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 Tarski's 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, Tarski's least fixed point theorem is a simple basic block to assert the existence of objects defined by recursive equations. These objects may be inductive types and recursive functions [12,11]. However, to use this theorem, one needs to be able to express that the domain of interest indeed has the required completeness property and that the function being considered indeed 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 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 [21,14], performs a cleaning operation so

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

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 Tarski's least fixed point theorem is given a sensible computational content.

Tarski'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, Tarski's 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 [2]. This fixed point equation is useful when we want to prove that under some conditions a function is guaranteed to terminate. The second property is that the obtained function is the least fixed point. Therefore it is also the least upper bound of a sequence obtained by iterating the functional. 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 [24]. 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 [9,4] and can be found on the Internet from the first author's web page[1].

2 Basic notions

In this section, we follow the lines of our Coq development on domain theory. We pursue the approach proposed in [20] 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 dependent record containing a carrier element A of type $Type$ 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 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 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 $O_1 \xrightarrow{m} O_2$.

Now we can use convenient notation to define a *chain* on a preorder O as $\mathsf{nat\_ord} \xrightarrow{m} O$. 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 $f : O_1 \xrightarrow{m} (O_2 \xrightarrow{m} O_3)$, *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 $f : O_1 \xrightarrow{m} O_2$ and $g : O_2 \xrightarrow{m} O_3$ 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

[1] The current address is `http://www-sop.inria.fr/marelle/Yves.Bertot/tarski.html`.

the three laws below:

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

In the present paper we abbreviate "complete preorder" as "cpo" as in [20].

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

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

For cpos D_1 and D_2, a function $f : D_1 \xrightarrow{m} D_2$ 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 $f : D_1 \xrightarrow{m} D_2$ 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 $D_1 \xrightarrow{c} D_2$.

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

For a preorder O and a cpo D, the cpo $O \xrightarrow{M} D$ of monotonic functions from O to D is constructed by reusing the structure of the cpo $O \xrightarrow{O} D$. 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 $D_1 \xrightarrow{C} D_2$.

2.2 The flat preorder

We define the flat order on a type A by specifying a binary relation $\leq_{\mathsf{option}\ A}$ (using the standard option type of Coq) such that, for $x\ y : A$, $x \leq_{\mathsf{option}\ A} y$ iff $x = y$ or $x = \mathsf{None}$. We denote this flat order by $\&\mathsf{ord}\ A$.

Lifting of the flat order 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 $\&\mathsf{ord}\ A$, there exists an x such that $\forall n, c\ n \leq_{\&\mathsf{ord}\ A} x$ and $\forall y, (\forall n, c\ n \leq_{\&\mathsf{ord}\ A} y) \rightarrow x \leq_{\&\mathsf{ord}\ A} y$, 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 $\mathsf{lub}_{\&\mathsf{ord}\ A} : \mathsf{chain}\ (\&\mathsf{ord}\ A) \rightarrow \&\mathsf{ord}\ A$ function 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 [7]. Fortunately, this is not the default for Coq and our work is done with predicative Set.

3 Proving the fixed point theorem

The statement of the theorem we are interested in is the following:

Theorem 1. *In a complete preorder, every continuous function has a least fixed point for the derived equality.*

Our proof is inspired by the ones found in courses on programming language semantics like [16,24].

For the proof, we consider a cpo D and a function $f : D \xrightarrow{m} D$. First, we define a function f_iter as follows:

```
Fixpoint f_iter (n:nat_ord) : D := match n with 0 ⇒ bot D | 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 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 equivalence f_fixp == f f_fixp, we then define a continuous version of f_fixp, and we also have the fixed point property for this functional.

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

The fixed point returned by fixp is the least by construction, as stated in Kleene's theorem. For us, it is important that this value is the least upper bound of the iter chain. We will use this in proofs about recursive functions.

All arguments in the proof of fixp_eq are constructive, but non-constructive arguments may be hidden in the proof that chains have least upper bounds, and this part may differ from one domain to another.

4 Using the fixed point theorem to define recursive functions

In general, we want to model a recursive function f of type $A \to$ &cpo B and we have to construct a continuous functional F of type

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

such that $f = Ff$.

The next problem is to show that the functionals we encounter are continuous. In practice, our recursive functions will respect a smooth regularity: the value $\bot$ is added to the result type to represent the fact that the function may not terminate. The condition of continuity on the functional F corresponds to this interpretation of "potential non-termination": 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, if we want to define a factorial function we can specify a functional f_fact as follows:

```
Definition f_fact : (f:Z → &ord Z)(z:Z) : &ord Z :=
  match Zeq x 0 with
    true ⇒ Some 1
  | false ⇒ match f_fact (z - 1) with None ⇒ None | Some v ⇒ Some (z*v) end
  end.
```

Next, we prove monotonicity and then continuity of f_fact and specify the functions monofact and contfact with the proof of monotonicity and, respectively, continuity attached as follows:

```
Definition monofact : (Z -m-> &ord Z) -m-> (Z -m-> &ord Z) := ...
Definition contfact : (Z -O-> &cpo Z) -C-> (Z -O-> &cpo Z) := ...
```

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

```
Definition fact := fixp contfact.
```

Proving that functionals are continuous can be cumbersome, but they can usually be understood as the composite 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 Some true ⇒ x | Some false ⇒ y | None ⇒ None end.
```

and we can prove once and for all that this function preserves monotonicity and continuity of the functions it combines, with a definition and a lemma of the following form:

```
Definition m_cond
  (O:ord)(t:O -m-> &cpo bool)(F G:(O -m-> &ord O) -m-> (O -m-> &ord O)) :
    (O -m-> &ord O) -m-> (O -m-> &ord O) := ...
```

```
Lemma m_cond_continuous :
 ∀ (D:cpo)(A:Type)(t:D -c-> &cpo bool)(F G:(D -M-> &cpo A) -c-> (D -M-> &cpo A)),
   continuous (m_cond t F G).
```

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
   Some y ⇒ match f with Some g ⇒ g y | None ⇒ None end
 | None ⇒ None
 end.
```

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

```
Definition Apply (A B:Type) : (&cpo (A→&cpo B)) -C-> (&cpo A) -C-> (&cpo B) := ...
```

With these basic units, the code for our factorial example is convertible to the following one:

```
Definition f_fact2 : (Z → &ord Z) → (Z → &ord Z) :=
 fun f z ⇒
   cond
     (Some (Zeq_bool z 0))
     (Some 1)
     (Apply (Some (fun v ⇒ Some (z*v))) (f (z-1))).
```

and the proof that the function is continuous boils down to a traversal of the basic elements that are combined in the function, it only requires several lines in our experiments.

Notice that the definition of the Apply function 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 [24], 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 [23,15], 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 order as the codomain. In this case, we have a theorem which states 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 (f : (A -O-> &cpo B) -C-> (A -O-> &cpo B)) x, ∃ n, fixp f x = 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 function's value on x. Thanks to this theorem, we can reason by induction on n and simulate the fixed point induction of [24].

For instance, to prove that our function fact never terminates on negative inputs, we first prove the following statement by induction on n

```
∀ n x, x < 0 → iter monofact n x = None
```

This proof is done by induction on n. Note that the quantification over x is part of the statement proved by induction. This is important because x changes as the computation progresses, while remaining negative. This proof has only a dozen steps.

Using the fixp_flat_witness we can conclude with the following lemma:

```
Lemma fact_neg_none : ∀ x, x < 0 → fact x = None.
```

In one of our experiments [3], we defined the semantics of a small programming language in the spirit of [16,24]. We used Tarski's fixed point theorem to describe the semantics of while loops as suggested in [24]. 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 [17].

Another lemma relates computations done with iter and values of the recursive function. This lemma has the following statement:

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

Using this lemma, we can compute values of recursive functions, provided that none of these vales is the minimal value None. We simply need to guess the right argument n that leads to a definite value of the form Some 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 iter f n x is the uninformative None.

Here is an example that uses this theorem:

```
Lemma compute_fact_6 : fact 6 = 720.
Proof.
unfold fact; apply iter_some_eq_fixp with (n:=100%nat); 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 fact 6 (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 form Some 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

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 the 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 | 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 wherever the axiom is used. The process cannot be mechanized and one cannot help extraction at this level.

We have a solution for this problem. The "constructive definite description" axiom is only used in the definition of the function fixp, to transform the existential statement of Tarski's fixpoint theorem into a value that can be used in other functions. We can provide a handwritten constructive content for the fixp function, 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.

Basically, we want to have a function fix that computes the fixpoint of functionals, so that this function should satisfy this equality:

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

Just reversing the equation and making it into a defining equation seems enough to make 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 somehow by blocking execution through encapsulation in a λ-expression. Using Ocaml syntax this is done in the following manner.

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

This function fix is the function we propose to attach to the fixp function for extraction purposes. Obviously this imposes a restriction: while in Coq, we can use the fixp function 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 the following:

```
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 2. *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. In this proof, we assume the extraction process to behave 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 executing f' (fun y → fix f' y) a, and the next steps concern executing 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, executing (fun y → fix f' y) e behaves like executing fix f' e. Together with the assumption that the extracted code in f' and a' outside of fix expressions behaves as expected, this ensures the property.

A few remarks are to be made. Our result concerns only terminating computations. However, we believe there should be cases when extracted programs loop even though the Coq models predict terminating computation, simply because the call-by-value evaluation strategy can occasionally make bad choices when computing values of function arguments.

In [3], we described the denotational semantics of a simple imperative programming language and proved it sound with respect to a natural semantics specification, in the spirit of [17,24]. Once extracted to ML, this gives a certified interpreter for the language.

There are two improvements that we can propose but for which we have not been able to produce an experiment yet. The first improvement is that the fixed point function should actually be inlined directly in every recursive function that is based on the least fixed point theorem. In the current situation, the extracted code for our fact function is a direct call to the fix function with the extracted code for contfact as first argument. Because of this, every recursive call of the function we want to model is implemented with two function calls in the target language. Moreover, this precludes optimizations like tail-recursion optimization.

The second improvement concerns useless pattern-matching constructs that are added in the code to mirror the pattern-matching constructs that appear in the Calculus of Constructions models to handle possible non-termination. These pattern-matching constructs appear as pattern-matching on option types and are useless because no function ever explicitly produces a None value, at least if the models are written with the discipline that None should be used only to represent failure to terminate and not any other cause of failure. To avoid these useless pattern-matching constructs, we suggest that a specific type constructor, other than option, should be used to construct flat domains and the use of the bottom constructor from this specific type should be forbidden except to allow for non-termination in continuous combinators like cond or Apply. This discipline should be verified syntactically by the extraction tool itself. In this manner, the pattern-matching constructs related to non-termination can still appear in the models, but they should be absent in the extracted code.

The reason for this improvement is that the bottom value can only be produced after an infinite time by functions that do not terminate. It is safe and straightforward not to account for this possibility, because such a value can never be computed by a program.

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 equipped with terminating computation derived from the notion of primitive recursion [6,19,1]. In particular, it was shown that the notion of accessibility or noetherian induction could be described using an inductive predicate in [18]. 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 [10,5], 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, but this 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.

Other work [2,4] 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.

We have 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 have been performed in the LCF system [22]. It was also formalized in Isabelle's HOL instantiation to provide a package known as HOLCF [23,15]. We believe these other experiments can give us guidelines for improvements.

In early version of the Calculus of Constructions, formalizations of Tarski's least fixpoint theorem were also used to show how inductive definitions could be encoded directly in the pure (impredicative) Calculus of Constructions [12]. In this respect, it is also worthwhile to mention that [11] shows how this theorem can be used to give a definitional justification of inductive types in higher-order logic.

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 Tarski's least fixed point theorem, a well-known 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 Tarski's least fixed point theorem is faithfully realized by the functional value fix. We provided an informal proof for the correctness of fix, but it would be more satisfying to consider a formal proof of this statement, for instance by relying on the formal description of a minimal functional language such as [8].

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 [13].

References

1. 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.
2. Antonia Balaa and Yves Bertot. Fonctions récursives générales par itération en théorie des types. In *Journées Francophones pour les Langages Applicatifs*, January 2002.
3. Yves Bertot. Theorem proving support in programming language semantics, 2007. `http://hal.inria.fr/inria-00160309`.
4. 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.

5. 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.
6. Juanito Camilleri and Tom Melham. Reasoning with inductively defined relations in the HOL theorem prover. Technical report, University of Cambridge, 1992.
7. 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.
8. 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.
9. Coq development team. *The Coq Proof Assistant Reference Manual, version 8.0*, 2004.
10. 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`.
11. 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.
12. Gérard Huet. Induction principles formalized in the calculus of constructions. In *TAPSOFT'87*, volume 249 of *LNCS*, pages 276–286. Springer, 1987.
13. 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`.
14. 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.
15. Olaf Müller, Tobias Nipkow, David von Oheimb, and Oskar Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
16. Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, 1992.
17. Tobias Nipkow. Winskel is (almost) right:towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10:171–186, 1998.
18. Bengt Nordström. Terminating general recursion. *BIT*, 28:605–619, 1988.
19. 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.
20. Christine Paulin-Mohring. A constructive denotational semantics for kahn networks in coq, 2007. `http://www.lri.fr/~paulin/PUBLIS/paulin07kahn.pdf`.
21. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
22. Lawrence C. Paulson. *Logic and computation, Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.
23. 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.

24. Glynn Winskel. *The Formal Semantics of Programming Languages, an introduction*. Foundations of Computing. The MIT Press, 1993.