Termination Proofs for Recursive Functions in FoCaLiZe

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 lightened 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] (formerly FoCaL [12]) 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] [7] 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- functions terminate. 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 relation. Some tools, e.g. Isabelle and Agda [10] [3], try to automatically find a convenient lexicographic order and verify termination. 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 write 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 that 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. Hence, 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 [5]. This 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. In order to be as general as possible, we rely on a compilation scheme that restricts `Function` to use a well-founded relation or a measure defined on the tuple of all the arguments of the recursive function. This general compilation process will ease future work, e.g. allowing the user to set a measure involving several arguments or to make several arguments decrease. Thus, the translation is not syntactic and 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 for targeting 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. Section 5 explains the current limitations and proposes some work in progress and perpectives. Many works exist on termination proof, so in Section 6, we discuss some related work.

2 An Overview of FoCaLiZe

FoCaLiZe[1] is a development environment providing a unique language to write properties, functions and proofs, allowing high-level programming constructs like inheritance, late-binding, and parametrization. It is the continuation of FoC [8] and FoCaL [18].

A FoCaLiZe development is compiled into an executable (or object to link) OCaml code and a Coq term. The OCaml code only contains the computational aspects of the development, while the Coq code is a complete model, also embedding the logical aspects (i.e. properties and proofs). During the compilation process, the logical model is sent to Coq that acts as an assessor (i.e. it checks the code issued by the FoCaLiZe compiler and Zenon).

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 later), *theorems*, *signatures* (declarations of functions to be defined later) or *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-lifts both types and methods to allow code consistency and sharing [18,17]. 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 [16]. 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.

As an example, the following proof has two outer steps `<1>1` and `<1>2`. The step `<1>1` introduces hypotheses `h1`, `h2`, `h3` and the sub-goal `c`. It is proved by a 2-steps sub-proof. The step `<2>1` uses `h1` and `h2` to prove `b`. The step `<2>2` uses `<2>1` and `h3` in order to prove `c`. The step `<1>2` ends the whole 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
```

During the compilation process, proofs are compiled and sent to Zenon which tries to find a proof and returns a Coq term. This proof term is then injected in the final generated Coq script which is sent to Coq. If Zenon fails finding a proof with the hints given by the user, then the compilation process fails. Because Zenon does not support higher-order, λ-lifting must be temporarily replaced by Coq `Hypothesis` and `Variable` in `Sections`. This requires the compiler to prepare a suitable Coq environment to host the term that Zenon will return. The compiler must also transmit the user's hints to Zenon. This leads to a slightly verbose generated code for proofs but this remains still readable.

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 Relations

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 the 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 Euclidean division of two positive integers. The function, whose definition is given below, 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:

```
   let pos_int_order (i1, i2)=(0 <=i2)&& (i1 < i2)
```

The well-foundedness obligation of this relation is stated by `is_well_founded pos_int_order`, which relies on the FoCaLiZe standard library's definition of `is_well_founded` as:

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

The well-foundedness obligation is easily proved thanks to the library theorem `pos_int_order_wf`. Notice that `well_founded` here is a Coq predicate, defined in the Coq standard library (see Fig. 1 for the Coq definition). 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).

```
Variable A : Type.
Variable R : A -> A -> Prop.
Inductive Acc (x: A) : Prop :=
     Acc_intro : (forall y:A, R y x -> Acc y) -> Acc x.
(* A relation is well-founded if every element is accessible. *)
Definition well_founded := forall a:A, Acc a.
```

Fig. 1. Coq definition of well_founded

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

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

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 termination proof for `div` is given in Fig. 2. The statements of proof obligations (here lines 3-5 and line 21) are indicated to the user by the compiler. The proof that the argument of the recursive call is smaller than the initial one is quite long because Zenon has no support for arithmetic. However, this could be improved by using Zenon Arith which is an extension of Zenon, to handle linear arithmetic [11].

```
 1  let rec div (a, b) = ...
 2    termination proof = order pos_int_order on a
 3      <1>1 prove all a : int, all b : int,
 4        ~ (a <= 0 || b <= 0) ->
 5        ~ (a < b) -> pos_int_order (a - b, a)
 6        <2>1 assume  a : int, b : int,
 7             hypothesis H1: ~ (a <= 0 || b <= 0),
 8             hypothesis H2: ~ (a < b),
 9             prove pos_int_order (a - b, a)
10             <3>1 prove b <= a
11                  by property int_not_lt_ge, int_ge_le_swap hypothesis H2
12             <3>2 prove 0 <= a
13                  by property int_not_le_gt, int_ge_le_swap, int_gt_implies_ge
14                  hypothesis H1
15             <3>3 prove 0 < b
16                  by property int_not_le_gt, int_gt_lt_swap hypothesis H1
17             <3>4 prove (a - b) < a
18                  by step <3>1, <3>2, <3>3 property int_diff_lt
19             <3>e qed by step <3>4, <3>2 definition of pos_int_order
20        <2>e conclude  (* = qed by all previous steps of this nesting level. *)
21      <1>2 prove is_well_founded (pos_int_order)
22         by  property pos_int_order_wf
23     <1>e qed coq proof {*wf_qed*} ;
```

Fig. 2. Termination Proof for `div`

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 (the order of the obligations does not matter):

```
<1>x proofs of decreasing for each recursive call arguments
     (the same statements as the 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 as 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 as 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, 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.

Figure 2 gives an overview of the structure of the compilation of a function. Grayed blocks are those generated by the compiler while white ones are the code of the user.

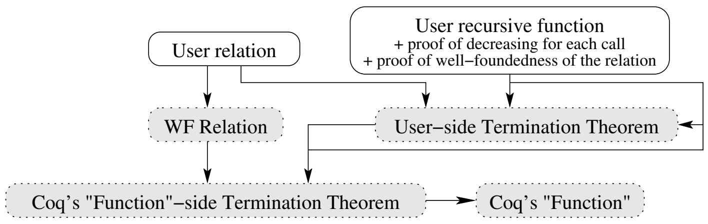

User relation
User recursive function
+ proof of decreasing for each call
+ proof of well–foundedness of the relation
WF Relation
User–side Termination Theorem
Coq's "Function"–side Termination Theorem
Coq's "Function"

Fig. 3. Global Structure of a Function with its Proof

3.2.1 Function

In this part, we briefly recall how the command Function works.

The Coq proof assistant provides the command Function [5] (see also the Coq manual reference) that allows the definition of both structural and well-founded recursive functions. When defining a non-structurally recursive function, the user is asked to provide a well-founded relation (or a measure function) and an argument (the decreasing one) and to show that the corresponding arguments in the recursive calls are smaller than the initial ones according to the well-founded relation (or the measure). Furthermore, the user has also to prove that the relation is indeed a wellfounded one. So a definition using this construct looks like a definition by pattern-matching in a functional language, annotated with a measure or a wellfounded relation and completed with some proof scripts discharging the termination proof obligations generated by Function. Note that the statements of the termination proof obligations do not appear in the definition. These ones are made explicit in a FoCaLiZe definition. Once the definition has been accepted by Coq, a set of definitions is automatically derived, in particular a fixpoint equation and an induction principle that follows the structure of the function.

3.2.2 Creation of the Relation Expected by Function

The expected relation takes two arguments __x and __y being tuples of all the function's arguments. To extract the one used for decreasing, a built-in projection on tuples is used. In Coq, tuples are encoded as pairs nested to the left: $(x, y, z) \equiv ((x, y), z)$. Projections are part of the low-level standard library of FoCaLiZe and internally known by the compiler which generates the right name from the number of components of the tuple and the position of the component to extract. Here the projection __tpl_firstprj2 extracts the first component of a 2-uple. and is defined as:

```
Definition __tpl_firstprj2 (__var_a : Set) (__var_b : Set)
 (x : (__var_a * __var_b)) : __var_a := __left __var_a __var_b x.
```

Once components are extracted, there remains to apply the user relation to them. The corresponding Coq definition is as follows (where _ stands for inferred arguments):

```
Definition div_wforder (__x __y : (int__t * int__t)) : Prop :=
  Is_true
    (pos_int_order (__tpl_firstprj2 _ _ __x) (__tpl_firstprj2 _ _ __y)).
```

3.2.3 Creation of the User-Side Termination Theorem

The user's termination proof must now be compiled and generated. Except the concluding step, all steps are compiled using the usual FoCaLiZe compilation process. Because Zenon does not handle higher-order, dependencies cannot be λ-lifted (c.f. 2). Instead, we enclose the proof in a Coq section, where dependencies lead to Variable and Hypothesis clauses. For the same reason, each step of proof is also embedded in a Section. In the generated proof below, line numbers and proof steps refer to the code given at Fig. 2.

```
Section Proof_of_div.
  Section __A_1.
    (* Step <1>1 line 3. *)
    Theorem __A_1_LEMMA :
      forall a : int__t, forall b : int__t,
        ~ Is_true (_bar__bar_ (_lt__eq_ a 0) (_lt__eq_ b 0)) ->
        ~ Is_true (_lt_ a b) -> Is_true (pos_int_order (_dash_ a b) a).
    (* ... Zenon proof term inlined here ... *)
    End __A_1.
  Section __A_2.
    (* Step <1>2 line 21. *)
    Theorem __A_2_LEMMA : ((is_well_founded _) pos_int_order).
    (* ... Zenon proof term inlined here ... *)
    End __A_2.
  (* Theorem's body. *)
  Theorem for_zenon_div :
    (forall a : int__t, forall b : int__t,
       ~ Is_true (_bar__bar_ (_lt__eq_ a 0) (_lt__eq_ b 0)) ->
       ~ Is_true (_lt_ a b) -> Is_true (pos_int_order (_dash_ a b) a))
      /\
      (well_founded (fun __a1 __a2 => Is_true (pos_int_order __a1 __a2))).
  Proof.
  generalize __A_2_LEMMA.
  generalize __A_1_LEMMA.
  unfold is_well_founded.
  intros.
  SplitandAssumption.
  Qed.
End Proof_of_div.
```

Once each step of the proof but the concluding one has been generated, the compiler detects that the concluding one refers to termination. It then generates and proves the theorem `for_zenon_div` stating the termination obligations as the user sees them: only dealing with the unique decreasing argument. This theorem's statement is the conjunction of all the decreasing lemmas and the well-foundedness of the user's relation introduced in 3.1. Since the user is expected to have proved the different parts of this conjunction in the previous steps, each lemma introduced by the steps is generalized, and a Coq tactic `SplitandAssumption` is used to automate splitting the goal then picking in the context to solve each sub-goal.

3.2.4 Creation of the `Function`-Side Termination Theorem

Once the user-side theorem is available, its counterpart as expected by `Function` must be generated as a separate theorem. This theorem states the same proof obligations, but dealing with the tuple of arguments of the recursive function. Hence it must use the relation automatically generated in 3.2.2 and the user-side termination theorem generated in 3.2.3 as shown in the following generated code.

```
Theorem div_termination :
  (forall a : int__t, forall b : int__t,
   ~ Is_true (_bar__bar_ (_lt__eq_ a 0) (_lt__eq_ b 0)) ->
   ~ Is_true (_lt_ a b) -> div_wforder ((_dash_ a b), b) (a, b))
   /\
   (well_founded (div_wforder)).
  Proof.
  unfold div_wforder;simpl.
  elim for_zenon_div.
```

```
intro __user_dec1.
intro __user_rem_dec_n_wf.
(* Separate decreasing obligations and well-foundedness. *)
split.
auto. (* For each rec call. *)
(* There, only remains the well-foundedness obligation. *)
set (R := (fun __a __b => Is_true (pos_int_order __a __b))).
change
  (well_founded (fun __c __d : (int__t * int__t)
    => R (__tpl_firstprj2 _ _ __c) (__tpl_firstprj2 _ _ __d))).
apply wf_inverse_image.
assumption.
Qed.
```

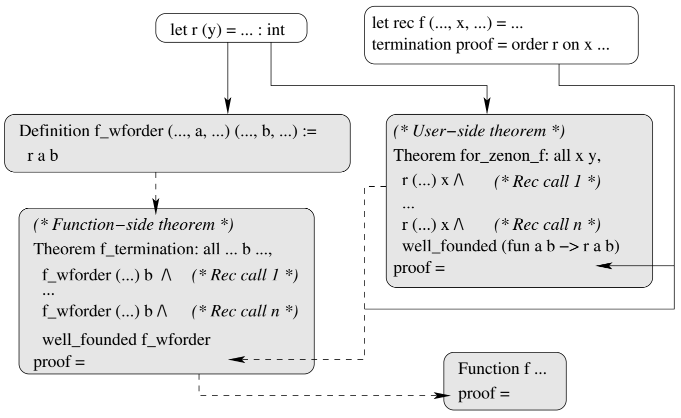
The proof of the theorem is automatically generated by the compiler. The proof proceeds as follows. It first introduces in the hypothesis context the conclusions of the user-side termination theorem (named here `__user_dec1` and `__user_rem_dec_n_wf`). The proof is split, thus separating the decreasing and well-foundedness obligations. Decreasing obligations are proved automatically thanks to the simplification of the goal and hypotheses found in the context. This is to be repeated as many times as there are recursive calls. The last part proves that the relation defined for `Function` is well-founded. We reformulate the goal statement such that it appears as well-foundedness of the user-defined relation (denoted by `R` in the proof script) composed with the projection needed to extract the decreasing argument from the tuple of arguments (here `__tpl_firstprj2`). The proof ends with the application of the Coq standard library theorem `wf_inverse_image` that establishes that if a relation is well-founded then this relation composed with any function is also well-founded. It asks for the well-foundedness of the user-relation which is a hypothesis (here `__user_rem_dec_n_wf`).

3.2.5 Creation of the Recursive Function Definition Using `Function`

The Coq `Function` can now be generated, using the generated relation (in our example, `div_wforder`). The compiler emits the "glue code" of the final proof, using the theorem generated in 3.2.4 and some low-level theorems of the FoCaLiZe standard library. These low-level theorems mostly deal with properties about the equality.

```
Function div (__arg: (int__t * int__t))
  {wf div_wforder __arg}: int__t :=
  match __arg with
    | (a, b) =>
       (if (_bar__bar_ (_lt__eq_ a 0) (_lt__eq_ b 0)) then 0
       else ((if ((_lt_ a b)) then 0 else _plus_ 1 (div ((_dash_ a b), b)))))
    end.
  Proof.
    elim div_termination.
    intros __for_function_dec1 __for_function_rem_dec_n_wf.
    split. intros.
    eapply __for_function_dec1 ; eauto ||
      (apply coq_builtins.EqTrue_is_true; assumption) ||
      (apply coq_builtins.IsTrue_eq_false2; assumption) ||
      (apply coq_builtins.syntactic_equal_refl).
    (* Remaining proof of well-foundedness  ... *)
    assumption.
    Qed.
```

Figure 3 gives a detailed structure of the complete compilation of a function `f` whose termination relies on a relation `r`. Grayed blocks are those generated by the compiler while white ones are the code of the user.



let r (y) = ... : int
let rec f (..., x, ...) = ...
termination proof = order r on x ...
Definition f_wforder (..., a, ...) (..., b, ...) :=
r a b
(* User−side theorem *)
Theorem for_zenon_f: all x y,
r (...) x /\ (* Rec call 1 *)
...
r (...) x /\ (* Rec call n *)
well_founded (fun a b −> r a b)
proof =
(* Function−side theorem *)
Theorem f_termination: all ... b ...,
f_wforder (...) b /\ (* Rec call 1 *)
...
f_wforder (...) b /\ (* Rec call n *)
well_founded f_wforder
proof =
Function f ...
proof =

Fig. 4. Detailed Structure of a Function with its Proof

4 Measure Functions

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-foundedness is no more asked of 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-foundedness 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. Indeed, 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 `qsort` quick-sorting the elements of a list. These elements are provided by the parameter `A` of the species `AList` and are sorted according to their ordering method `le`. At each step, the quick-sort algorithm performs two recursive calls on two sub-lists obtained by splitting the list to sort. The termination is ensured by the shorter lengths of these sub-lists compared to the initial list. We first write the method `length` whose termination is simply structural on its argument `l`. The sorting function requires a `partition` method to split a list into the two sub-lists respectively containing the elements lower or equal and strictly greater than a "pivot" value, according to the method `le` which is left only declared. Finally, we write the method `qsort`, stating a termination proof using the measure `length` on its argument `l`, but we do not show the proof itself yet:

```
species AList (A is Comparable) =
  let rec length (l : list (A)) =
    match l with
    | [] -> 0
    | h :: q ->  1 + length (q)
    termination proof = structural l ;

  let rec partition (l , x : A) =
    match l with
    | [] -> ([], [])
    | h :: q ->
       let p = partition (q, x) in
       if A!le (h, x) then
         (h :: (fst (p)), snd (p))
       else (fst (p), h :: (snd (p)))
  termination proof = structural l ;
```

```
  let rec qsort (l : list (A)) =
    match l with
    | [] -> []
    | x :: r ->
      match r with
      | [] -> l
      | y :: q ->
         let p = partition (r, x) in
         app (qsort (fst (p)),
                 x :: (qsort (snd (p))))
  termination proof =
    measure length on l ... ;
end ;;
```

Using the species and collection parameters features of FoCaLiZe, this example also shows that the compilation model introduced in this paper fits the usual dependency calculus and λ-lifting mechanisms used by the compiler.

Note that in this example, `qsort` has only one argument. In this particular case, the mechanism described in the section 3 to manage the tuple of all the arguments of a function will have no work to do.

4.1 User View

From the user's point of view, the measure being `length` on the argument `l`, the first proof obligation is $\forall l : list(A), 0 \leq length(l)$.

Then, the method `qsort` having two recursive calls, the two decreasing proof obligations are:
$\forall l : list(A), \forall x : A, \forall r : list(A), \forall y : A, \forall q : list(A), \forall p : list(A) * list(A),$
$(l = x :: r) \rightarrow (r = y :: q) \rightarrow (partition(r, x) = p) \rightarrow length(fst(p)) < length(l)$
and
$\forall l : list(A), \forall x : A, \forall r : list(A), \forall y : A, \forall q : list(A), \forall p : list(A) * list(A),$
$(l = x :: r) \rightarrow (r = y :: q) \rightarrow (partition(r, x) = p) \rightarrow length(fst(p)) < length(l)$
where variables bound on the execution path leading to the recursive call must be accumulated as hypotheses. Here, the recursive calls being in pattern-matching cases, `x` and `r` must be related to the matched value `l` (idem for `y`, `q` and `r`). The core of the decreasing facts is the $<$ relation between the length of the recursive calls arguments and the length of the list in the current call.

For readability, instead of inlining the proofs of these obligations, the user can state two related properties or theorems before the function `qsort` itself:

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

  theorem mes_decr_fst :
    all l : list (A), all x : A, all r : list (A), all y : A,
    all q : list (A), all p : list (A) * list (A),
    (l = x :: r) -> (r = y :: q) -> (partition (r, x) = p) ->
      length (fst (p)) < length (l)
  proof = ... ;
```

```
theorem mes_decr_snd :
  all l : list (A), all x : A, all r : list (A), all y : A,
  all q : list (A), all p : list (A) * list (A),
  (l = x :: r) -> (r = y :: q) -> (partition (r, x) = p) ->
    length (snd (p)) < length (l)
 proof = ... ;
```

Note that `length_pos` is a *property*, not a theorem: it is not yet proved. Hence, thanks to the dependency calculus, it will be λ-lifted. This shows that our code generation model is not impacted by termination proofs.

Now the termination proof (see Fig. 5) 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 the compiler to assemble the previous steps and generate some stub code to close the proof.

```
let rec qsort (l : list (A)) = ...
termination proof = measure length on l
  <1>1 prove all l : list (A), all x : A,
       all r : list (A), all y : A, all q : list (A),
       all p : list (A) * list (A),
       (l = x :: r) -> (r = y :: q) -> (partition (r, x) = p) ->
         length (fst (p)) < length (l)
        by property mes_decr_fst
  <1>2 prove all l : list (A), all x : A,
       all r : list (A), all y : A, all q :  list (A),
       all p : list (A) * list (A),
       (l = x :: r) -> (r = y :: q) -> (partition (r, x) = p) ->
         length (snd (p)) < length (l)
       by property mes_decr_snd
  <1>3 prove all l: list (A), 0 <= length (l)
       by property length_pos
  <1>e qed coq proof {*wf_qed*} ;
```

Fig. 5. Termination Proof for `qsort`

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 following shape:

```
<1>x proofs of decreasing for each recursive call
     (the same statements as 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 as 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. It 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 needed by the proof done in 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 as 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 point is exactly the same as for a termination proof using a well-founded relation. This allows a code generation model as close as possible for both kinds of proofs.

Figure 3 gives an overview of structure of the compilation of a function. Grayed blocks are those generated by the compiler while white ones are the code of the user.

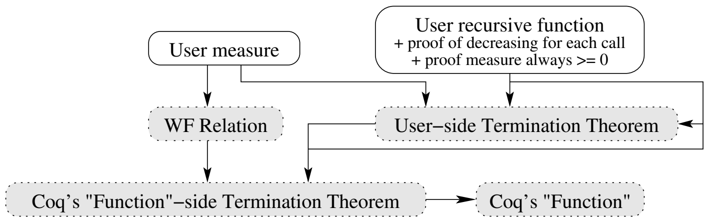

User measure
User recursive function
+ proof of decreasing for each call
+ proof measure always >= 0
WF Relation
User–side Termination Theorem
Coq's "Function"–side Termination Theorem
Coq's "Function"

Fig. 6. Global Structure of a Function with its Proof

4.2.1 Creation of the Relation Expected by `Function`

The dependency calculus of `FoCaLiZe`, extended to termination proofs indicates that the proof decl-depends on the declarations of `length`, `app`, `partition`, `length_pos`, `mes_decr_fst` and `mes_decr_snd`. Hence these dependencies are first λ-lifted and lead to extra parameters (whose names are prefixed by `abst_`) for the relation. Since `Coq` makes polymorphism explicit, the types of collection parameters also must be λ-lifted (here `_p_A_T : Set` for the parameter `A`).

The expected relation takes two arguments `__x` and `__y` which are tuples bringing together all the arguments of the function. We use the same projection mechanism that in Subsection 3.2.2. Once components are extracted, there remains to apply the user measure on them and compare the results with the standard order on naturals (<, which is generated as `_lt_`, itself bound to the corresponding Coq definition). The generated relation is given below (where `_amper__amper_` denotes the conjunction):

```
Definition qsort_wforder (_p_A_T : Set)
  (abst_app : list__t _p_A_T -> list__t _p_A_T -> list__t _p_A_T)
  (abst_length : list__t _p_A_T -> int__t)
  (abst_partition : list__t _p_A_T -> _p_A_T ->
                        list__t _p_A_T * list__t _p_A_T)
  (abst_length_pos :
    forall l : list__t _p_A_T, Is_true (_lt__eq_ 0 (abst_length l)))
  (abst_mes_decr_fst : ...) (abst_mes_decr_snd : ...) :=
  (__x __y : list__t _p_A_T) : Prop :=
    Is_true (_amper__amper_
              (_lt__eq_ 0 (abst_length __y))
              (_lt_ (abst_length __x) (abst_length __y))).
```

4.2.2 Creation of the User-Side Termination Theorem

Next, the user termination proof must be generated. We find again the `Section` mechanism used in 3.2.3 in order to keep Zenon in a first-order environment. The shape of the generated code is very close to the one generated for termination proofs by a relation and is shown below. Line numbers and proof steps refer to the code given at Fig. 5.

```
Section qsort.
  Section Proof_of_qsort.
  Variable _p_A_T, abst_app, abst_length, abst_partition : ...
  Hypothesis abst_length_pos, abst_length_pos, abst_mes_decr_fst,
       abst_mes_decr_snd : ...
  Section __E_1.
    (* Step <1>1 line 3. *)
    Theorem __E_1_LEMMA : forall ...,
      Is_true ((_eq_ _) l (@ List.cons _p_A_T x r)) ->
      Is_true ((_eq_ _) r (@ List.cons _p_A_T y q)) ->
      Is_true ((_eq_ _) (abst_partition r x) p) ->
      Is_true (_lt_ (abst_length ((fst _ _) p)) (abst_length l)).
    (* ... Zenon proof term inlined here ... *)
    End __E_1.
  Section __E_2.
  (* Step <1>2 line 9. *)
  Theorem __E_2_LEMMA : forall ...,
    Is_true ((_eq_ _) l (@ List.cons _p_A_T x r)) ->
    Is_true ((_eq_ _) r (@ List.cons _p_A_T y q)) ->
    Is_true ((_eq_ _) (abst_partition r x) p) ->
    Is_true (_lt_ (abst_length ((snd _ _) p)) (abst_length l)).
    (* ... Zenon proof term inlined here ... *)
    End __E_2.
  Section __E_3.
  (* Step <1>3 line 16. *)
  Theorem __E_3_LEMMA :
    forall l : (list__t _p_A_T), Is_true (_lt__eq_ 0 (abst_length l)).
    (* ... Zenon proof term inlined here ... *)
  (* Theorem's body. *)
  Theorem for_zenon_qsort :
    (forall ...,
      (Is_true ((_eq_ _) l (@ List.cons _p_A_T x r))) ->
      (Is_true ((_eq_ _) r (@ List.cons _p_A_T y q))) ->
      (Is_true ((_eq_ _) (abst_partition r x) p)) ->
      Is_true (_lt_ (abst_length ((snd _ _) p)) (abst_length l)))
    /\
    (forall ...,
      (Is_true ((_eq_ _) l (@ List.cons _p_A_T x r))) ->
      (Is_true ((_eq_ _) r (@ List.cons _p_A_T y q))) ->
      (Is_true ((_eq_ _) (abst_partition r x) p)) ->
      Is_true (_lt_ (abst_length ((fst _ _) p)) (abst_length l)))
    /\
    (forall __x, Is_true (_lt__eq_ 0 (abst_length __x))).
  Proof.
  generalize __E_3_LEMMA. generalize __E_2_LEMMA. generalize __E_1_LEMMA.
  unfold is_well_founded. intros. SplitandAssumption. Qed.
End Proof_of_qsort.
```

Once each step of the proof but the concluding one has been generated, the compiler detects that the concluding one refers to termination. It then generates and proves the theorem `for_zenon_qsort` stating the termination obligations as the user sees them: only dealing with the unique decreasing argument. This theorem's statement is the conjunction of all the decreasing lemmas and the positiveness of the measure introduced in 4.1. The proof generated by the compiler is again very close to the one for a termination by a well-founded relation, and uses the same automation techniques.

4.2.3 Creation of the `Function`-Side Termination Theorem

Since the user-side theorem is available, its counterpart as expected by `Function` has to be emitted. Like for proofs by a relation in 3.2.4, this theorem states the same proof obligations as the user-side theorem but dealing with the tuple of all the arguments. This theorem requires the same `Variable` and `Hypothesis` as the user-side one since the dependencies are the same. To lighten the presentation we do not repeat them in the following generated code sample.

```
Theorem qsort_termination :
  (forall ...,
    (Is_true ((_eq_ _) l (@ List.cons _p_A_T x r))) ->
    (Is_true ((_eq_ _) r (@ List.cons _p_A_T y q))) ->
    (Is_true ((_eq_ _) (abst_partition r x) p)) ->
    (qsort_wforder
      _p_A_T abst_app abst_length abst_partition abst_length_pos
      abst_mes_decr_fst abst_mes_decr_snd) ((snd _ _) p) l)
   /\
   (forall ...,
     (Is_true ((_eq_ _) l ((@ List.cons _p_A_T x r)))) ->
     (Is_true ((_eq_ _) r ((@ List.cons _p_A_T y q)))) ->
     (Is_true ((_eq_ _) (abst_partition r x) p)) ->
     (qsort_wforder
       _p_A_T abst_app abst_length abst_partition abst_length_pos
       abst_mes_decr_fst abst_mes_decr_snd) ((fst _ _) p) l)
   /\
   (well_founded
     (qsort_wforder
       _p_A_T abst_app abst_length abst_partition abst_length_pos
       abst_mes_decr_fst abst_mes_decr_snd)).
  Proof.
  unfold qsort_wforder;simpl.
  elim (for_zenon_qsort _p_A_T abst_app abst_length abst_partition
          abst_length_pos abst_mes_decr_fst abst_mes_decr_snd).
  intro __user_dec1.
  intro __user_rem_dec_n_wf.
  (* Separate decreasing obligations and well-foundation. *)
  split. intros. apply coq_builtins.andb_intro; eauto.
  split. intros. apply coq_builtins.andb_intro; eauto.
  (* There, only remains the well-foundation obligation. *)
  set (R := fun x y : int__t =>
              Is_true (_amper__amper_ (_lt__eq_ 0 y) (_lt_ x y))).
  change (well_founded (fun __c __d : ((list__t _p_A_T)) =>
                          R (abst_length __c) (abst_length __d))).
  apply wf_inverse_image.
  apply wf_incl with (R2 := (fun x y : Z => 0 <= y /\ x < y)).
  unfold inclusion, R.
  unfold int__t, _amper__amper_, _lt__eq_, _lt_, bi__and_b, bi__int_leq,
      bi__int_lt.
  intros x y.
  elim (Z_le_dec 0 y); intro; elim (Z_lt_dec x y); simpl; intros;
  intuition. apply (Zwf_well_founded 0). Qed.
```

Again, the proof of the theorem is automatically generated by the compiler. It also consists in separating the decreasing and well-foundedness obligations. The interconnecting Coq script is more complex than for proofs by a relation. In particular, in the first part about measure decreasing, the proof is not so straightforward than previously because we have to deal with integers and the $<$ relation. The well-foundedness obligation proof follows the same scheme, it relies on three library theorems, `wf_inverse_image` to deal with the relevant projection, `Zwf_well_founded 0` which is the proof that the restriction of $<$ to the positive integers is a well-founded relation and `wf_incl` establishing that a relation included in a well-founded one, is also well-founded.

4.2.4 Creation of the Recursive Function Definition Using `Function`

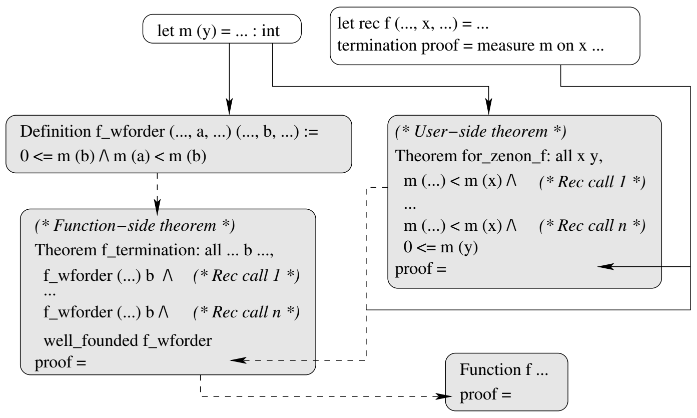
The Coq `Function` can now be generated, using the generated relation. Exactly in the same way as for proofs with a relation, the compiler ensures the glue to build the final proof, taking benefit from the theorem generated in 4.2.3. In particular, the final soldering Coq script is exactly the same as for a termination proof with a relation.

Note that because of dependencies in the user code that were λ-lifted, `qsort_wforder` had some extra arguments. They must be instantiated to provide `Function` with a correct relation in its `wf` clause. This is done by applying `qsort_wforder` to the effective methods definitions computed by the late-binding resolution (`_p_A_T`, `abst_length`, etc). Below is shown the final and complete code of the compiled function `qsort`.

```
Function qsort (__arg: (list__t _p_A_T))
  {wf (qsort_wforder _p_A_T abst_app abst_length abst_partition
         abst_length_pos abst_mes_decr_fst abst_mes_decr_snd) __arg}:
  list__t _p_A_T :=
  match __arg with (l) =>
    match l with
     | List.nil => @ List.nil  _p_A_T
     | List.cons x r =>
         match r with
          | List.nil => l
          | List.cons  y q =>
              let p := abst_partition r x in
              abst_app (qsort ((fst _ _) p))
                       (@ List.cons _p_A_T x ((qsort ((snd _ _) p))))
          end
     end
end.
Proof.
elim qsort_termination.
intros __for_function_dec1 __for_function_rem_dec_n_wf.
```

```
elim __for_function_rem_dec_n_wf. clear __for_function_rem_dec_n_wf.
intros __for_function_dec2 __for_function_rem_dec_n_wf.
split. intros. eapply __for_function_dec1 ; eauto || (apply coq_builtins.
     EqTrue_is_true; assumption) || (apply coq_builtins.IsTrue_eq_false2;
     assumption) || (apply coq_builtins.syntactic_equal_refl).
split.intros. eapply __for_function_dec2 ; eauto || ... (* Same than above.
     *)
(* Remaining well-foundation... *)
assumption. Qed.
```

Figure 4 gives a detailed structure of the complete compilation of a function `f` whose termination relies on a measure `m`. Grayed blocks are those generated by the compiler while white ones are the code of the user.



let m (y) = ... : int
let rec f (..., x, ...) = ...
termination proof = measure m on x ...
Definition f_wforder (..., a, ...) (..., b, ...) :=
0 <= m (b) /\ m (a) < m (b)
(* User−side theorem *)
Theorem for_zenon_f: all x y,
m (...) < m (x) /\ (* Rec call 1 *)
...
m (...) < m (x) /\ (* Rec call n *)
0 <= m (y)
proof =
(* Function−side theorem *)
Theorem f_termination: all ... b ...,
f_wforder (...) b /\ (* Rec call 1 *)
...
f_wforder (...) b /\ (* Rec call n *)
well_founded f_wforder
proof =
Function f ...
proof =

Fig. 7. Detailed Structure of a Function with its Proof

5 Limitations and Perspectives

The termination proofs as described in this paper are available in the FoCaLiZe repository. 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-foundedness proof from the user’s orders and their own well-foundedness 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: 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 kinds of termination proofs, it is not fully clear how to provide late-binding at the termination level. In FoCaLiZe, definitions can use methods only declared. 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. Indeed, 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.

Finally, only one of the function's parameters can be used to prove decreasing. There is no particular obstacle for this extension, it is only an implementation matter. This restriction only impacts proofs by a measure (which could use several parameters), since such an extension for orders directly leads to lexicographic orders.

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 [19], 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 [5] 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 [4]. 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 due 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 [6] 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 [15], 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 [10]. The termination for another class is handled by using the Size-Change principle [14]. This principle is also used in [13] to provide a tool that automatically determines whether one or mutually recursive functions terminate. This approach allows for a local increasing of recursive arguments. FoCaLiZe does not intend to automatically find proofs: it lets this task to an external prover thanks to some hints given by the user. Moreover, it is yet unclear how to generate a Coq term from such a termination proof.

More generally, Bove, Krauss and Sozeau review in [9] different techniques that have been proposed to formalize partial and general recursive functions in interactive theorem provers.

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 relieve 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. methods that are only declared, e.g. properties used in proof obligations or even the measure or the well-founded relation) thanks to the λ-lifting mechanism used by FoCaLiZe.

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. Abel and T. Altenkirch. A predicative analysis of structural recursion. *J. Funct. Program.*, 12(1):1–41, 2002.
4. 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.
5. 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.
6. Y. Bertot and V. Komendantsky. Fixed point semantics and partial recursion in coq. In *Proceedings of PPDP'2008, Valencia, Spain*, pages 89–96, 2008.
7. 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.
8. S. Boulmé, T. Hardin, D. Hirschkoff, V. Ménissier-Morain, and R. Rioboo. On the way to certify computer algebra systems. In *Proceedings of the Calculemus workshop of FLOC'99*, volume 23 of *ENTCS*. Elsevier, 1999.
9. A. Bove, A. Krauss, and M. Sozeau. Partiality and recursion in interactive theorem provers an overview. *Mathematical Structures in Computer Science*, FirstView:1–51, 6 2015.
10. 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.
11. G. Bury and D. Delahaye. Integrating simplex with tableaux. In H. D. Nivelle, editor, *TABLEAUX 2015, Wrocław, Poland*, volume 9323 of *LNCS*, pages 86–101. Springer, 2015.
12. C. Dubois, T. Hardin, and V. Donzeau-Gouge. Building certified components within FOCAL. volume 5 of *Trends in Functional Programming*, pages 33–48, 2006.
13. P. Hyvernat. The Size-Change Termination Principle for Constructor Based Languages. *Logical Methods in Computer Science*, 10(1), Feb. 2014.
14. A. Krauss. Certified size-change termination. In *Automated Deduction - CADE-21, Bremen, Germany, 2007, Proceedings*, volume 4603 of *LNCS*, pages 460–475. Springer, 2007.
15. A. Krauss. Partial and nested recursive function definitions in higher-order logic. *J. Autom. Reasoning*, 44(4):303–336, 2010.
16. L. Lamport. How to write a proof. Research report, Digital Equipment Corporation, 1993.
17. F. Pessaux. Focalize: Inside an F-IDE. In *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014*, volume 149 of *EPTCS*, pages 64–78, 2014.
18. V. Prevosto. *Conception et Implantation du langage FoC pour le développement de logiciels certifiés*. PhD thesis, Université Paris 6, sep 2003.
19. K. Slind. Another look at nested recursion. In *TPHOLs 2000, Portland, Oregon, USA, Proceedings*, volume 1869 of *LNCS*, pages 498–518. Springer, 2000.