Fixed point semantics and partial recursion in Coq

Yves Bertot, Vladimir Komendantsky

Fixed point semantics and partial recursion in Coq

Yves Bertot and Vladimir Komendantsky*

INRIA Sophia Antipolis

Abstract. We propose to use Knaster–Tarski least fixed point theorem as a basis to define recursive functions in the Calculus of Inductive Constructions. This widens the class of functions that can be modeled in type-theory based theorem proving tools to potentially non-terminating functions. This is only possible if we extend the logical framework by adding some axioms of classical logic. We claim that the extended framework makes it possible to reason about terminating or non-terminating computations and we show that extraction can also be extended to handle the new functions.

1 Introduction

For theoretical computer scientists, Knaster–Tarski least fixed point theorem, as well as its application – the so-called first fixed point theorem of Kleene [27], is a firm theoretical ground to assert the existence of objects defined by recursive equations. These objects can be inductive types or recursive functions as in [15,14]. We consider the following generalized statement of Knaster–Tarski theorem [1]:

Theorem 1 (complete partial order version of Knaster–Tarski). *Given a monotone function f on a chain-complete partial order, consider the following transfinite sequence:*

$$\begin{aligned} x_o &= \bot \\ x_{\alpha+1} &= f(x_\alpha) \\ x_\beta &= \text{ the least upper bound of the chain } \{f(x_\alpha)\}_{\alpha<\beta} \text{ if } \beta \text{ is a limit ordinal.} \end{aligned}$$

The function f has a least fixed point. Moreover, if f is continuous then the least fixed point is obtained in at most ω iterations.

To use this theorem, one should be able to express that the domain of interest has the required completeness property and that the function being considered is continuous. If the goal is to define a partial recursive function then this requires using axioms of classical logic, and for this reason the step is seldom made in the user community of type-theory based theorem proving. However, the constructive prejudice is not a necessity: adding classical logic axioms to the constructive

* This work was partially supported by the French ANR Project CompCert

logic of type theory can often be done safely to retain the consistency of the whole system.

In this paper, we work in classical logic to reason about potentially non-terminating recursive functions. No inconsistency is introduced in the process, because potentially non-terminating functions of type $A \to B$ are actually modeled as functions of type $A \to B_\perp$: the fact that a function may not terminate is recorded in its type, non-terminating computations are given the value $\perp$ which is distinguished from all the regular values, and one can reason classically about the fact that a function terminates or not. This is obviously non-constructive but does not introduce any inconsistency.

One of the advantages of type-theory based theorem proving is that actual programs can be derived from formal models, with guarantees that these programs satisfy properties that are predicted in formally verified proofs. This derivation process, known as extraction [24,17], performs a cleaning operation so that all parts of the formal models that correspond to compile-time verifications are removed. The extracted programs are often reasonably efficient. In this paper, we also show that extraction can accommodate the new class of potentially non-terminating functions.

When axioms are added to the logical framework, three cases may occur: first, the new axioms may make the system inconsistent; second, the new axioms may be used only in the part of the models that is cleaned away by the extraction process; third, the axioms may be used in the part that becomes included into the derived programs. We don't really need to discuss the first case that should be avoided at all costs. In the second case, the extraction process still produces a consistent program, with the same guarantee of termination even if this guarantee relies on classical logic reasoning steps. In the third case, the added axiom needs to be linked to a computation process that implements the behavior predicted by the axiom. We claim that this can be done safely if Knaster–Tarski least fixed point theorem is given a sensible computational content.

Kleene's least fixed point theorem can be used to justify the existence of recursive functions, because these functions can be described as the least fixed point of the functional that arises in their recursive equation. However, it is necessary to ensure that the function space has the properties of a complete partial order and that the functional is continuous. These facts can be motivated using a simple development of basic domain theory. With the help of the axiom of *definite description*, this theorem can be used to produce a function, which we shall call fixp, that takes as argument a continuous function and returns the least fixed point of this function. When the argument of fixp is a functional, the least fixed point is a recursive function, which can then be combined with other functions to build larger software models.

With respect to extraction, we also suggest a few improvements to the extraction process that should help making sure that fairly efficient code can be obtained automatically from the formal models studied inside our extension of the Calculus of Inductive Constructions.

From the formal proof point of view, Knaster–Tarski least fixed point theorem provides us with two important properties of the function it produces. The first property is that the obtained function satisfies the fixed point equation. The importance of fixed point equations is straightforward and was already described in [3]. This fixed point equation is useful when we want to prove properties of the function, for instance, that under some conditions it is guaranteed to terminate. The second property is that the least fixed point is obtained in at most ω iterations. As a result, we can reason by induction on the length of computations, thus providing a poor man's approximation of what is called *fixed point induction* in [27]. This possibility allows to prove properties of the function result when it exists, and can also be used to prove that under some conditions a function fails to terminate.

In this paper, we recapitulate our basic formalization of domain theory. Then, we show how this theory can be used in the definition of simple recursive functions. In particular, we give an example that contains proofs about recursive function as a support to discuss the techniques that are available. Next, we discuss matters concerned with extraction and execution of the recursive programs that are obtained in this way. Related work and opportunities for further extensions are reviewed at the end of the paper. All the experiments described in this paper were done with Coq [11,5] and can be found on the Internet from the first author's web page [6].

2 Basic notions

In this section, we follow the lines of our Coq development on domain theory. We pursue the approach proposed in [23] to define various types of records for classes of functions with given properties.

2.1 Building complete preorders of continuous functions

We define a *preorder* O to be a record containing a carrier element $A : \mathsf{Type}$ and a reflexive and transitive relation on A, denoted $\leq_O$. All preorders form a type ord. For any preorder O, the *derived equality* is the relation, denoted $==_O$, such that, for any $x\ y : O$, $x ==_O y$ whenever $x \leq_O y$ and $y \leq_O x$.

For A a type and O a preorder, we define the *pointwise order relation* $\leq_{A\to O}$ on functions $A \to O$ to be such that, $f \leq_{A\to O} g$ whenever, for all $x : A$, $f\ x \leq_O g\ x$. This relation can be proven to be a preorder. Hence we construct a preorder (a *function space*) on the type $A \to O$. We denote this space by $A \xrightarrow{o} O$.

As an instance of the ord type, we construct the ordered lifting of the standard type nat of natural numbers in Coq with respect to the standard relation $\leq$ on nat. We call this lifting $\mathsf{nat\_ord}$.

For O_1 and O_2 preorders, we define the type of monotonic functions from $f : O_1$ to O_2 as the type of records containing a function and a proof that f is monotonic. The pointwise preorder on functions can be lifted to monotonic

functions, so that we can construct a preorder of monotonic functions. This preorder is denoted $O_1 \stackrel{m}{\to} O_2$.

Now we can use convenient notation to define a *chain* on a preorder O as an element of $\mathsf{nat\_ord} \stackrel{m}{\to} O$. Chains play an important role in the implementation of complete preorders below.

In many cases it is helpful to have notation for partial evaluation of a function depending on more than one arguments. For a function $f : O_1 \stackrel{m}{\to} (O_2 \stackrel{m}{\to} O_3)$, *application* of f to $x : O_2$ is defined to be the monotonic function $\lambda y.fyx$ and denoted $f \_ x$.

Next we define the *composition* of monotonic functions $f : O_1 \stackrel{m}{\to} O_2$ and $g : O_2 \stackrel{m}{\to} O_3$ to be the monotonic function, denoted $g@f$, such that

$$(g@f)\ x = g\ (f\ x)\ .$$

A *complete preorder* is a dependent record consisting of a preorder O, an element $\bot : O$, and a least upper bound function $\mathsf{lub} : (\mathsf{chain}\ O) \to O$ satisfying the three laws below:

$$\forall x : O, \bot \leq x$$
$$\forall (c : \mathsf{chain}\ O)(n : \mathsf{nat\_ord}), c\ n \leq \mathsf{lub}\ c$$
$$\forall (c : \mathsf{chain}\ O)(x : O), (\forall n : \mathsf{nat\_ord}, c\ n \leq x) \to \mathsf{lub}\ c \leq x$$

In the present paper we abbreviate "complete preorder" as "cpo" as in [23].

Among the immediate properties of lub are monotonicity and precontinuity, given respectively by

$$\forall (D : \mathsf{cpo})(c\ c' : \mathsf{chain}\ D),\ c \leq c' \to \mathsf{lub}\ c \leq \mathsf{lub}\ c'\ ,$$
$$\forall (D_1\ D_2 : \mathsf{cpo})(f : D_1 \stackrel{m}{\to} D_2)(c : \mathsf{chain}\ D),\ \mathsf{lub}(f@c) \leq f(\mathsf{lub}\ c)\ .$$

For cpos D_1 and D_2, a function $f : D_1 \stackrel{m}{\to} D_2$ is *continuous* whenever, for any chain c on D_1, $f(\mathsf{lub}\ c) \leq \mathsf{lub}(f@c)$. As before, we define the type of a continuous functions from D_1 to D_2 as the type of records containing a function $f : D_1 \stackrel{m}{\to} D_2$ and a proof that f is continuous. There is the obvious pointwise preordering relation and we use it to construct a preorder on continuous functions. This preorder is denoted $D_1 \stackrel{c}{\to} D_2$.

For a type A and a cpo D, the cpo $A \stackrel{O}{\to} D$ of functions from A to D is constructed naturally by defining $\bot_{A \stackrel{O}{\to} D}$ to be $\lambda x : A.\bot_D$ and $\mathsf{lub}_{A \stackrel{O}{\to} D}$ to be $\lambda x : A.\mathsf{lub}(c \_ x)$. It is rather straightforward to obtain the required proofs for the three cpo laws.

For a preorder O and a cpo D, the cpo $O \stackrel{M}{\to} D$ of monotonic functions from O to D is constructed by reusing the structure of the cpo $O \stackrel{O}{\to} D$. The extra work is to prove that $\lambda x.\bot_D$ and $\lambda x.\mathsf{lub}_D(c \_ x)$ are monotonic. Changing continuity for monotonicity, the same process is applied again to define the cpo of continuous functions from a cpo D_1 to a cpo D_2, denoted $D_1 \stackrel{C}{\to} D_2$.

2.2 The flat preorder

We define the following inductive type which turns out to be isomorphic to the standard option type of Coq:

```
Inductive pointed (A : Type) : Type :=
  in_pointed : A → pointed A | bottom : pointed A.
```

The *flat preorder* on a type A is defined by specifying a binary relation $\leq_{\mathsf{pointed}\ A}$ such that, for $x\ y : A$, $x \leq_{\mathsf{pointed}\ A} y$ iff $x = y$ or $x = \mathsf{bottom}$. We denote this flat preorder by &ord A.

Lifting of the flat preorder to a flat cpo requires a non-constructive definition of the lub function. Namely, using the excluded middle law of classical propositional logic we can prove $\leq_{\&\mathsf{ord}\ A}$ being complete in a sense that, for each chain on &ord A, there exists an x such that $\forall n, c\ n \leq_{\&\mathsf{ord}\ A} x$ and $\forall y, (\forall n, c\ n \leq_{\&\mathsf{ord}\ A} y) \to x \leq_{\&\mathsf{ord}\ A} y$, which proves the two laws for the required least upper bound function. Since we can prove that this least upper bound is unique, using the classical definite description axiom found in Coq libraries, we obtain a Σ-type definition of the least upper bound of $c : \mathsf{chain}\ A$ that contains two parts: an element $a : \&\mathsf{ord}\ A$ and the proof of a being the least upper bound of c. In this way we obtain the function $\mathsf{lub}_{\&\mathsf{ord}\ A} : \mathsf{chain}\ (\&\mathsf{ord}\ A) \to \&\mathsf{ord}\ A$ as the first projection of this Σ-type object. Thus we have a cpo structure on &ord A; we denote it by &cpo A.

Type theory specialists may note that one should be careful when using the definite description axiom because it is incompatible with variants of the Calculus of Constructions where the Set sort is impredicative [9]. Fortunately, this is not the default for Coq and our work is done with predicative Set.

3 Proving the fixed point theorem

The well-known fixed point theorem of Kleene, see [27], has a mild generalization in the setting of complete preorders. The statement of the theorem we are interested in is the following:

Theorem 2 (complete preorder version of Kleene's fixpoint). *In a complete preorder, every continuous function has a least fixed point for the derived equality.*

Below we outline a formalized proof for this statement. Our proof follows the lines of the classical textbook proofs found in, e.g., [19,27]. The fixed point functional defined for this proof can and will be used in this paper to define partial recursive functions and reason about them.

The construction of the fixed point functional is closely related to the one given in [23]. We consider a cpo D and a function $f : D \stackrel{m}{\to} D$. First, we define a function f_iter as follows:

```
Fixpoint f_iter (n:nat_ord) : D := match n with 0 ⇒ ⊥ | S n' ⇒ f (f_iter n') end.
```

Then we prove monotonicity of f_iter and define iter of type chain D to be f_iter with the attached proof of monotonicity; and we define the function f_fixp to be the least upper bound of this chain.

```
Definition f_fixp : D := lub iter.
```

For a continuous function f, we can prove the derived fixed point property f_fixp == f f_fixp. Next, for any complete preorder D, we define the required fixed point functional which is a continuous version of f_fixp, and we also have the corresponding fixed point property.

```
Definition fixp (D:cpo) : (D -C-> D) -C-> D := ...
Lemma fixp_eq : ∀ D (f:D -C-> D), fixp f == f (fixp f).
```

Therefore Theorem 2 can be formalized as follows:

Theorem 2 (formal Coq statement).

```
∀ (D:cpo)(f:D -C-> D), f (fixp f) == fixp f ∧ (∀ x, f x ≤ x → fixp f ≤ x).
```

The fixed point returned by fixp is the least by construction; it is the least upper bound of the iter chain, which allows reasoning on partial recursive functions.

4 Using the fixed point theorem to define recursive functions

To model partiality of a function f_0 with arguments of type A and values of type B, first we define a recursive function $f : A \to \&\mathsf{cpo}\ B$ for which we ought to construct a continuous functional F of type

$$(A \xrightarrow{O} \&\mathsf{cpo}\ B) \xrightarrow{C} (A \xrightarrow{O} \&\mathsf{cpo}\ B)$$

such that $f = Ff$. A proof of such a functional F being actually continuous is usually non-trivial but considerably regular. For a proof, one might use the following intuition: The condition of continuity of F corresponds to the interpretation of "potential non-termination" according to which every expression containing a potentially non-terminating computation should fail to terminate if it actually uses the value returned by this computation and that computation fails to terminate. To use the value of a potentially non-terminating computation one needs to write a pattern-matching construct on this computation: the continuity condition will be satisfied if we ensure that the $\bot$ value is returned in the $\bot$ case of this matching construct.

For example, consider the minimisation functional μ defined as follows: for all A : Type, $f : A \times \mathsf{nat} \to \mathsf{nat}$, the value of μf is a function $g : A \to \mathsf{nat}$ such that $g(x)$ is defined and has value y if and only if y is the least value for which $f(x, y) = 0$ holds. From this definition is follows that $g(x)$ is undefined in case no least value y is found, that is μ can be used to define partial functions.

Let A:Type and f:A → nat → pointed nat. First, we specify a functional f_mu as follows:

```
Definition f_mu (mu : A → nat → pointed nat) : A → nat → pointed nat :=
  fun x y ⇒
    match f x y with
      bottom ⇒ bottom
    | in_pointed 0 ⇒ in_pointed y
    | _ ⇒ mu x (S y)
    end.
```

Then we prove monotonicity and then continuity of f_mu and specify the functions mono_mu and cont_mu with the proofs of monotonicity and, respectively, continuity attached as follows:

```
Definition mono_mu :
  (A -o> nat_ord -o> &ord nat) -m> (A -o> nat_ord -o> &ord nat) := ...
Definition cont_mu :
  (A -O> nat_ord -O> &cpo nat) -C> (A -O> nat_ord -O> &cpo nat) := ...
```

Once the continuity proof is completed, we can define the functional with a command of the following form:

```
Definition mu := fixp cont_mu.
```

Now we can illustrate the use of our mu. Consider the function $\lambda xy.|x - y^2|$ with the following definition (note the use of truncated subtraction):

```
Definition abs_x_minus_y_squared (x y : nat) :=
  in_pointed ((x - y*y) + (y*y - x)).
```

The value of $\mu(\lambda xy.|x - y^2|)k$ is defined if and only if k is a perfect square. This can be defined in Coq as follows:

```
Definition perfect_sqrt (x:nat) := mu abs_x_minus_y_squared x 0.
```

It should be also instructive to consider here a few methods that one could use to prove continuity of functionals in our setting. Functionals can usually be understood as composites of elementary continuous functions, and composition can be shown to preserve continuity.

For instance we can define a continuous test function for tests on the type bool:

```
Definition f_cond (A:Type)(t:&ord bool)(x y:&ord A) : &ord A :=
  match t with in_pointed true ⇒ x | in_pointed false ⇒ y | bottom ⇒ bottom end.
```

Next, by proving monotonicity and then continuity in each of the arguments of this function, we arrive at its continuous version:

```
Definition cond (D:cpo) : (&cpo bool) -C> D -C> D -C> D := ...
```

Speaking informally, the function cond is essentially the same as f_cond but also provided with a proof of continuity in each of the arguments. We believe that the same work can be done systematically for the pattern matching constructs that are associated with any other basic recursive type.

We can also define a function f_Apply that mimics the application of a potentially non-terminating function to a value, also computed by a potentially non-terminating function. Below is a possible definition:

```
Definition f_Apply (A B:Type)(f:&ord (A→ &ord B))(x:&ord A) : &ord B :=
  match x with
    in_pointed y ⇒ match f with in_pointed g ⇒ g y | bottom ⇒ bottom end
  | bottom ⇒ bottom
  end.
```

This function can also be provided with a continuity statement and defined as a family of cpos of continuous functions as follows:

$$\mathsf{Definition\ Apply\ (A\ B{:}Type)} : (\&\mathsf{cpo}\ (A \to \&\mathsf{cpo}\ B)) \xrightarrow{C} (\&\mathsf{cpo}\ A) \xrightarrow{C} (\&\mathsf{cpo}\ B) := \ldots$$

Notice that the definition of the function Apply suggests computing with our potentially non-terminating functions in a call-by-value fashion, since the application of a function to an argument would fail if the computation of one of the arguments is non-terminating. A different approach would be needed to model execution of programs by the call-by-name strategy. However, we feel it is quite satisfactory that we can describe precisely when a program fails to terminate for a given execution strategy.

5 Proving properties of functions

In [27], Winskel describes *fixed point induction* as a helpful tool to reason about recursive functions. However, this style of induction is restricted to certain predicates which are called *refining* in his work. The same notion also appears in HOLCF, see [26,18], under the name of *admissible* predicates and the authors argue that it is important to provide strong automation facilities to manage the corresponding proofs of admissibility. Our work is less advanced than HOLCF, but nevertheless we can perform some proofs when the recursive functions that we consider have a flat target type.

In our setting, we want to prove properties of functions obtained using fixp and we have two tools at hand. The first tool is the lemma fixp_eq. The second tool is a theorem that can only be used when the recursive function has a flat preorder as the codomain. The theorem states that for any input, the value of fixp f can also be computed by iter f n for some natural number n:

$$\mathsf{Lemma\ fixp\_flat\_witness} :\ \forall\ A\ B\ (f : (A \xrightarrow{O} \&\mathsf{cpo}\ B) \xrightarrow{C} (A \xrightarrow{O} \&\mathsf{cpo}\ B))\ x, \exists\ n, \mathsf{fixp}\ f\ x = \mathsf{iter}\ f\ n\ x.$$

The number n can intuitively be understood as an upper bound of the number of recursive calls that are needed to compute the value at x. Thanks to this theorem, we can reason by induction on n and simulate the fixed point induction of [27].

For instance, using fixp_flat_witness we can conclude with the following lemma:

```
Lemma perfect_sqrt_bottom :
 ∀ x, (∀ y:nat, ∼x = y*y) → perfect_sqrt x = bottom.
```

This lemma asserts that the function perfect_sqrt never terminates on inputs which are not perfect squares. As part of the proof of this lemma, we first prove the following statement by induction on n:

```
∀ n x, (∀ y:nat, ∼x = y*y) → iter (mono_mu abs_x_minus_y_squared) n x 0 = bottom.
```

The below two lemmas relate computations done with iter and values of one- and two-argument recursive functions:

```
Lemma iter_in_pointed_eq_fixp :
 ∀ A B (f : (A -O-> &cpo B) -C-> (A -O-> &cpo B)) x n v,
   iter f n x = in_pointed v → fixp f x = in_pointed v.
Lemma iter_in_pointed_eq_fixp_2 :
 ∀ A B C (f:(A -O-> B -O-> &cpo C) -C-> (A -O-> B -O-> &cpo C)) x y n v,
   iter f n x y = in_pointed v → fixp f x y = in_pointed v.
```

Using these lemmas, we can compute values of recursive functions, provided that none of these values is bottom. We simply need to guess the right argument n that leads to a definite value of the form in_pointed v and in this case we know the value of the recursive function for the corresponding argument. Of course, the problem is to guess the right number n. If we choose a value that is too small, the value returned by the iterative process is the uninformative bottom.

The below example uses the lemma iter_in_pointed_eq_fixp_2:

```
Lemma compute_perfect_sqrt_36 : perfect_sqrt 36 = in_pointed 6.
Proof.
unfold perfect_sqrt; unfold mu.
apply iter_in_pointed_eq_fixp_2 with (n:=100); reflexivity.
Qed.
```

The number 100 used in this example only needs to be an upper bound of the number of recursive calls needed to compute perfect_sqrt 36 (in this case 7 is enough). This approach may be used in reflexive tactics for example, as some proof may require computing the value of a recursive function. We can try with a fixed number of calls. If a value of the form in_pointed is returned, the proof can proceed, otherwise the tactic fails, which does not mean that the recursive function diverges.

6 Extraction to functional programming languages

The function perfect_sqrt from Section 4 can be successfully extracted using the extraction mechanism provided by Coq. The extracted code exhibits the expected partial behavior, that is it loops exactly on arguments which are not perfect squares. Below we show how this can be made possible.

We rely on the Classical and ClassicalDescription extensions of the Coq libraries. These extensions only add two axioms to the Coq logic: the axiom of excluded middle, and the axiom of "constructive definite description". The statement of the latter is as follows:

```
Axiom constructive_definite_description :
  ∀ A (P : A→Prop), (∃! x : A, P x) → { x : A | P x }.
```

Obviously, once this axiom is used, the distinction between purely logical and purely informative data is blurred. The left hand side of the implication states that we know there exists an x satisfying a property P but we don't know how to construct it, the right-hand side says that we can use a value that satisfies P. Using this axiom, we eliminate the distinction between "constructive" values and "logically existing and unique" values, but this distinction plays a central role in the extraction mechanism of Coq.

In principle, the extraction mechanism relies on the fact that, for every element of the form $\{x : A \mid P\ x\}$, there exists a constructive procedure for obtaining the x part. However, the axiom produces elements of that form without providing any constructive procedure. For this reason, the extraction mechanism expects the user to handwrite the procedure that returns the expected value wherever the axiom is used. The process cannot be mechanized and one cannot help extraction at this level.

We propose the following solution to this problem. The constructive definite description axiom is only used in the definition of the function fixp, to transform the existential statement of Theorem 1 into a value that can be used in other functions. We can provide a handwritten constructive content for the function fixp, so that the logical value of this function is not used, and hence make sure that the definite description axiom is never used in the extracted code. We simply need to choose a constructive procedure for fixp that can be written in the target language of extraction and whose behavior corresponds to the behavior described in the Coq development.

We construct a function fix that computes the fixpoint of functionals, so that this function should satisfy the following equality:

```
f (fix f) = fix f
```

Reversing the equation seems enough to do the trick:

```
let rec fix f = f (fix f)
```

However, this is not satisfactory in a call-by-value language, since this code directly attempts to compute fix f again and enters a looping computation. This

can be corrected if the expression is encapsulated in a λ-expression and hence the execution is stalled. Using OCaml syntax this is done in the following manner:

```
let rec fix f = f (fun y → fix f y)
```

The function fix is the function we propose to attach to the function fixp for extraction purposes. Obviously this imposes a restriction: while in Coq, we can use the function fixp to model fixed points even in the case of first order functions; in the extracted code, fix can only be applied to a higher-order function f. Therefore the pair fixp–fix should only be used to define recursive functions.

In the Coq system, the directive to perform this binding of a Coq function and an OCaml value is as follows:

```
Extract Constant fixp ⇒ "let rec fix f x = f (fun y → fix f y) x in fix".
```

There is a leap of faith in this binding, which is as strong as the leap of faith one does when using extra axioms. In our current understanding of this extraction strategy, we can give a partial correctness result: when computation terminates and returns a first-order value, the result is still predicted by the Coq model.

Theorem 3. *If the extracted code for* fixp f a *is some expression* fix f' a', *and the computation of* fix f' a' *terminates in the target language, then the expression* fixp f a *can be proved to terminate with the corresponding value in Coq.*

Proof. Assume that the extraction process behaves correctly for expressions that do not contain fixp. We reason by induction on the number of execution steps in the execution of fix f' a'. When executing fix f' a', the first steps lead to execution of f' (fun y → fix f' y) a, and the subsequent steps concern execution of a and f'. Any execution of a fix expression occurring in f' or a uses less steps, so that these executions behave as predicted by the corresponding fixp expression in the Coq model. Moreover, execution of (fun y → fix f' y) e has the same behavior as execution of fix f' e. Together with the assumption that the extracted code in f' and a' outside of fix expressions behaves as expected, the latter consideration ensures the property. □

The extracted code can be improved in two ways, based on a few basic observations. In our model of recursive functions, the value bottom is only used for functions that are undefined because they fail to terminate (and hence bottom expresses *divergence*). We say that a Coq expression contains a *spontaneous divergence* if it contains an instance of bottom which is not encapsulated inside a pattern-matching construct on the type pointed, or if it appears in any of the parts e and e_2 of a pattern matching construct of the following shape:

match e with bottom ⇒ e_1 | in_pointed ⇒ e_2 end

In our approach to modeling recursive functions, the functions we produce in Coq always satisfy the property that they contain no spontaneous divergence. We can also assume that programs extracted to OCaml or Haskell inherit the

corresponding property from the initial Coq functions. We will now discuss optimisations that can be performed for this class of programs.

We have the following result:

Theorem 4. *If an expression contains no spontaneous divergence then its value can never be the value* bottom.

This can be proved, by induction on the length of an execution of this expression. The only sub-expressions that can produce the bottom value are the bottom expressions that appear in the bottom branch of pattern-matching constructs. But in these pattern-matching constructs, the matched expression (named e in the example above) also has the property of containing no spontaneous divergence and we can use an induction hypothesis. This concludes the proof.

We verified a formal Coq model of this informal proof in the context of the Mini-ML language [10] extended with a data-type representing the type pointed and the corresponding pattern-matching construct [6].

If bottom values can never be produced then the operations of encapsulating expressions inside the constructor in_pointed and the operations of removing the constructor in_pointed done by pattern-matching seem to be useless. They can be deleted from the program, and the type pointed can be deleted from the program. In other words, every instance of in_pointed e can be replaced with e and every instance of the construct

$$\mathsf{match}\ e\ \mathsf{with}\ \mathsf{bottom} \Rightarrow e_1\ |\ \mathsf{in\_pointed}\ x \Rightarrow e_2$$

can be replaced with $\mathsf{let}\ x = e\ \mathsf{in}\ e_2$. In what follows, we denote by e' the result of the transformation of e.

Theorem 5. *If e is an expression containing no spontaneous divergence and the value of executing e is v then the value of executing e' is v'.*

As in Theorem 4, the statement can be proved by induction on the length of an execution of e. We actually prove a more general statement which also includes the environments in which e and e' are executed: we need to assume that the expression e is executed in an environment ρ and the expression e' is executed in an environment ρ', so that the value associated to each variable in ρ' is obtained by applying the transformation to the corresponding value in ρ.

For example, if the value of e is in_pointed 3 then the value of e' is 3. If e is fun x → in_pointed a, and the current environment binds a with in_pointed 3 and b with 4, then the value of e is the closure that can be written

$$\langle \mathsf{fun\ x} \rightarrow \mathsf{in\_pointed\ a}, (\mathsf{a}, \mathsf{in\_pointed}\ 3) \cdot (\mathsf{b}, 4) \rangle$$

In this case, e' is fun x → a, the transformed environment binds a with the value 3 and b with 4, and the result of the evaluation of e' is

$$\langle \mathsf{fun\ x} \rightarrow \mathsf{a}, (\mathsf{a}, 3) \cdot (\mathsf{b}, 4) \rangle$$

The two most important cases of the proof concern transformations of e:

1. If e has the form $\mathsf{in\_pointed}\ e_1$ then e_1 also contains no spontaneous divergence, and we can use the induction hypothesis on the evaluation of e_1. Hence if v_1 is the value of e_1, the value of e'_1 is v'_1. But in this case, we have $e' = e'_1$ and the value of e is $\mathsf{in\_pointed}\ v'_1$. However, we obviously have $(\mathsf{in\_pointed}\ v'_1)' = v'_1$, which justifies the result.
2. If e has the form

$$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{bottom} \rightarrow \ldots \mid \mathsf{in\_pointed}\ \mathsf{x} \rightarrow e_2$$

then e' is the expression

$$\mathsf{let}\ x = e'_1\ \mathsf{in}\ e'_2$$

If v_1 is the value of e_1, we know that v_1 necessarily has the shape $\mathsf{in\_pointed}\ w$ for some w, because e_1 contains no spontaneous divergence and therefore the result cannot be bottom. Therefore the result of evaluating e is the result of evaluating e_2 in the environment $(x, \mathsf{in\_pointed}\ w) \cdot \rho$. We call this value v_2. By induction hypothesis on the evaluation of e_1, we have that the result of evaluating e'_1 is $v'_1 = (\mathsf{in\_pointed}\ w)' = w'$. Hence the result of evaluating e' is the result of evaluating e'_2 in $(x, w') \cdot \rho' = ((x, \mathsf{in\_pointed}\ w) \cdot \rho)'$. By induction hypothesis, this evaluation yields v'_2.

A model of this proof was formally verified using the Mini-ML description of the language. This proof is available in [6].

7 Related work

The work described here contributes to all the work that was done to ease the description and formal proofs about general recursive functions. A lot of efforts were put in providing relevant collections of inductive types with terminating computation derived from the notion of primitive recursion [8,22,2]. In particular, it was shown that the notion of accessibility or noetherian induction could be described using an inductive predicate in [21]. This accessibility predicate makes it possible to encode well-founded recursion when one can prove that all elements of the input type satisfy the accessibility predicate for a well-chosen relation (such a relation is called well-founded). If it is not true that all elements are accessible (or if one cannot exhibit a well-founded relation that suits the function being defined), the recursive function may still be defined but have well-defined values only for the elements that can be proved to be accessible. This idea was further refined in [12,7], where termination is not described using an accessibility predicate, but directly with an inductive predicate that actually describes exactly those inputs for which the function terminates. As a result, formal developers still need to prove that a potentially non-terminating function is only used when termination is guaranteed and extracted programs inherit the termination guarantee. By contrast, the approach of this paper relieves the developer from the burden of proving termination and does not guarantee it either, which can be useful for for some applications, like interpreters for Turing-complete

languages (where termination is undecidable) or functions for which potential non-termination is an accepted defect.

Another work [3,5] attempts to provide tools that stay closer to the level of expertise of programmers in conventional functional programming. The key point is to start from the recursive equation and to generate the recursive function definition from this equation. Users still need to prove that the recursive calls happen on predecessors of the initial input for a chosen well-founded relation, but these requirements appear as proof obligations that are generated as a complement of the recursive equation. The tool produces the recursive function and a proof of the recursive equation. Again, this approach is restricted to total functions, but we plan to follow this work as inspiration to provide a similar tool where users can describe their program in a language that is as close as possible to a conventional programming language.

In one of our experiments [4], we defined the semantics of a small programming language in the spirit of [19,27]. We used Knaster–Tarski fixed point theorem to describe the semantics of while loops as suggested in [27]. Then we were able to prove that when a value is returned, the same computation can be modeled by a natural semantics derivation, using an encoding of the natural semantics based on an inductive predicate. This reproduces a similar formalized proof in [20]. Once extracted to ML, this gives a certified interpreter for the language.

In an early version of the Calculus of Constructions, formalizations of Knaster–Tarski least fixed point theorem were also used to show how inductive definitions could be encoded directly in the pure (impredicative) Calculus of Constructions [15]. In this respect, it is also worthwhile to mention that [14] shows how this theorem can be used to give a definitional justification of inductive types in higher-order logic.

In this paper, we formalized only the minimal amount of domain theory just enough to make it possible to define simple potentially non-terminating functions and perform basic reasoning steps on these functions. More complete studies of domain theory were performed in the LCF system [25]. It was also formalized in Isabelle's HOL instantiation to provide a package known as HOLCF [26,18]. We believe these other experiments can give us guidelines for improvements.

Our effort to formalize optimizations of the extracted code should also be compared with efforts done to give a formal description of the extraction procedure, as studied by Letouzey and Glondu [13].

8 Conclusion

There is a popular belief that type-theory based proof tools can only be used to reason about functions that are total and terminating for all inputs, because termination of reductions is needed to ensure the consistency of the systems in question. One of the aims of this paper is to confront this belief by providing yet another way to model potentially non-terminating functions. To do so we re-use Knaster–Tarski least fixed point theorem of domain theory.

In this paper, there are three claims that deserve a closer look. The first claim is that, in certain cases, users of type-theory based theorem provers can relax the constructivity requirement and accept non-constructive axioms that do not endanger the consistency of the logical system. We believe that it is high time for a comprehensive set of axioms to be designed to perform a merge of type theory (with the advantage that it contains a quite efficient reduction mechanism to compute directly in the logical framework) with classical higher-order logic (with the advantage of representing more directly the usual concepts of mathematics). We believe our paper displays a direct link between the non-constructive domain theory for computable functions and program extraction.

The second claim we make is that modeling an arbitrary recursive function is tractable in the setting we propose. In particular, proving continuity may be the real difficulty of this approach. We hope that the existing body of work on formalized domain theory can be of use here.

The third claim we make is that the first fixed point theorem of Kleene is faithfully realized by the functional value fix and that the resulting code can be optimized to remove all uses of the data-type pointed that is added in the Coq models to account for possible non-termination of functions. We provided proofs of these statements and some of these proofs have been formally verified on a simple model of functional programming language, based on Mini-ML [10].

Acknowledgments

Benjamin Werner and Hugo Herbelin played a significant role in understanding what form of the axioms of classical logic provide safe extensions of the Calculus of Inductive Constructions. This work also benefited from early experiments by Kuntal Das Barman and suggestions by P. Aczel. The first author also wishes to remember the late Gilles Kahn, who started work on formalizing domain theory in the context of the Calculus of Inductive Constructions in 1996 [16].

References

1. S. Abian and A. B. Brown. A theorem on partially ordered sets with applications to fixed point theorems. *Canadian J. Math.*, 13:78–82, 1961.
2. Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1977.
3. Antonia Balaa and Yves Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, January 2002.
4. Yves Bertot. Theorem proving support in programming language semantics, 2007. `http://hal.inria.fr/inria-00160309`.
5. Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development, Coq'art: the calculus of inductive constructions*. Texts in Theoretical Computer Science: an EATCS series. Springer-Verlag, 2004.

6. Yves Bertot and Vladimir Komendantsky. Proofs on domain theory and partial recursion, 2008. http://www-sop.inria.fr/marelle/Yves.Bertot/tarski.html.
7. A. Bove. General recursion in type theory. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, International Workshop TYPES 2002, The Netherlands*, number 2646 in Lecture Notes in Computer Science, pages 39–58, March 2003.
8. Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical report, University of Cambridge, 1992.
9. Laurent Chicli, Loïc Pottier, and Carlos Simpson. Mathematical quotients and quotient types in Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, number 2646 in LNCS, pages 95–107. Springer, 2003.
10. Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, August 1986.
11. Coq development team. *The Coq Proof Assistant Reference Manual, version 8.0*, 2004.
12. Catherine Dubois and Véronique Viguié Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates, July 1998. `www.cs.bham.ac.uk/~ mmk/cade98-partiality`.
13. Stéphane Glondu. Garantie formelle de correction pour l'extraction Coq, 2007. http://stephane.glondu.net/rapport.2007.pdf.
14. John Harrison. Inductive definitions: Automation and application. In P. J. Windley, T. Schubert, and J. Alves-Foss, editors, *Higher Order Logic Theorem Provoing and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Sciences*. Springer-Verlag, 1995.
15. Gérard Huet. Induction principles formalized in the calculus of constructions. In *TAPSOFT'87*, volume 249 of *LNCS*, pages 276–286. Springer, 1987.
16. Gilles Kahn. Elements of constructive geometry group theory and domain theory, 1995. available as a Coq user contribution at `http://coq.inria.fr-/contribs-eng.html`.
17. Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
18. Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
19. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
20. Tobias Nipkow. Winskel is (almost) right:towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
21. Bengt Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.
22. Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, 1993. LIP research report 92-49.
23. Christine Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq, 2007. `http://www.lri.fr/~paulin/PUBLIS/paulin07kahn.pdf`.
24. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
25. Lawrence C. Paulson. *Logic and computation, Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

26. Franz Regensburger. HOLCF: Higher order logic of computable functions. In P. J. Windley, T. Schubert, and J. Alves-Foss, editors, *Higher Order Logic Theorem Provoing and Its Applications: Proceedings of the 8th International Workshop*, volume 971 of *Lecture Notes in Computer Sciences*. Springer-Verlag, 1995.
27. Glynn Winskel. *The Formal Semantics of Programming Languages, an introduction*. Foundations of Computing. The MIT Press, 1993.