Proof Pearl: A Practical Fixed Point Combinator for Type Theory

Arthur Charguéraud

INRIA
arthur.chargueraud@inria.fr

Abstract. Type theories need to enforce some restrictions on recursive definitions in order to remain sound. Depending on the implementation, these restrictions may prevent the user from defining recursive functions as conveniently as in a functional programming language. This paper describes a fixed point combinator that can be applied to any functional. A fixed point equation can be derived for the recursive function produced, provided that all recursive calls are made on arguments that are smaller than the current argument, with respect to a decidable well-founded relation or a measure. The approach is entirely constructive, and does not require the user to program with dependent types. It supports partial functions, n-ary functions, mutual recursion, higher-order recursion and nested recursion. It has been implemented and experimented in Coq.

1 Introduction

In type theories such as those of Coq [5], Isabelle/HOL [15] or HOL [14], all functions must be total. In particular, all recursive functions must be provably terminating. If this was not the case, one could define the diverging function "$\mathsf{let\ rec}\, f\, x = 1 + f\, x$", and then simplify the equality "$f\, x = 1 + f\, x$" towards "$0 = 1$", which would be an obvious inconsistency. Thus, the definition of recursive functions must be restricted in some way.

While Isabelle and HOL feature packages that support a large class of recursive functions, a similar technology is not quite available in Coq. Existing proposals for defining recursive functions in Coq run into complications with higher-order recursion, nested recursion and mutual recursion. In this work, we show how to combine and adapt ingredients that have appeared in the literature in order to derive a fixed point combinator that can be defined in a constructive type theory.

More precisely, this paper describes a technique for building a recursive function f of type "$A \to B$" given a functional F that describes its body. The functional F can be any function of type "$(A \to B) \to (A \to B)$", where the first argument corresponds to the function to be applied for making recursive calls. The only requirement for constructing the fixed point of F is the existence of a decidable well-founded relation R of type "$A \to A \to \mathsf{Prop}$", such that all recursive calls are made on values that decrease with respect to R. In particular, this relation can be implemented using a computable measure that decreases on each recursive call.

The key properties of our technique can be summarized as follows:

- **Definitional.** All the development is conducted within the theory, namely the Calculus of Inductive Constructions, and is thus entirely constructive.
- **Lightweight.** The fixed point combinator and its corresponding fixed point equation theorem are packaged in a library. The core of the theory fits in 30 lines of Coq. No tool support is needed.
- **Simple to use.** The only work involved for the user is to define the well-founded relation (or the measure) and to establish the decrease on recursive calls. Basic knowledge of Coq suffices to use the technique. In particular, it is not required to write definitions in dependently-typed programming style.
- **Well-integrated.** A fixed point equation is established for the recursive function, so that its body can be unfolded at any time through a simple rewriting operation. Moreover, recursive function may be used before being proved terminating.
- **Executable.** Recursive functions obtained through this technique are executable, in the sense that Coq's normalization is able to evaluate the result of a function applied to given arguments.
- **General.** The technique applies to a very large class of recursive functions. Moreover, all of Coq's language is available for defining the body of the function.

In short, our technique[1] provides the same result as the built-in recursion facility of Coq, except that it does not enforce a syntactic restriction to ensure that arguments of recursive calls are structurally smaller, but instead requires a termination proof based on a well-founded relation. This proof takes the form of an external lemma to be proved interactively.

The paper is organized as follows. First, we give some theoretical background useful to this development. Second, we describe our constructive fixed point combinator and the theorem for deriving the associated fixed point equation, in the case of a total recursive function of one argument. Third, we explain how to extend the technique so as to support more advanced forms of recursion. Fourth, we describe an interesting variation, specialized for the case of decreasing measures. Finally, we discuss related work and conclude.

2 Theoretical background

We start by recalling standard knowledge useful to this paper: fixed point combinators, fixed point equations, termination arguments in terms of measures and well-founded relations, decidable relations, and contraction conditions.

[1] The Coq script corresponding to the development can be found at: `http://arthur.chargueraud.org/research/2009/fixwf`

2.1 Fixed point combinator and fixed point equation

A *fixed point combinator* is a function able to build a recursive function f given its functional F. The two functions f and F are then related by a *fixed point equation*. Such an equation states that the application of f to an argument x is equal to the application of F to f and x. Thereby, the fixed point equation allows a one-step unfolding of the body of the recursive function. In summary, a fixed point combinator is a function fix satisfying the following property:

$$f = \mathsf{fix}\, F \quad \Rightarrow \quad f\,x = F\,f\,x$$

While a generic fixed point combinator can be easily defined in an untyped functional programming languages (e.g. the λ-calculus), sound type theories cannot accommodate such a combinator. As explained in the introduction, there must be some restriction in order to ensure that the recursive functions being built using the combinator are terminating on all arguments.

2.2 Proofs of termination: measures and well-founded relations

A *decreasing-measure argument* is a simple and powerful technique for establishing that a recursive function terminates on all input. The principle is to exhibit a particular function, called the measure, that computes the *size* of an argument, and to verify that all recursive calls are made on arguments of smaller size than the current argument. More precisely, a measure, written μ, is a total function of type "$A \rightarrow \mathsf{nat}$", where A is the type of arguments and nat is the type of natural numbers. To show that a recursive function f terminates, it suffices to prove that for any application of f to an argument x, if a recursive call is made on a value y then the inequality "$\mu\,y < \mu\,x$" holds.

A generalization of the decreasing measure technique relies on the use of *well-founded relations*. Consider a relation R of type "$A \rightarrow A \rightarrow \mathsf{Prop}$". Throughout the paper, we say that x is smaller than y if and only if "$R\,x\,y$". The purpose of well-foundedness is to ensure that there exists no infinite decreasing chain of values starting from any given value. More precisely, the relation R is said to be *well-founded* if all the values of type A are *accessible*, where x is accessible for R, written "$\mathsf{Acc}\,R\,x$", if and only if all the elements smaller than x are themselves accessible. The formal definition appears below in Coq syntax.

```
Inductive Acc (R:A->A->Prop) (x:A) : Prop :=
  | Acc_intro : (forall y:A, R y x -> Acc R y) -> Acc R x.

Definition well_founded (R:A->A->Prop) :=
  forall x:A, Acc R x.
```

Given a well-founded relation R, the induction principle associated with the definition of accessibility leads to a *well-founded induction principle*:

$$\forall P : (A \rightarrow \mathsf{Prop}),\ \big(\forall x : A,\ (\forall y : A,\ R\,y\,x \Rightarrow P\,y) \Rightarrow P\,x\big) \Rightarrow (\forall x : A,\ P\,x)$$

It states that in order to show that a property P is true for all values of type A, it suffices to prove that P holds for an arbitrary value x under the assumption that P already holds for any value y smaller than x with respect to R. In a similar way, one can derive a *well-founded recursion principle*, that allows to define functions that may call themselves recursively on any value smaller than the current argument.

Note that the use of a measure argument is simply a particular case of well-founded recursion. The well-founded relation associated with a measure μ is the predicate "$\lambda x.\, \lambda y.\, (\mu\, x < \mu\, y)$".

2.3 Decidable relations

A binary relation R is *decidable* if, for any x and y, the proposition "$R\, x\, y$" can be shown to be either true or false. In a classical logic, all propositions are decidable, since the excluded middle principle states than any proposition is either true or false. In a constructive logic however, not all propositions are decidable.

We argue that relations used to argue for termination are most often decidable. First, the well-founded relation associated to any computable measure μ is decidable. Second, any relation that can be implemented as a boolean function, that is, a function of type "$A \to A \to \mathsf{bool}$", is immediately decidable. In particular, this concerns all the comparison functions on first-order data structures (provided these structures contain elements for which equality is decidable). Third, relations defined by combination of decidable relations, such as lexicographical product, relational product, or multiset predecessor ordering, are also decidable. Thus, decidability applies to a very large class of comparison relations. Examples of relations that are typically not decidable are comparison relations defined on higher-order data structures, i.e. data structures that contain functions.

In Coq, for any decidable binary relation of type "$A \to A \to \mathsf{Prop}$", one can define a binary boolean function of type "$A \to A \to \mathsf{bool}$" that describes the same relation. Thus, rather than quantifying over binary relations for which there exists a proof of decidability, we simply quantify over binary boolean functions. Note that thanks to its coercion mechanism, Coq is able to typecheck boolean values as propositions.

2.4 A termination criteria: the contraction condition

Given a functional F and a well-founded relation R, the *contraction condition* gives a sufficient condition for the functional F to describe a terminating function. The contraction condition for F and R is the following proposition:

$$\forall\, x\, f_1\, f_2, \quad (\forall\, y,\, R\, y\, x \;\Rightarrow\; f_1\, y = f_2\, y) \quad \Rightarrow \quad F\, f_1\, x = F\, f_2\, x$$

The expression "$F\, f_1\, x$" describes the body of the recursive function, with an abstract function f_1 being used for recursive calls. Similarly, "$F\, f_2\, x$" describes the same body, with another abstract function f_2 being used for recursive calls.

The goal is to prove that these two expressions yields equal results, knowing only one fact about f_1 and f_2: they produce equal results when applied to arguments strictly smaller than x, with respect to R. The key is to realize that proving this implication amounts to justifying that the each recursive call is applied to a value y smaller than the current argument x.

The use of contraction conditions is a standard technique for establishing fixed point equations. Its use appears throughout the type theory literature, in particular in the works of Harrison [7], Matthews [12], and Balaa and Bertot [1].

3 A constructive fixed point combinator

We now define a constructive fixed point combinator by well-founded recursion and derive the associated fixed point equation in terms of a contraction condition.

3.1 Definition of the fixed point combinator

We define the fixed point of a function F by well-founded recursion on values of the input type A. Thereby, as we define the value "$f\,x$" for some x, we may use the value of "$f\,y$" for any y smaller than x with respect to R, that is, whenever "$R\,y\,x$" holds. We define f as follows

$$f\,x \quad\equiv\quad F\,f'\,x$$

where f' is the restriction of f to values smaller than x. More precisely, we define f' as:

$$f' \quad\equiv\quad \lambda y : A.\ \mathsf{if}\ R\,y\,x\ \mathsf{then}\ f\,y\ \mathsf{else\ arbitrary}$$

An arbitrary value exists as soon as the return type B is inhabited. One can check that f is applied to y only if y is smaller than x, that is, only when the recursion hypothesis is available. The case analysis performed to test whether y is smaller than x, i.e. "$R\,y\,x$", is where the decidability of R is needed.

Abstracting over F and R in the above, we define our constructive fixed point combinator, named fixwf. Its Coq definition appears below. It takes several arguments. First, A is the input type: it has type Type. Second, B is the return type: it has type iType, which describes the subset of Type containing all inhabited types. Thereafter, the notation arbitrary can be used wherever a value of type B is expected. Then, F is the functional that describes the body of the recursive function, R is the relation used to argue for termination, and W is a proof that R is well-founded.

```
Definition fixwf (A:Type) (B:iType) (F:(A->B)->(A->B))
  (R:A->A->bool) (W:well_founded R) (x:A) :=
  Acc_rect _ (fun x _ f =>
     let f' y := match sumbool_of_bool (R y x) with
                 | left H => f y H
                 | _ => arbitrary end in
     F f' x)
  (W x).
```

When applied to these arguments, fixwf returns a function that maps a value x of type A towards a value of type B. In the body of the definition, Acc_rect is the recursion combinator associated with the predicate Acc, "$W\,x$" is a proof that x is accessible for R, f is the function that comes from the induction hypothesis, sumbool_of_bool allows to perform a dependent case analysis on "$R\,y\,x$", and H is a proof that "$R\,y\,x$" holds.

Some users of Coq care about efficiency of functions, either because they want to execute functions in Coq, or because they plan to extract functional programs. Recursive functions built with fixwf suffer from a potential source of inefficiency, coming from the fact that the relation R is evaluated at every recursive call. Since the cost of evaluating R is typically proportionnal to the size of its arguments, a recursive function that could have been executed in linear time will typically end up with an implementation of quadratic complexity. In §5, we show how to recover a linear complexity in the case where R is implemented using a measure.

3.2 Deriving the fixed point equation

The formal statement of the theorem that allows to derive the fixed point equation from the contraction condition is printed below. Its first hypothesis is that f is a fixed point built using fixwf (it is always true by definition of f when we instantiate the theorem). The second hypothesis is the contraction condition, given in §2.4. The conclusion is the fixed point equation.

```
Theorem fixwf_eq : forall A B F R (W:well_founded R) (f:A->B),
  f = fixwf F W ->
  (forall x f1 f2,
    (forall y, R y x -> f1 y = f2 y) ->
     F f1 x = F f2 x) ->
  forall x, f x = F f x.
Proof.
   introv Eqf H. intros. rewrite Eqf. unfold fixwf.
   match goal with |- ?f x (W x) = _ => set (g := f) end.
   assert (Hany : forall z Hz1 Hz2, g z Hz1 = g z Hz2).
     clear x. intros.
     induction Hz1 as [x Hz1' Eq] using Acc_inv_dep.
     case Hz2. intros Hz2'. simpl. apply H. intros y Ryx.
     destruct (sumbool_of_bool (R y x)).
       apply Eq.
       elimtype False. congruence.
   case (W x). intros K. simpl. apply H. intros y Ryx.
   destruct (sumbool_of_bool (R y x)).
     apply Hany.
     elimtype False. congruence.
Qed.
```

The core of the proof, which comprises the first two lines and the last four lines, is surprisingly straightforward. It consists in unfolding the definition, reasoning by inversion on the proof of the accessibility of x, applying the contraction hypothesis, and then performing a case analysis to check that y is smaller than x. The intermediate result stated and proved in the middle of the script describes the fact that results computed by the fixed point do not depend on the shape of the proof-term provided to justify well-foundedness. Note that this rather technical intermediate result is in fact an immediate consequence of the axiom of Proof Irrelevance, which states that any two proofs of a same proposition are observationally equal.[2]

3.3 Example of use: the log function

Consider the following "log" function: when applied to a natural number n, this function returns the number of time that n can be divided by 2 before reaching 0. The corresponding functional, named Log, is defined below. It uses a boolean comparison function $\leq$, and a function named div2 that implements integer division by 2.

```
Definition Log log n :=
  if n <= 1 then 0 else 1 + log (div2 n).
```

Next, we instantiate the fixed point combinator fixwf onto the functional Log in order to build the recursive function log.

```
Definition log := fixwf Log wf_lt.
```

Note that the input and return types on the one hand, and the boolean "less than" comparison function (named `lt`) on the other hand, need not be provided explicitly as they can be inferred by Coq's *implicit arguments* mechanism. The term `wf_lt` is a proof that the relation `lt` is well-founded. In practice, it is also possible to provide `lt` to the fixed point combinator and have Coq produce a sub-goal stating the well-foundedness of `lt`.

After these definitions, log is a well-defined value of type "nat $\rightarrow$ nat" that may be used in subsequent definitions. Importantly, the function log is able to effectively compute results. For instance, Coq's normalization process reduces the application "log 23" towards the value 4. More advanced reasoning on the function log requires the proof of its fixed point equation. The statement is:

```
Lemma fix_log : forall n,
  log n = Log log n.
```

To prove it, we invoke the theorem fixwf_eq. We are asked to prove the equality "$\mathsf{Log}\, f_1\, n = \mathsf{Log}\, f_2\, n$", using as sole hypothesis the fact that the equality "$f_1\, m = f_2\, m$" holds for any m smaller than n. So, we unfold the definition of Log. If

[2] The author is most grateful to Stéphane Lescuyer for suggesting how to prove the lemma fixwf_eq without using the axiom of Proof Irrelevance.

n is less or equal to 1, then both sides are equal to 0. Otherwise, we have to prove the equality "$f_1\,(\mathsf{div2}\,n) = f_2\,(\mathsf{div2}\,n)$". We apply the hypothesis on f_1 and f_2, and are left to justify that "$\mathsf{div2}\,n < n$". This proposition holds since n is greater than 1. Notice that the latter obligation describes exactly the fact that the recursive call is applied to an argument smaller than n.

Once the fixed point equation established, one can reason on applications of the recursive function. Indeed, it is possible to unfold the body of the recursive function at any time, simply by rewriting the associated fixed point equation and unfolding the definition of the functional that describes the body of the function. For example, in order to unfold the body of the function log in Coq, we invoke the tactic "`rewrite fix_log`", followed with "`unfold Log`".

Packages for defining recursive function often provide an *f-induction* principle. Such a principle allows to establish a property of a recursive function by induction on the structure of the function. This means that one can assume the property to hold for any recursive call occurring in the body of the function. We do not see an easy way of stating such a principle without generating ad-hoc definitions or lemmas for each recursive function. Nevertheless it is possible to establish inductive properties of a recursive function without an explicit f-induction principle. In fact, there are two possibilities for doing so.

First, one can prove the property by well-founded induction on the same relation used to argue for the termination of the recursive function. The unfortunate drawback of this approach is that one has to justify that the recursive calls are applied to smaller arguments, though this has already been done for establishing the fixed point equation. It seems that, it practice, this duplication is not so costly. Indeed, a large majority of subgoals dedicated to verifying the decrease on recursive calls can be discharged automatically. In the rare cases where a nontrivial argument is required, an intermediate lemma can be set up so as to avoid replicating the proof of a same subgoal across several lemmas.

The second possibility is to verify properties of the recursive function in the same time as establishing its termination, that is, in the same time as proving the fixed point equation. This process requires a generalized version of the theorem fixwf_eq, which is described further on (§4.2). With this approach, there is no duplication of proof obligations, however the proof obligations are slightly heavier to work with.

In conclusion, there are two ways of establishing properties of recursive function, both of which seem to be convenient enough to use in practice. The generation of f-induction principles is certainly possible. It could bring some improvement, but it presumably involves a good amount of work to set up and to maintain through evolutions of Coq.

4 Extensions

4.1 Partial function

Some recursive functions are *partial* in that they terminate only on a given domain. A domain may be described by a predicate of type "$A \to \mathsf{Prop}$" (where

A is the type of arguments), called the *pre-condition* of the function. Partial recursive functions can be defined using the combinator fixwf, exactly as total functions. Such partial functions simply return arbitrary values outside their domain.

The fixed point equation can be proved to hold for any argument satisfying the pre-condition. To that end, one has to prove that for any argument x in the domain f, all the recursive calls occurring in the body of "$f\,x$" are applied to values smaller than x and belonging to the domain of f. Thus, in order to take partiality into account, we generalize the theorem `fixwf_eq` so as to quantify only over values satisfying a certain pre-condition P, given by the user. The extended theorem then establishes the fixed point equation only for arguments satisfying P. Its statement appears further on (§4.2).

4.2 Nested recursion

The theorem `fixwf_eq` can also be extended so as to incorporate a *post-condition*, that is, a predicate that describes the result of the recursive function in terms of its argument. The use of post-conditions allows for supporting *nested recursive calls*, i.e. recursive calls whose argument depend on the result of another recursive call (nested recursion appears for instance in the definition of unification algorithms and normalization algorithms). Without the use of a post-condition, it is in general not possible to argue that the outer recursive call is applied to a value smaller than the current argument.

The post-condition can be an arbitrary predicate "$A \to B \to \mathsf{Prop}$", relating arguments of f to their return value. The user has to establish that the body of the recursive function applied to x returns a value r such that "$Q\,x\,r$", under the hypothesis that recursive calls to arguments smaller than x themselves satisfy the post-condition Q. Completing this proof allows to derive not only the fixed point equation, but also the fact that any application of the recursive function satisfy the post-condition Q.

The following theorem strengthens the statement of the fixed point theorem fixwf_eq with both a pre-condition and a post-condition. It can be used to derive a fixed point equation for partial functions and for functions with nested recursion.

```
Theorem fixwf_eq_pre_post : forall A B (P:A->Prop) (Q:A->B->Prop)
  (F:(A->B)->(A->B)) (R:A->A->bool) (W:well_founded R) (f:A->B),
  f = fixwf F W ->
  (forall x f1 f2, P x ->
    (forall y, P y -> R y x -> f1 y = f2 y /\ Q y (f1 y)) ->
     F f1 x = F f2 x /\ Q x (F f1 x)) ->
  forall x, P x -> f x = F f x /\ Q x (f x).
```

Krstić and Matthews [11] call the post-condition Q the *inductive invariant* of the functional F. Based on practical investigations, they conjecture that "termination proofs of most functions defined by nested recursion can be naturally based on some invariant property". They also point out that Q needs not capture the complete specification of the recursive function f. Indeed, it is often the case that a simpler property suffices to argue for the termination of f.

4.3 Specialization to the case of measures

The results that we have obtained can be specialized to the case of computable measures. The statements are simplified since a measure always corresponds to a decidable well-founded relation. Below, we give the prototype of this fixed point combinator specialized for measures, and the statement of its associated fixed point equation theorem.

```
Definition fixm
  (A:Type) (B:iType) (F:(A->B)->(A->B)) (mu:A->nat) : A -> B.

Theorem fixm_eq : forall A B F mu (f:A->B),
  f = fixm F mu ->
  (forall x f1 f2,
    (forall y, mu y < mu x -> f1 y = f2 y) ->
     F f1 x = F f2 x) ->
  forall x, f x = F f x.
```

4.4 N-ary recursive functions

A recursive function of several arguments can be viewed as a function of a single tupled argument. From a theoretical viewpoint, a fixed point combinator for unary function is thereafter sufficient. Nevertheless, from a practical perspective, it is more convenient to have a direct support for curried n-ary functions, so that the uncurrying process needs not be carried out by the user.

Thereafter, we define a fixed point combinator for curried functions. It is implemented in terms of the original fixed point combinator fixwf for unary functions, using currying and uncurrying operators. For example, the combinator for binary functions is defined as follows:

```
Definition fixwf2
  (A1 A2:Type) (B:iType) (F:(A1->A2->B)->(A1->A2->B))
  (R:(A1*A2)->(A1*A2)->Prop) (W:well_founded R) :=
  curry2 (fixwf (fun f => uncurry2 (F (curry2 f))) W).
```

Above, the functional F describes a recursive function of two arguments. The well-founded relation R is a relation on pairs of arguments, that is, a predicate of type "$(A_1 * A_2) \to (A_1 * A_2) \to \mathsf{Prop}$". This pairing allows for the use of the standard definition of well-foundedness. Measures, however, can be defined in a curried fashion, with type "$A_1 \to A_2 \to \mathsf{nat}$".

The fixed point equation for a binary recursive function f describes the unfolding of f when it is applied to two arguments. It takes the form "$f\,x_1\,x_2 = F\,f\,x_1\,x_2$". The contraction condition is generalized similarly, so that the functions involved take two arguments.

```
Theorem fixwf2_eq :
  forall A1 A2 B F R (W:well_founded R) (f:A1->A2->B),
  f = fixwf2 F W ->
  (forall x1 x2 f1 f2,
    (forall y1 y2, R (y1,y2) (x1,x2) -> f1 y1 y2 = f2 y1 y2) ->
     F f1 x1 x2 = F f2 x1 x2) ->
  forall x1 x2, f x1 x2 = F f x1 x2.
```

4.5 Mutually-recursive functions

Mutually-recursive functions can also be encoded in terms of simple recursive functions, using a sum type. Suppose that one wishes to define a pair of two mutually recursive functions f_1 and f_2, of type "$A_1 \to B_1$" and "$A_2 \to B_2$", respectively. Two functionals are to be provided: F_1 of type "$(A_1 \to B_1) \to (A_2 \to B_2) \to (A_1 \to B_1)$" and F_2 of type "$(A_1 \to B_1) \to (A_2 \to B_2) \to (A_2 \to B_2)$". For both functionals, the first argument corresponds to the function for making recursive calls to f_1, while the second argument corresponds to the function for making recursive calls to f_2.

Termination has to be argued by a well-founded relation R of type "$(A_1 + A_2) \to (A_1{+}A_2) \to \mathsf{Prop}$". In the particular case of measures, one may simply use a pair of two measure functions, one of type "$A_1 \to \mathsf{nat}$" and one of type "$A_2 \to \mathsf{nat}$". The size of arguments should decrease on all recursive calls, whether to f_1 or to f_2, and whether from f_1 or from f_2. Once termination is established, two fixed point equations are produced: "$f_1\, x = F_1\, f_1\, f_2\, x$" and "$f_2\, y = F_2\, f_1\, f_2\, y$", which hold for any x and y.

4.6 Higher-order recursion

Higher-order recursion consists in applying a higher-order function to a recursive function, inside the body of this recursive function. While higher-order recursion causes trouble to many approaches to defining recursive functions, it is naturally supported by our fixed point combinator fixwf. We illustrate this with a function treemap, that applies a given function f to all the values in a tree, where a tree is an inductive data structure with two constructors: one for leaves, and one for nodes, which are built on list of trees. The functional that defines treemap is:

```
Definition Treemap treemap f t :=
  match t with
  | leaf n => leaf (f n)
  | node l => node (List.map (treemap f) l)
  end.
```

One can then easily build the recursive function and establish its fixed point equation, using the subtree relationship to argue for termination. The core of the proof involves showing that, for any list of trees l, the equality "$\mathsf{map}\, f_1\, l = \mathsf{map}\, f_2\, l$" holds, under the assumption that the equality "$f_1\, t = f_2\, t$" holds for any tree t member of the list l. This can easily be established by induction on l.

4.7 Dependently-typed recursive functions

The definition of fixwf can be modified so as to build recursive functions with a dependent type of the form "$\forall x : A.\, B\, x$", where A has the type Type and B has the type "$A \to \mathsf{Type}$". Apart from the types, the only difference with the non-dependent version is that we need to show the type "$B\, x$" inhabited for all x. To that end, we require a dummy function d of type "$\forall x : A.\, B\, x$". Of course, if B is too precise a type, then it might not be possible to exhibit such a value d. In this case, one should consider using Sozeau's fixpoint combinator [19], which is more appropriate when programming with dependent types.

5 A variation for the case of measures

In the case of recursive functions that admit a decreasing measure, we can use another fixed point combinator, called fixn. There are two motivations for describing this alternative combinator. Firstly, fixn executes more efficiently than fixwf, because it only needs to evaluate the measure of the initial argument, rather than computing a measure at every recursive call. Secondly, the implementation of fixn relies on the interesting technique of *approximations*, which has also shown to be useful in closely related work [1, 2].

The first key idea is that fixed point of a functional F can be obtained as the limit of a sequence of *approximations*. Intuitively, the n-th approximation of the fixed point of F is a function that behaves exactly as the fixed point of F on any argument whose evaluation requires less than n levels of recursive calls. The sequence of approximation is formally defined as follows. The 0-th approximation is an arbitrary function of type "$A \to B$", and the $(n+1)$-th approximation is defined as the application of the functional F to the n-th approximation:

$$\begin{cases} \mathsf{approx}\ 0 & \equiv \mathsf{arbitrary} \\ \mathsf{approx}\ (n+1) & \equiv F\,(\mathsf{approx}\ n) \end{cases}$$

The second key idea is that, given a computable measure μ that decreases on each recursive call, we can effectively compute a value of n such that the n-th approximation is correct for an argument x. Indeed, we know that at most "$\mu\, x$" levels of recursive calls need to be made in order to evaluate f on the input x. Thus, for any value of n greater than "$\mu\, x$", the n-th approximation is certainly good enough for evaluating f on the argument x. Formally:

$$\forall\, x, \quad \forall\, n > \mu\, x, \quad \mathsf{approx}\ n\ x = f\ x$$

In particular, we may consider the first value of n that works, that is "$1 + \mu\, x$", and *define* the function f as the function that maps a value x to "$\mathsf{approx}\ (1 + \mu\, x)\ x$". The combinator fixn implements this idea. Its formal definition follows.[3]

[3] Variations on fixn can be defined so as to provide direct support for n-ary functions or mutually-recursive functions. Details can be found in the Coq development.

```
Definition fixn (A:Type) (B:iType) (mu:A->nat)
                (F:(A->B)->(A->B)) (x:A) :=
  (fix approx (n:nat) {struct n} :=
     match n with
     | 0    => fun _ => arbitrary
     | S n' => F (approx n')
     end) (1 + mu x) x.
```

If the measure μ decreases on recursive calls, then it can be proved that the combinator fixn effectively builds the fixed point of the functional F, meaning that a fixed point equation can be established. The theorem, not shown here due to space limitations, has a statement similar to the theorem fixm_eq (see §4.3).

6 Related work

Slind [16, 17, 18] has developed a recursion package called TFL, implemented both in Isabelle [15] (in the package *recdef*) and in HOL4 [14]. The starting point is a definition of the *restriction* of a function with respect to a relation R and a value x, standard in the literature (see e.g. [9]):

$$(f \mid R,\, x) \quad \equiv \quad \mathsf{if}\ R\,y\,x\ \mathsf{then}\ f\,y\ \mathsf{else\ arbitrary}$$

Given a well-founded relation R and a functional F, a classic theorem states that there exists a unique function f such that for all x the equation "$f\,x = F\,(f \mid R, x)\,x$" holds. Slind shows that f can be obtained as a function of F and R, in the form "$f \equiv \mathsf{wfrec}\,F\,R$". The definition of the combinator wfrec involves the application of Hilbert's choice operator to a set of so-called approximants. Slind [16] mentions Nipkow as the original author of this formalization. The proof of the fixed point equation "$f\,x = F\,f\,x$" for a particular functional F is obtained by showing that for any recursive call on a value y, the application "$(f \mid R, x)\,y$" is equal to "$f\,y$". By definition of the restriction operator, this is equivalent to showing "$R\,y\,x$", which is indeed the expected proof obligation. These termination conditions are brought to top-level by applying a number of contextual rewriting operations performed mechanically by a tool, and can then be discharged by the user.

Krstić and Matthews [11] later suggested a better support for nested recursion, making use of contraction condition and inductive invariants (see §4.2). Writing P the pre-condition that describes the domain of the function and Q the inductive invariant (also called the post-condition), their result is summarized by the following theorem.

$$\begin{array}{l}
\big(\forall x\,f.\ (\forall y.\ P\,y \Rightarrow\ R\,y\,x \Rightarrow\ Q\,y\,(f\,y)) \Rightarrow\ Q\,x\,(F\,f\,x)\big) \ \Rightarrow \\
\big(\forall x\,f_1\,f_2.\ (\forall y.\ P\,y \Rightarrow\ R\,y\,x \Rightarrow\ f_1\,y = f_2\,y) \Rightarrow\ (\forall y.\ P\,y \Rightarrow\ Q\,y\,(f_1\,y)) \\
\hfill \Rightarrow\ F\,f_1\,x = F\,f_2\,x\big) \ \Rightarrow \\
f = \mathsf{wfrec}\,F\,R \ \Rightarrow\ f\,x = F\,f\,x
\end{array}$$

Notice the close similarity with the statement of our theorem fixwf_eq_pre_post. Of course, the proofs of the two theorems are quite different, since the fixed point combinators wfrec and fixwf have different definitions.

Krauss has developed the *function* package [10] for Isabelle/HOL [15]. Rather than using well-founded recursion, it relies on the definite description operator in order to build fixed points. To that end, the tool builds a binary relation between input and output values that describes the graph of the recursive function. It proves, by well-founded induction, that each input value is in relation with exactly one output value. It then derives, using the description operator, the existence of a function that implements that binary relation. All the results about the recursive function being built are quantified over values that are in the domain of the function. The package generates introduction rules for showing that a particular argument belongs to the domain of the function. Thus, termination proofs can be completely separated from the definition of the function. Nested recursion is supported via a general form of f-induction due to Giesl [6]. Mutually-recursive functions are encoded using sum types. Higher-order recursion is partially supported, through a mechanism of congruence rules (these rules have to be declared in advanced for the higher-order functions used in a recursive function definition). A fairly similar package for recursion has been developed by Harrison for HOL Light [8].

Balaa and Bertot [1] define a constructive fixed point combinator recwf, defined by well-founded recursion, which builds a function f of dependent type "$\forall x : A.\ (B\,x)$", in terms of a well-founded relation R, and of a functional F of dependent type "$\forall x : A.\ (\forall y : A.\ R\,y\,x \Rightarrow B\,y) \Rightarrow B\,x$". Note that the functional F already carries the information that recursive calls have to be made on smaller arguments. There is thus no separation between the definition and the proof of termination. The fixpoint equation can be derived through the following theorem, whose hypothesis is a form of contraction condition:

$$\begin{array}{l}\Big(\forall x : A.\ \forall f_1 : (\forall x : A.\ B\,x).\ \forall f_2 : (\forall x : A.\ R\,y\,x \Rightarrow B\,x).\\ \quad \big(\forall y : A.\ \forall\, h : (R\,y\,x),\ \ f_1\,y = f_2\,y\,h\big)\ \Rightarrow\\ \qquad F\,x\,(\lambda y : A.\,\lambda h : (R\,y\,x).\,(f_1\,y)) = F\,x\,(\lambda y : A.\,\lambda h : (R\,y\,x).\,(f_2\,y\,h))\Big)\\ \forall x : A.\ (\mathsf{recwf}\,R\,F\,x) = F\,x\,(\lambda y : A.\,\lambda h : (R\,y\,x).\ \mathsf{recwf}\,R\,F\,y)\end{array}$$

Two proofs of this theorem are given: a one line proof that relies on the axiom of functional extensionality, and a three-page long proof of the same theorem that does not use this axiom, and that applies only to a certain class of functions (this class is described in the next paragraph).

The `Function` feature of Coq [5] aims at providing some facility in Coq for constructively defining non-structural recursive functions. It is based on a combination of the above work [1] with the work of Barthe *et al* [2]. When defining a recursive function, the tool produces a set of proof obligations that entails termination, and then generates a fixed point equation, as well as an induction principle which may be used to prove properties of the function. The implementation of the fixed point relies on the construction of a function defined by well-founded recursion, whose body refines the body of the original function so as to admit the following type:

$$\forall x : A.\ \{z : B \,|\, \exists\, m : \mathsf{nat},\ \ \forall\, n : \mathsf{nat},\ \ n > m\ \Rightarrow\ z = \mathsf{approx}_F\, n\, x\}$$

This means that for any argument x, the function returns a value z equal to the result of evaluating the n-th approximation (as defined in §5) of the fixed point of F on the argument x, for any n above a certain rank m. `Function` applies only to function built as a pure pattern-matching tree with applications only at the end of each branch (in other words, the argument of a function cannot be the result of a pattern matching). Nevertheless, as far as we understand, this restriction would not be required if the axiom of functional extensionality was admitted. Note that the approach does not support nested recursion, nor higher-order recursion, and that the current implementation does not support mutual recursion, nor dependent case analysis in the function body.

Sozeau emphasizes the interest of using *subset types* [19] for type-checking a generic fixed point combinator. The subset type $\{x : A \,|\, P\,x\}$ is the type of values of type A that satisfies the predicate P of type "$A \to \mathsf{Prop}$". Any value of this type can be coerced by the first projection π_1 towards a value of type A, and, reciprocally, any value a of type A can be coerced to a value of type $\{x : A \,|\, P\,x\}$ given a proof of "$P\,a$". One strength of subset types is that the specification component is always erased through program extraction, thus allowing for the production of efficient functional programs. Subset types can be used to state a general recursion principle based on well-founded recursion. Given a well-founded relation R on values of type A, and given a dependent type B, a recursive function of type "$\forall x : A. (B\,x)$" can be built using a generic fixed point combinator of type:

$$\big(\forall x : A.\ \big(\forall y : \{y : A \,|\, R\,y\,x\}.\ B\,(\pi_1\,y)\big)\ \Rightarrow\ B\,x\big)\quad\Rightarrow\quad \forall x : A. (B\,x)$$

This technique is implemented by the (new) `Program` feature of Coq. In practice, the decreasing hypotheses of the form "$R\,y\,x$" can be discharged as auxiliary lemmas rather. There are two ways to use this technique. On the one hand, one can state through the type B the entire specification of the function. In this case, the proof of correctness of the function, including its termination, is to be established at definition time. On the other hand, one can assign a non-dependent type to the function (of the form "$A \to B$"), that is, not specifying the result in the type, and then derive a fixed point equation. This fixed point equation can be derived easily, though the proof requires two axioms: Functional Extensionality and Proof Irrelevance. Defining functions using higher-order or nested recursion requires putting the inductive invariant in the type and making use of the first technique. Mutual recursion is currently not supported, though it should be possible to use the encoding based on sum types.

Another approach to reasoning on recursive functions consists in writing functions in a setting that allows non-terminating programs, and then prove particular functions to be terminating. Capretta [4] and Megacz [13] make use of a co-inductive monad, where functions have a monadic type of the form "$A \to$

$\mathsf{cmp}\, B$". Others have suggested instead to extend the type theory with a type of partial functions "$A \rightharpoonup B$" (see, for example, Bove and Capretta's work [3]).

7 Conclusion

We have presented a simple, lightweight and effective technique for building recursive functions in a constructive way. It applies to a very large class of recursive functions, including those with complex recursion patterns. The function produced can be unfolded at any time using the fixed point equation, and moreover they are executable. To carry out the proof of termination, one simply investigates the body of the function and proves that recursive calls are applied to smaller values, with respect to a well-founded relation or a measure. This process is intuitive, and does not require the use of any advanced technique. It has been entirely implemented in Coq, and experimented on a number of classical examples. We hope this work will help Coq users in the task of defining nontrivial recursive functions.

References

[1] Antonia Balaa and Yves Bertot. Fix-point equations for well-founded recursion in type theory. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *LNCS*, pages 1–16. Springer, 2000.

[2] Gilles Barthe, Julien Forest, David Pichardie, and Vlad Rusu. Defining and reasoning about recursive functions: A practical tool for the coq proof assistant. In Masami Hagiya and Philip Wadler, editors, *FLOPS*, volume 3945 of *LNCS*, pages 114–129. Springer, 2006.

[3] Ana Bove and Venanzio Capretta. A type of partial recursive functions. In Otmane Aït Mohamed, César Muñoz, and Sofiène Tahar, editors, *TPHOLs*, volume 5170 of *LNCS*, pages 102–117. Springer, 2008.

[4] Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[5] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version 8.2*, 2009.

[6] Jürgen Giesl. Termination of nested and mutually recursive algorithms. *Journal of Automated Reasoning*, 19(1):1–29, August 1997.

[7] John Harrison. Inductive definitions: Automation and application. In E. Thomas Schubert, Phillip J. Windley, and Jim Alves-Foss, editors, *TPHOLs*, volume 971 of *LNCS*, pages 200–213. Springer, 1995.

[8] John Harrison. *The HOL Light Manual*, 2009. Version 2.2.

[9] Peter T. Johnstone. *Notes on logic and set theory.* Cambridge mathematical textbooks. Cambridge University Press, 1987 (1992 [printing]).

[10] Alexander Krauss. Partial recursive functions in higher-order logic. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *LNCS*, pages 589–603. Springer, 2006.

[11] Sava Krstić and John Matthews. Inductive invariants for nested recursion. In David A. Basin and Burkhart Wolff, editors, *TPHOLs*, volume 2758 of *LNCS*, pages 253–269. Springer, 2003.

[12] John Matthews. Recursive function definition over coinductive types. In Yves Bertot, Gilles Dowek, André Hirschowitz, C. Paulin, and Laurent Théry, editors, *TPHOLs*, volume 1690 of *LNCS*, pages 73–90. Springer, 1999.

[13] Adam Megacz. A coinductive monad for prop-bounded recursion. In Aaron Stump and Hongwei Xi, editors, *PLPV*, pages 11–20. ACM, 2007.

[14] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[15] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[16] Konrad Slind. Function definition in higher-order logic. In Joakim von Wright, Jim Grundy, and John Harrison, editors, *TPHOLs*, volume 1125 of *LNCS*, pages 381–397. Springer, 1996.

[17] Konrad Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, Institut für Informatik, Technische Universität München, 1999.

[18] Konrad Slind. Another look at nested recursion. In Mark Aagaard and John Harrison, editors, *TPHOLs*, volume 1869 of *LNCS*, pages 498–518. Springer, 2000.

[19] Matthieu Sozeau. Subset coercions in coq. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *LNCS*, pages 237–252. Springer, 2006.