Fixed point semantics and partial recursion in Coq

Yves Bertot, Vladimir Komendantsky

Fixed Point Semantics and Partial Recursion in Coq [*]

Yves Bertot Vladimir Komendantsky
INRIA Sophia Antipolis, France
{bertot,vkomenda}@sophia.inria.fr

Abstract

We propose to use the 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 modelled 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.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Lambda Calculus and Related Systems

General Terms Verification

Keywords least fixed point semantics, non-terminating functions, the Knaster–Tarski theorem, automated theorem proving, program extraction

1. Introduction

For theoretical computer scientists, the Knaster–Tarski least fixed point theorem, as well as its application – the first fixed point theorem of Kleene [30], 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 [17, 16]. We consider the following generalised statement of the Knaster–Tarski theorem [1]:

Theorem 1 (Knaster–Tarski; complete p.o.). *Given a monotonic function f on a chain-complete partial order, consider the following transfinite sequence $\{x_i\}$:*

$$\begin{aligned} x_0 &= \bot \\ x_{\alpha+1} &= f(x_\alpha) \\ x_\beta &= \text{ the least upper bound of the chain } \{f(x_\alpha)\}_{\alpha<\beta} \\ &\qquad \text{if } \beta \text{ is a limit ordinal.} \end{aligned}$$

The function f has a least fixed point obtained by an iterative process on $\{x_i\}$ starting from x_0. 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, adding classical logic axioms to the constructive 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 $A \rightarrow B$ are actually modelled as functions of type $A \rightarrow B_\bot$: the fact that a function may not terminate is recorded in its type, non-terminating computations are given the value $\bot$ 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 [27, 19], 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. There is no need to discuss the first case which 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 behaviour predicted by the axiom. We claim that this can be done safely if the 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 sim-

[*] This work was partially supported by the French ANR project CompCert

ple 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.

The Knaster–Tarski least fixed point theorem guarantees two important properties of the function that is defined by the iterative process. The first property is that the obtained function satisfies the fixed point equation. The importance of fixed point equations is straightforward [3, 4]. 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 [30]. 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 give an overview of our basic formalisation of domain theory. Then, we show how this theory can be used in the definition of simple recursive functions. We give examples with proofs about recursive functions. We discuss extraction and execution of the recursive programs that are obtained in this way. Future work and related work is reviewed at the end of the paper. All the experiments described in this paper were done with Coq [13, 6] and can be found in the Internet from the first author's web page [7].

2. Domain-Theoretic Constructions

Our domain-theoretic development, found at [7], comprises basic ideas of domain theory built on the notion of a preorder, that is a theory of (pointed) complete preorders as formalised by C. Paulin-Mohring [26]. In addition to the constructive complete preorders of Paulin-Mohring, we introduce flat complete preorders that we define making certain non-constructive steps which are to be specified below. Various libraries of domain theory already exist in several proof systems. Therefore we just invite the reader to grasp the basic constructions by reading our development since these constructions are quite standard for such a library.

In the present paper we abbreviate "pointed complete preorder" as "cpo", following the convention in [26], and we systematically omit the word "pointed". In Figure 1, we summarise some useful notation.

We define the following inductive type which is equivalent to the standard option type of Coq:

```
Inductive partial (A : Type) : Type :=
  Def : A → partial A | Undef : partial A.
```

The type partial is introduced in order to provide the hiding mechanism that would prevent the user of our library to encode, for instance, a classical proof for the halting problem but still permit unconstrained use of the standard option type of Coq. This hiding mechanism, which is just a side-effect of the continuity proofs, ensures that no value of the kind Def x is returned in the case Undef of a pattern-matching construct. The abstraction of the type partial also guarantees the safety of optimisations suggested in Section 6. (The consistency of the system and the validity of the extraction process do not depend on this abstraction.)

ord	preorder
cpo	(pointed) complete preorder, cpo
$\xrightarrow{o}$	preordered function space
$\xrightarrow{m}$	preordered monotonic function space
$\xrightarrow{c}$	preordered continuous function space
$\xrightarrow{O}$	function cpo
$\xrightarrow{M}$	monotonic function cpo
$\xrightarrow{C}$	continuous function cpo
nat_ord	natural numbers ordered by $\leq$
chain O	nat_ord$\xrightarrow{m}O$
$\_{}_{o}$	(preordered) function application
@	monotonic function composition
$==_O$	derived equality on the preorder O

Figure 1. Notation

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

Lifting of the flat preorder to a flat cpo requires a non-constructive definition for 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 c on $\&\mathsf{ord}\ A$, 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:

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

we obtain a Σ-type definition for the least upper bound of a chain c on A that contains two parts: an element a of $\&\mathsf{ord}\ A$ 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 $\&\mathsf{ord}\ A$; we denote it by $\&\mathsf{cpo}\ A$.

The constructive definite description axiom is quite strong. However, it does not make the system inconsistent, assuming the predicative Set. By introducing this axiom the only thing one loses is the assurance that it will be possible to extract a sensible program form any proof. In Section 6 we show how the extraction mechanism can be instructed to extract correct programs from definitions made with our library.

The constructive definite description axiom is incompatible with variants of the Calculus of Constructions, where the Set sort is impredicative, due to the Chicli–Pottier–Simpson paradox [11] which was shown to hold for even a weaker version, the classical definite description axiom (in Prop rather than Type), in the presence of classical logic. Therefore our development is restricted to the predicative Set which is, fortunately, default for Coq.

3. Kleene's Fixed Point Theorem

The well-known fixed point theorem of Kleene, see [30], has a mild generalisation in the setting of complete preorders. Below we outline a formalised proof for this statement. Our proof follows the lines of the classical textbook proofs found in, e.g., [22, 30]. 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 [26]. First, we define a function f_iter as follows ($\perp$ denotes the bottom element of the cpo D):

```
Fixpoint f_iter (D:cpo)(f:D $\xrightarrow{m}$ D)(n:nat_ord) : D :=
  match n with
     0 ⇒ ⊥
   | S n' ⇒ f (f_iter f 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:cpo)(f:D $\xrightarrow{m}$ D) : D := lub (iter f).
```

We prove the fixed point property f_fixp f == f (f_fixp f). Next, 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 $\xrightarrow{C}$ D) $\xrightarrow{C}$ D := . . .
Lemma fixp_eq : ∀ D (f:D$\xrightarrow{c}$D), fixp f == f (fixp f).
```

In fact, the type of fixp with implicit argument D, that is (D $\xrightarrow{C}$ D) $\xrightarrow{C}$ D, is not a continuous function cpo as it may seem from the above definition. It is implicitly coerced by Coq to the class of functions, Funclass, whose objects are of the form (∀ x:A, B), for A B : Type. Therefore a proper argument for fixp has type Type. This is exactly the case with f : D $\xrightarrow{c}$ D since the type of f here is implicitly coerced to Type. Moreover, note that a coercion from D $\xrightarrow{c}$ D to Type requires only one step, whereas a coercion from D $\xrightarrow{C}$ D to Type would have required two.

This allows to formalise Kleene's theorem as follows:

Theorem 2 (Kleene). ∀ (D:cpo)(f:D $\xrightarrow{c}$ 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. Fixed Point Definitions of Partial 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 $f : A \rightarrow$ &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 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 value Undef is returned in the case Undef of this matching construct.

For example, consider the minimisation functional μ defined as follows: for all A : Type, $f : A \rightarrow$ nat $\rightarrow$ nat, the value of μf is a function $g : A \rightarrow$ 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 it 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 → partial nat. First, we specify a functional f_mu as follows:

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

Then we prove monotonicity and 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 $\xrightarrow{o}$ nat_ord $\xrightarrow{o}$ &ord nat) $\xrightarrow{m}$ (A $\xrightarrow{o}$ nat_ord $\xrightarrow{o}$ &ord nat)
    := . . .
Definition cont_mu :
 (A $\xrightarrow{O}$ nat_ord $\xrightarrow{O}$ &cpo nat) $\xrightarrow{C}$ (A $\xrightarrow{O}$ nat_ord $\xrightarrow{O}$ &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. Note that the value $|a - b|$ on nat, for a and b in nat, can be defined using the standard truncated subtraction on nat as $(a - b) + (b - a)$. Consider the function $\lambda xy.|x - y^2|$ with the following definition:

```
Definition abs_x_minus_y_squared (x y : nat) :=
 Def ((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 (with the trailing 0 being the value for y in f_mu):

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

Next, we demonstrate some proof ideas concerning this partial recursive function.

5. Certification of Functions

In [30], Winskel describes *fixed point induction* as a technique for proving properties of least fixed points of continuous functions. This style of induction is restricted to certain predicates which are called *refining* in his work. The same notion also appears in HOLCF, see [29, 21], 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 – we do not provide automated admissibility proofs – still we provide basic techniques for proofs about recursive functions with flat target types.

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 an omnipresent lemma that we employ to return a value of the least upper bound of a chain on a flat cpo:

```
Lemma lub_flat_cpo_witness :
 ∀ (c : chain &cpo), ∃ n, c n = lub c.
```

The above lemma can be used to prove that, for any input, the value of fixp f can also be computed by iter f n for some natural number n:

```
Lemma fixp_flat_witness :
 ∀ (A B:Type) (f : (A$\xrightarrow{O}$&cpo B)$\xrightarrow{C}$(A$\xrightarrow{O}$&cpo B)) (x:A),
   ∃ n, fixp f x = iter f n x.
Proof.
 intros A B f x.
```

Next, we extract a witness n using lub_flat_cpo_witness from the chain iter f _$_o$ x on &cpo B; the witness equality (iter f _$_o$ x) n = lub (iter f _$_o$ x) is given the name Hn:

```
  destruct (lub_flat_cpo_witness (iter f _o x)) as [n Hn].
```

We provide this n as the witness:

```
  exists n.
```

and rewrite the goal with the witness equality:

```
  rewrite Hn.
```

We arrive at the goal fixp f x = lub (iter f _$_o$ x) which is proved by unfolding the structure of f:

```
  case f; intro f'; case f'; reflexivity.
Qed.
```

Thus the number n is an upper bound on the number of recursive calls that are needed to compute the value of f x. Thanks to this theorem, one can reason by induction on n and simulate the fixed point induction of [30]. Notably, there is an additional match goal with step required before the rewrite step in some specific versions of Coq where the unification algorithm is not powerful enough to solve the given higher-order unification problem.

For instance, using fixp_flat_witness we can conclude (by simulating the fixed point induction) with the following lemma which asserts that the function perfect_sqrt never terminates on inputs not being perfect squares.

```
Lemma perfect_sqrt_Undef :
  ∀ x, (∀ y:nat, x ≠ y*y) → perfect_sqrt x = Undef.
Proof.
  intros x Hx.
  unfold perfect_sqrt, mu.
  destruct (fixp_flat_witness_2
    (@cont_mu nat abs_x_minus_y_squared) x 0) as [n Hn].
  rewrite Hn.
  apply perfect_sqrt_Undef_iter with (1:=Hx).
Qed.
```

For the proof, we extract a witness n and give the name Hn to the corresponding witness equality:

```
Hn: fixp (cont_mu abs_x_minus_y_squared) x 0 =
      iter (cont_mu abs_x_minus_y_squared) n x 0
```

then, after the rewrite step, we arrive at the goal

```
iter (cont_mu abs_x_minus_y_squared) n x 0 = Undef
```

that is proved by a technical lemma we will explain shortly. In the proof of this lemma we refer to the two-argument version of fixp_flat_witness (which can be proved either directly or by uncurrying followed by an application of the one-argument fixp_flat_witness):

Lemma fixp_flat_witness_2 :
  ∀ A B C (f : (A $\xrightarrow{O}$ B $\xrightarrow{O}$ &cpo C) $\xrightarrow{C}$ (A $\xrightarrow{O}$ B $\xrightarrow{O}$ &cpo C)) x y,
    ∃ n, fixp f x y = iter f n x y.

and then we refer to the following statement proved by induction on n:

```
Lemma perfect_sqrt_Undef_iter :
  ∀ n x y, (∀ z:nat, x ≠ z*z) →
    iter (mono_mu abs_x_minus_y_squared) n x y = Undef.
Proof.
  induction n.
```

Now we have to prove the inductive basis:

```
∀ x y:nat, (∀ z:nat, x ≠ z * z) →
  iter (mono_mu abs_x_minus_y_squared) 0 x y = Undef
```

and the proof is easy:

```
  reflexivity.
```

Next, we prove the inductive step:

```
∀ x y:nat, (∀ z:nat, x ≠ z * z) →
  iter (mono_mu abs_x_minus_y_squared) (S n) x y = Undef
```

which we do by case analysis:

```
  simpl; unfold f_mu; simpl.
  intros x y Hxneq.
  case_eq (x - y * y + (y * y - x)).
```

The first case, x-y*y + (y*y-x) = 0; we conclude by contradiction:

```
  intro Heq0.
  contradiction (Hxneq y); omega.
```

The second case, x-y*y + (y*y-x) = S n0; we conclude by the inductive hypothesis, that is IHn:

```
  intros _ _.
  apply (IHn x (S y) Hxneq).
Qed.
```

The third tool provided in our library is the following lemma that relates computations done with iter and values of a recursive function:

Lemma iter_Def_eq_fixp :
  ∀ A B (f : (A$\xrightarrow{O}$&cpo B)$\xrightarrow{C}$(A$\xrightarrow{O}$&cpo B)) x n v,
    iter f n x = Def v → fixp f x = Def v.

Using this lemma, one can compute values of recursive functions, provided that none of these values is Undef. One simply needs to guess the right argument n that leads to a definite value of the form Def v which in this case is known to be the value of the recursive function for the corresponding argument. The problem is to decide whether a given number n is the right one. If the chosen value is too small, the value returned by the iterative process is the uninformative Undef.

For the sake of the example below we can prove the two-argument version iter_Def_eq_fixp by uncurrying and then applying iter_Def_eq_fixp (the alternative direct proof is easy):

Lemma iter_Def_eq_fixp_2 :
  ∀ A B C (f:(A$\xrightarrow{O}$B$\xrightarrow{O}$&cpo C)$\xrightarrow{C}$(A$\xrightarrow{O}$B$\xrightarrow{O}$&cpo C)) x y n v,
    iter f n x y = Def v → fixp f x y = Def v.

Now we can demonstrate a simulation of an iterative computation:

```
Lemma compute_perfect_sqrt_36 : perfect_sqrt 36 = Def 6.
Proof.
  unfold perfect_sqrt, mu.
  apply iter_Def_eq_fixp_2 with (n:=100); reflexivity.
Qed.
```

The number 100 used in this example is only required to be an upper bound on the recursive calls enough to compute perfect_sqrt 36 (in this case 7 would be enough). This approach may be used in reflexive tactics, for example, in a proof that involves computation of the value of a recursive function. One can try a fixed number of calls, and if a value of the form Def is returned, the proof can proceed, otherwise the tactic fails, which nevertheless does not signify the divergence of the recursive function.

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 behaviour, that is it loops exactly on arguments which are not perfect squares. Below we show how this can be made possible.

Some of our lemmas rely on the Classical and ClassicalDescription extensions of the Coq libraries. These extensions add only two axioms to the logic of Coq, namely, the axiom of excluded middle, and the constructive definite description axiom: ∀ A (P : A→Prop), (∃! x : A, P x) → {x : A | P x}. The antecedent states that there exists a unique x satisfying the property P but it does not provide a method to construct x. The consequent asserts that one can use this unique x as a constructive value. Using this axiom, one eliminates the distinction between constructive values and logically unique values. This of course disrespects the distinction between Set and Prop that plays the central role during the extraction process of Coq.

The extraction mechanism relies on the assumption that, for every element of the form {x : A | P x}, there exists a constructive procedure for obtaining the x's part. However, the axiom produces elements of that form without providing any constructive procedure. For example, the flat_cpo type extracts to OCaml as

```
let flat_cpo = {
  o_cpo = flat_ord;
  bot = (Obj.magic (Obj.magic Undef));
  lub = (fun x → projT1 (ExistT
    ((match excluded_middle_informative with
        | Left → constructive_definite_description __
        | Right → Obj.magic Undef), __))) }
```

Obj.magic casts a value of any type into type Obj.t, which is convertible with any type, and is used, in this example, by the Coq extraction mechanism to represent implicit coercions. The fixp functional in OCaml, after systematic inlining of functions and erasing occurrences of Obj.magic, becomes

```
let fixp d = fun n → d.lub (f_iter d ((fun c → c) n))
```

Thus one can see that, in OCaml, the lub of a flat_cpo has essentially no computational content, and therefore fixp on a flat cpo does not compute the expected value. For this reason, the extraction mechanism expects the user to handwrite the procedure that returns the expected value wherever the axiom is used.

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 behaviour corresponds to the behaviour described in the Coq development.

We construct a function fix that computes the fixed point of functionals. This function should satisfy the following equality:

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

The introduced argument d corresponds to the cpo argument in Coq that is in most cases implicit there but explicit in OCaml. Changing the orientation of the equation seems enough to do the job:

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

```
(** val mu : ('a1 → nat → nat partial) → Obj.t **)

let mu f =
  Obj.magic (Obj.magic (fixp (funcpo (funcpo flat_cpo))))
    (Obj.magic
      (Obj.magic
        (Obj.magic (fun x x0 x1 →
          match f x0 x1 with
            | Def n →
              (match n with
                | O → Def x1
                | S n0 → x x0 (S x1))
            | Undef → Undef))))
```

Figure 2. Unoptimised mu

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. Execution can be delayed as follows:

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

The obtained function fix is the function we propose to attach to the function fixp for extraction purposes. The corresponding instruction to the Coq extraction mechanism is the following:

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

The extracted OCaml code is for the function mu is shown in Figure 2. Occurrences of Obj.magic correspond to implicit coercions that appear in Coq.

For the rest of this section we assume that the argument d of fix is implicit.

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 steps in execution of fix f' a'. When fix f' a' is being executed, 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; hence, 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 behaviour 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. □

In our model of recursive functions, the value Undef is only used for functions that are undefined because they fail to terminate (and hence Undef expresses *divergence*). We say that a Coq expression contains a *spontaneous divergence* if it contains an instance of Undef which is not encapsulated inside a pattern-matching construct on the type partial, 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 Undef ⇒ e_1 | Def ⇒ e_2 end

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 e contains no spontaneous divergence then its value can never be the value* Undef*.*

Proof. One proves the statement by induction on the length of execution of e. The only sub-expressions that can produce the value Undef are the Undef expressions that appear in the Undef branch of a pattern-matching construct. In the pattern-matching construct, the matched expression also has the property of containing no spontaneous divergence and one can therefore use the induction hypothesis. □

We verified a formal Coq model of this proof in the context of the Mini-ML language [12] extended with a data-type representing the type partial and the corresponding pattern-matching construct [7].

If values Undef can never be produced then the encapsulations of expressions inside the constructor Def and the operations of removing the constructor Def done by pattern-matching turn out to be redundant. These, along with the occurrences of type partial, can be recursively eliminated from the program since every instance of Def e can be replaced with e and every instance of the construct

$$\mathsf{match}\ e\ \mathsf{with}\ \mathsf{Undef} \Rightarrow e_1\ \mid\ \mathsf{Def}\ x \Rightarrow e_2$$

can be replaced with let $x = e$ in e_2. In what follows, we denote by $e\!\downarrow$ the result of this recursive elimination in e.

Theorem 5. *If e is an expression containing no spontaneous divergence and the value of execution of e is v then the value of execution of $e\!\downarrow$ is $v\!\downarrow$.*

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

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

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

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

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

Proof. Here we focus on the two most important cases:

1. If e has the form Def 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 $e_1\!\downarrow$ is $v_1\!\downarrow$. But in this case, we have $e' = e_1\!\downarrow$ and the value of e is Def $v_1\!\downarrow$. However, we obviously have (Def $v_1\!\downarrow)\!\downarrow = v_1\!\downarrow$, which justifies the result.
2. If e has the form

   $$\mathsf{match}\ e_1\ \mathsf{with\ Undef} \rightarrow \ldots \mid \mathsf{Def\ x} \rightarrow e_2$$

   then $e\!\downarrow$ is the expression

   $$\mathsf{let}\ x = e_1\!\downarrow\ \mathsf{in}\ e_2\!\downarrow$$

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

□

```
(** val mu : ('a1 → nat → nat) → Obj.t **)

let mu f =
  Obj.magic (Obj.magic (fixp (funcpo (funcpo flat_cpo))))
    (Obj.magic
      (Obj.magic
        (Obj.magic (fun x x0 x1 →
          match f x0 x1 with
            | O → x1
            | S n0 → x x0 (S x1)))))
```

Figure 3. mu, optimised according to Theorem 5

A model of this proof was formally verified using the Mini-ML description of the language. This proof is available in [7].

The extracted code of the mu function can be optimised according to Theorem 5 as shown in Figure 3. Thus we have eliminated the type partial from the code. Occurrences of Obj.magic still remain as traces of coercions that appear in the Coq code for mu. All these occurrences can be safely erased from the code.

7. Automation of the Least Fixed Point Definition

We develop a command we called Fcpo Function that allows to define least fixed points of partial recursive functions and automate certain routine steps in this process. The definition of mu using the new command (in its current version) is the following:

```
Variable A : Type.
Variable f : A → nat → partial nat.

Fcpo Function mu : A → nat → partial nat :=
  fun x y ⇒
    match f x y with
      Undef ⇒ Undef
    | Def 0 ⇒ Def y
    | _ ⇒ mu x (S y)
    end.
```

Then the command generates the recursive functional f_mu, as in Section 4, and two proof obligations that the user is required to satisfy in order to complete the definition:

1. The first obligation is the monotonicity proof for f_mu, followed by the automated definition of the monotonic version of f_mu, that is mono_mu (see Section 4).
2. The second obligation is the continuity proof for mono_mu, followed by the definition of the continuous version of mono_mu, that is cont_mu.

Finally, the required least fixed point of cont_mu is defined automatically as fixp cont_mu.

The syntax of Fcpo Function is the following:

Fcpo Function $ident$ $[binder_1$ $[\dots$ $binder_n]]$:
$term_1 := term_2$

where $term_1$ is a type with target partial A, for A a type.

This command is compatible with the standard Coq extraction procedure augmented by the two extraction constants, as we described in Section 6.

8. Future Work

There are a few features related to Fcpo Function that are now in an early development stage:

1. Comprehensive automation of monotonicity and continuity proofs.
2. Automated elimination of the cpo structure (and hence occurrences of Obj.magic) from the extracted code.
3. Techniques that allow to hide partial from the user of the command.

In fact, hiding techniques would require full proof automation as well since otherwise one would still need to work at the level of partial types during the work on proof obligations generated by the command.

Our effort to formalise optimisations of the extracted code should also be compared with efforts done to give a formal description of the extraction procedure, as studied by Letouzey [19] and Glondu [15].

9. 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 effort was put into providing relevant collections of inductive types with terminating computation derived from the notion of primitive recursion [10, 25, 2]. In particular, it was shown that the notion of accessibility or noetherian induction could be described using an inductive predicate in [24]. 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 [14, 8], 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 for inputs for which 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 some applications, like interpreters for Turing-complete languages (where termination is undecidable) or functions for which potential non-termination is an accepted defect.

There are analogous techniques that also do not require a termination proof. One such method is Prop-bounded recursion based on a coinductive monad [20, 9]. In this method the coinductive type permits to represent infinitely long computations as (potentially) infinitely large objects of coinductive types. One is also able to write a function that directly refers to itself, which would be incompatible with, e.g., Fixpoint due to syntactic restrictions. However, the coinductive methods require a more complex notion of equality due to the presence of infinite objects, such as, in the case of [20], the John Major equality which relies on the JMeq axiom of Coq.

Another work [4, 6] 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. Following this work as an inspiration there was provided a more general command Function [13] that supports several ways of defining recursive functions and related objects such as induction principles.

In one of our experiments [5], we defined the semantics of a small programming language in the spirit of [22, 30]. We used the Knaster–Tarski fixed point theorem to describe the semantics of while loops as suggested in [30]. Then we were able to prove that when a value is returned, the same computation can be modelled by a natural semantics derivation, using an encoding of the natural semantics based on an inductive predicate. This reproduces a similar formalised proof in [23]. Once extracted to ML, this gives a certified interpreter for the language.

In an early version of the Calculus of Constructions, formalisations of the 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 [17]. In this respect, it is also worthwhile to mention that [16] shows how this theorem can be used to give a definitional justification of inductive types in higher-order logic.

In this paper, we formalised 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 [28]. It was also formalised in Isabelle's HOL instantiation to provide a package known as HOLCF [29, 21]. We believe these other experiments can give us guidelines for improvements.

10. 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. The major aim of this paper is to provide yet another way to model potentially non-terminating functions. Our inspiration comes from the Knaster–Tarski least fixed point theorem.

Our contributions can be summarised as follows.

First, we formalised a domain theory based on the notion of a preorder and equipped it with flat cpos. This work is analogous to Isabelle/HOLCF and allows one to provide potentially non-terminating functions with a least fixed point semantics in Coq.

Second, we provided arguments in favour of our claim that the fixed point combinator is the right computational value for the Knaster–Tarski theorem and should therefore be used for extraction of functional programs from Coq.

Third, we used two extraction axioms (least fixed point and constructive definite description) in course of extraction and obtained a powerful way to represent in Coq potentially non-terminating functions and reason about them.

Acknowledgments

The authors are grateful to the anonymous referees for their detailed comments. This work has also benefited from suggestions by Benjamin Werner, Hugo Herbelin and Peter Aczel, and also from early experiments by Kuntal Das Barman. The first author also wishes to remember the late Gilles Kahn, who started work on formalised domain theory in the context of the Calculus of Inductive Constructions in mid-90's [18].

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] P. 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] R. C. Backhouse. Fixed point calculus. In R. C. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *LNCS*. Springer-Verlag, 2002.

[4] A. Balaa and Y. Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, Jan. 2002.

[5] Y. Bertot. Theorem proving support in programming language semantics, 2007. `http://hal.inria.fr/inria-00160309`.

[6] Y. Bertot and P. 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.

[7] Y. Bertot and V. Komendantsky. Proofs on domain theory and partial recursion, 2008. `http://www-sop.inria.fr/marelle/Yves.Bertot/tarski.html`.

[8] 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.

[9] A. Bove and V. Capretta. Computation by prophecy. In S. R. D. Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 70–83. Springer, 2007.

[10] J. Camilleri and T. Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical report, University of Cambridge, 1992.

[11] L. Chicli, L. Pottier, and C. 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.

[12] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, Aug. 1986.

[13] Coq development team. *The Coq Proof Assistant Reference Manual, version 8.1*, 2006.

[14] C. Dubois and V. V. Donzeau-Gouge. A step towards the mechanization of partial functions: domains as inductive predicates, July 1998. `www.cs.bham.ac.uk/~mmk/cade98-partiality`.

[15] S. Glondu. Garantie formelle de correction pour l'extraction Coq, 2007. `http://stephane.glondu.net/rapport.2007.pdf`.

[16] J. 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.

[17] G. Huet. Induction principles formalized in the calculus of constructions. In *TAPSOFT'87*, volume 249 of *LNCS*, pages 276–286. Springer, 1987.

[18] G. 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`.

[19] P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[20] A. Megacz. A coinductive monad for prop-bounded recursion. In A. Stump and H. Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007*, pages 11–20, New York, NY, USA, 2007. ACM.

[21] O. Müller, T. Nipkow, D. v. Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

[22] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.

[23] T. Nipkow. Winskel is (almost) right:towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.

[24] B. Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.

[25] C. 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.

[26] C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq, 2007. `http://www.lri.fr/~paulin/PUBLIS/paulin07kahn.pdf`.

[27] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[28] L. C. Paulson. *Logic and computation, Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[29] F. 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.

[30] G. Winskel. *The Formal Semantics of Programming Languages, an introduction*. Foundations of Computing. The MIT Press, 1993.