b : int, \neg(a \leq 0 \vee b \leq 0) \rightarrow \ \neg(a < b) \rightarrow pos\_int\_order(a - b, a)$

where the conditions on the execution path leading to the recursion must be accumulated as hypotheses.

The termination proof consists in as many steps as there are recursive calls, each one proving the ordering (according to the relation) of the decreasing argument and the initial one, then one step proving that the termination relation is well-founded and an immutable concluding step telling to the compiler to assemble the previous steps, generate some stub code using a built-in Coq script to close the proof. The complete permination proof for `div` is given below:

```
let rec div (a, b) = ...
  termination proof = order pos_int_order on a
    <1>1 prove all a : int, all b : int,
      ~ (a <= 0 || b <= 0) ->
      ~ (a < b) -> pos_int_order (a - b, a)
      <2>1 assume  a : int, b : int,
           hypothesis H1: ~ (a <= 0 || b <= 0),
           hypothesis H2: ~ (a < b),
           prove pos_int_order (a - b, a)
           <3>1 prove b <= a
                by property int_not_lt_ge, int_ge_le_swap hypothesis H2
           <3>2 prove 0 <= a
                by property int_not_le_gt, int_ge_le_swap, int_gt_implies_ge
                hypothesis H1
           <3>3 prove 0 < b
                by property int_not_le_gt, int_gt_lt_swap hypothesis H1
           <3>4 prove (a - b) < a
```

```
            by step <3>1, <3>2, <3>3 property int_diff_lt
        <3>e qed by step <3>4, <3>2 definition of pos_int_order
   <2>e conclude
 <1>2 prove well_wrapper (pos_int_order)
    by  property pos_int_order_wf
<1>e qed coq proof {*wf_qed*} ;
```

To summarize, from the user's point of view, a recursive function whose termination relies on a well-founded relation is given by the four following points:

1. the relation,
2. the theorem stating that this relation is well-founded,
3. a theorem for each recursive call, stating that the arguments of the recursive calls are smaller than the initial arguments according to the given relation,
4. the recursive function with a termination proof of the shape:

```
<1>x proofs of right ordering for each recursive call
     (the same statements than corresponding theorems in point 3,
      even if it is possible to directly inline the proofs instead)
<1>x+1 proof of the relation being well-founded
     (same statement than in point 2, same remark than in steps <1>x)
<1>x+2 qed coq proof {*wf_qed*}
```

3.2 Internal view

From the compiler's point of view, the code generation is split in 4 steps:

1. **creation of the relation expected by** `Function`. This one takes two tuples with as many arguments as the user's recursive function has. It extracts the decreasing one from each tuple, and applies the user's relation on them.
2. **creation of the user-side termination theorem** containing the compiled proof of the user. This theorem only operates on the decreasing argument, hence uses the user's relation. This theorem is the conjunction of the decreasing obligations and the well-foundedness obligation.
3. **creation of the** `Function`**-side termination theorem**. This theorem operates on the tuple of arguments of the function and uses the generated relation. Roughly speaking, this theorem states the same property than the previous one, but operating on tuples and referring to the relation synthesized at step 1. This proof is fully done by the compiler, using the proof of well-foundedness of the user's relation and a Coq theorem (`wf_inverse_image`) stating that the reverse image of a well-founded relation by any function (here, tuple projectors) is a well-founded relation.
4. **creation of the recursive function definition using** `Function`. The Function body is obtained using the usual FoCaLiZe compilation scheme and the termination part is filled with the relation generated at step 1 and a final proof built with the theorem generated at the previous step.

4 Measure

We consider here the particular case when the termination relies on a *measure* - a function that returns a natural number - which must decrease at each recursive

call. We want to ease such termination proofs even if it would be possible for the user to use the previous approach, by constructing himself a well-founded relation from the measure. Precisely, the compiler does this job for him.

A measure has to be positive, which will be a proof obligation. However, the relation built from the measure being internalized, its well-foundness is no more asked to the user. From the user's point of view, the proof obligation for each recursive call must now show that the measure decreases on the argument of interest between each call. The compiler must build a well-founded relation from the measure operating on all the arguments of the function, to prove its well-foundness and build the final proof expected by Coq's construct `Function`.

Let us notice that `Function` natively supports a `measure` annotation that we do not use. In effect we think that only relying on a well-founded relation leaves the compilation scheme more open to other logical target languages.

We illustrate the approach with a function `mem` checking if an element belongs to a list. Although a structural argument could also be used, we chose to rely on the decreasing *length* of the list where the element is recursively searched. We first write the method `length` whose termination is simply structural on its argument `l`. Then we write the method `mem`, stating a termination proof using the measure `length` on its argument `l`, but we do not show the proof itself yet.

```
species Ex_mes (A is Basic_object) =
  let rec length (l : list(A)) =
    match l with
    | [] -> 0
    | h :: q ->  1 + length (q)
  termination proof = structural l ;
  let rec mem (l, x: A) =
    match l with
    | [] -> false
    | h :: q ->  h = x || mem (q, x)
  termination proof =
    measure length on l ... ;
end ;;
```

4.1 User View

From the user's point of view, despite `mem` has two arguments, only `l` is of interest for the termination. The measure being `length`, the first proof obligation is:

$\forall l : list(A), 0 \leq length(l)$

The method `mem` having only one recursive call, the only decreasing proof obligation is:

$\forall l : list(A), \forall q : list(A), \forall h : A, l = h :: q \rightarrow length(q) < length(l)$

where variables bound on the execution path leading to the recursion must be accumulated as hypotheses. Here, the recursion being in a pattern-matching case, `h` and `q` must be related to the matched value `l`. The core of the decreasing fact is the $<$ relation between the argument of the recursive call and the one in the current call.

For readability, instead of inlining the proofs of these obligations, the user will state two related properties or theorems before the function `mem` itself.

```
property length_pos : all l: list (A), 0 <= length (l) ;

theorem mes_decr : all l : list(A) , all q : list(A), all h : A,
  l = h :: q -> length (q) < length (l)
proof = ... ;
```

Now the termination proof consists in as many steps as there are recursive calls, each one proving the strict decreasing of the measure, then one step proving that the measure is positive and an immutable concluding step telling to the compiler to assemble the previous steps and generate some stub code to close the proof.

```
let rec mem (l, x: A) = ...
termination proof = measure length on l
  <1>1 prove all l : list(A) , all q : list(A), all h : A,
         l = h :: q -> length (q) < length (l)
         by property mes_decr
  <1>2 prove all l: list (A), 0 <= length (l)
       by property length_pos
  <1>e qed coq proof {*wf_qed*} ;
```

To summarize, from the user's point of view, a recursive function whose termination relies on a measure is given by the four following points:

1. The measure function returning a regular integer (which raises the issue that $<$ is well-founded on naturals, not integers).
2. The theorem stating that the measure is always positive or null.
3. A theorem for each recursive call, stating that the measure on the argument of interest decreases.
4. The recursive function with a termination proof of the shape:

```
<1>x proofs of decreasing for each recursive call
     (the same statements than corresponding theorems in point 3,
      even if it is possible to directly inline the proofs instead)
<1>x+1 proof of the measure being always positive or null
     (same statement than in point 2, same remark than in steps <1>x)
<1>x+2 qed coq proof {*wf_qed*}
```

4.2 Internal view

From the compiler's point of view, the code generation is split in 4 parts:

1. **creation of the relation expected by** `Function`. It takes two tuples with as many arguments as the user's recursive function has. It extracts the decreasing one from each tuple, say x and y. Its finally states that the measure on y is positive and that the measure on x is strictly lower than the measure on y. The first part of this definition is required by the proof done in the point 3.
2. **creation of the user-side termination theorem** containing the compiled proof of the user. This theorem only operates on the decreasing argument, hence uses the user's measure function. This theorem is the conjunction of the decreasing obligations and the measure positive obligation.
3. **creation of the** `Function`**-side termination theorem**. This theorem operates on the tuple of arguments of the function and uses the generated relation. Roughly speaking, this theorem states the same property than the previous one, but operating on tuples and referring to the relation generated at step 1. This proof is fully done by the compiler, using the user's proof that the measure is always positive and a Coq theorem stating that the usual order on positive integers is well-founded.

4. **creation of the recursive function definition using** `Function`. This step computes the `Function` body and the final proof built with the theorem generated at the previous step for the termination part.

The last two points are exactly the same than for a termination proof using a well-founded relation. This allows a code generation model as close as possible for both kinds of proofs.

5 Limitations and Perspectives

The termination proofs as described in this paper are already available in the FoCaLiZe distribution using the `--experimental` option. A "toolbox" containing some low-level theorems proved in Coq is also available to "manually" wrap a measure in a well-founded relation.

Termination proofs using lexicographic orders are currently under study. Again we want to stick to our approach that provides the user with some comfort for termination proofs. The compiler will have to generate itself the lexicographic order and its well-foundness proof from the user's orders and their own well-foundness proofs.

Some known limitations exist. Termination proofs being based on the Coq construct `Function`, only methods of species and toplevel functions are supported. Local recursive functions cannot be handled this way. Nested recursion is also not supported since the construct `Function` does also not. Mutual recursive functions cannot be compiled with the present scheme. Although encodings exist to deal with such functions, several issues already appear: what are the proof obligations to impose to the user, how strong will they be impacted by the encoding, how understandable will these obligations become, how to mix several termination proof schemes?

Aside the kind of termination proofs, it is not fully clear how to provide late-binding at a termination level. In FoCaLiZe, definitions can use only defined methods. The dependency calculus allows λ-lifting late-bound symbols. The termination proof of a recursive function is part of its definition, hence delaying the proof means delaying the function definition. The simplest solution would be to consider the function as a simple signature until its proof is provided. This would however delay other proofs depending on the definition of the function, despite the termination proof is not relevant for them. In effect, either the function terminates and other functions are not interested in its termination, or it does not terminate, and the complete logical model is possibly inconsistent.

6 Related Work

Our work is in line with those about the definition of recursive functions in theorem provers, and more precisely the proposals made to facilitate the definition and reasoning with general recursive functions. All these propositions allow some

separation of the computational and logical parts, as we do. As said previously, we would like to go a step further in this direction and defer a termination proof.

TFL [14], implemented for both HOL4 and Isabelle, allows the definition and reasoning about total recursive programs written in a purely functional manner. In this context, establishing the termination of a function requires the introduction of a well-founded relation (proved as such) and the proof that the arguments of the recursive calls decrease according to this relation.

Coq provides the `Function` package [4] that allows one to define recursive functions in a way close to TFL. It relies on previous work done on termination by Balaa and Bertot [3]. The main strength concerns induction principles automatically generated from the algorithmic definition of the function, e.g. functional induction. We decided to compile our - non structural - recursive functions to Coq functions defined with `Function` because of traceability. Except for the usual modifications du to the compilation of dependencies and the proof obligations part, the text of the FoCaLiZe recursive function and the Coq one are very close. Furthermore the fixpoint equation generated by `Function` and by the FoCaLiZe compiler for Zenon reasoning are again very close. Another compilation choice would have been to bypass `Function` and its limitations altogether and generate Coq definitions on top of the basic Coq `Wf` package that provides a well-founded induction principle. We could also have used Bertot and Komendantsky's approach [5] to general recursion. As said previously, the shapes of both definitions in Coq and FoCaLiZe would have been too different and furthermore it would have been more difficult to generate the proofs.

In [10], Krauss provides a way to define general well-founded recursion in Isabelle. It is based on principles close to those used by `Function` in Coq, and goes further in some directions (nested recursion, mutual recursion and partiality). The main strength of this work is the advances in the automation of termination proofs. It can prove automatically termination of a certain class of functions by searching for a suitable lexicographic combination of size measures [7]. The terminaison for another class is handled by using the Size-Change principle [9].

7 Conclusion

This work integrates in FoCaLiZe means to prove the termination of recursive functions that are not only structural. It brings the ability to state a well-founded relation or a measure and write the termination proof using the usual FoCaLiZe proofs shape: with a hierarchical structure and using the Zenon automated theorem prover to discharge the user. Proof obligations are indicated to the user by the compiler, which avoids tedious errors and the need to guess what proof obligations the compiler is expecting. The proofs done by the user are based on the usual termination proof obligations for a well-founded relation or a measure, and ask the user only to consider the decreasing argument of his function. This point of view is indeed the one always used for handmade proofs and it would be annoying to ask the user to cope with all the other arguments since they are of no interest for the termination.

Termination proofs can transparently involve late-bound methods (i.e. only declared methods, e.g. properties used in proof obligations or even the measure or the well-founded relation) thanks to the FoCaLiZe λ-lifting mechanism.
A more general compilation scheme seems required to solve the pending issues and have a more unified code generation model. But this scheme remains to be found. However, the current work is already an appreciable help for the user and a first step toward a more global problem. It already allowed to write some previously assumed termination proofs of the FoCaLiZe standard library.

Acknowledgements. We thank Renaud Rioboo for the useful discussions and case studies. Thanks to Julien Forest for the helpful discussions about `Function`. Lastly we thank William Bartlett for his work on a very first prototype.

References

1. http://focalize.inria.fr/.
2. http://zenon-prover.org/.
3. A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In *TPHOLs 2000, Portland, Oregon, USA, Proceedings*, volume 1869 of *LNCS*, pages 1–16. Springer, 2000.
4. G. Barthe, J. Forest, D. Pichardie, and V. Rusu. Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *LNCS*, pages 114–129, 2006.
5. Y. Bertot and V. Komendantsky. Fixed point semantics and partial recursion in coq. In *Proceedings of PPDP'2008, Valencia, Spain*, pages 89–96, 2008.
6. R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007*, volume 4790 of *LNCS*, pages 151–165. Springer, 2007.
7. L. Bulwahn, A. Krauss, and T. Nipkow. Finding lexicographic orders for termination proofs in Isabelle/HOL. In *TPHOLs 2007*, volume 4732 of *LNCS*, pages 38–53, 2007.
8. C. Dubois, T. Hardin, and V. Donzeau-Gouge. Building certified components within FOCAL. volume 5 of *Trends in Functional Programming*, pages 33–48, 2006.
9. A. Krauss. Certified size-change termination. In *Automated Deduction - CADE-21, Bremen, Germany, 2007, Proceedings*, volume 4603 of *LNCS*, pages 460–475. Springer, 2007.
10. A. Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reasoning*, 44(4):303–336, 2010.
11. L. Lamport. How to write a proof. Research report, Digital Equipment Corporation, 1993.
12. F. Pessaux. Focalize: Inside an F-IDE. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014*, pages 64–78.
13. V. Prevosto. *Conception et Implantation du langage FoC pour le développement de logiciels certifiés*. PhD thesis, Université Paris 6, sep 2003.
14. K. Slind. Another look at nested recursion. In *TPHOLs 2000, Portland, Oregon, USA, Proceedings*, volume 1869 of *LNCS*, pages 498–518. Springer, 2000.