While Loops in Coq

David Nowak, Vlad Rusu

While Loops in Coq

David Nowak, CNRS* Vlad Rusu, Inria†

While loops are present in virtually all imperative programming languages. They are important both for practical reasons (performing a number of iterations not known in advance) and theoretical reasons (achieving Turing completeness). In this paper we propose an approach for incorporating while loops in an imperative language shallowly embedded in the Coq proof assistant. The main difficulty is that proving the termination of while loops is nontrivial, or impossible in the case of non-termination, whereas Coq only accepts programs endowed with termination proofs. Our solution is based on a new, general method for defining possibly non-terminating recursive functions in Coq. We illustrate the approach by proving termination and partial correctness of a program on linked lists.

1 Introduction

The definition of recursive functions in the Coq proof assistant [4] is subject to certain restrictions to ensure their termination, which is essential for the consistency of Coq's underlying logic. Specifically, recursive calls must be made on strict subterms, effectively ensuring that the computation eventually reaches a base case. Alternatively, users have the option to prove that a specific quantity strictly decreases according to a well-founded order. In such cases, Coq can automatically transform the recursive calls into strict subterm calls, using a so-called accessibility proof to guarantee termination. Adhering to these constraints eliminates the risk of infinitely many calls, thereby ensuring that functions terminate.

An alternative, somewhat ad-hoc strategy is to introduce an additional natural-number argument called the *fuel*. The fuel's value is decremented with each recursive call, thereby guaranteeing finitely many recursive calls, hence, termination. However, a crucial concern arises as one must supply enough fuel so that termination does not disrupt the intended computation of the program by occurring too early.

In this paper we present a novel approach to defining possibly partial recursive functions in Coq while achieving separation of concerns: write the program first, and prove its properties (including termination) later. In broad terms our technique consists in providing an infinite amount of fuel for recursive functions. By doing so the function can proceed with its computations without risk of exhausting its fuel.

As a result, this approach empowers developers to focus on the core logic of the recursive function, separate from the termination concern, streamlining the development process and enhancing the modularity and readability of the code.

A key property guaranteed by our technique is that, given the *functional* of the recursive function under definition (i.e., an abstract description of the function's body), the resulting function is the *least fixpoint* of its functional. We prove this general result under mild constraints on the functional - it must be monotonic and, in some sense described precisely in the paper, must preserve continuity.

The method is applied to while loops in an imperative shallowly embedded in Coq. By proving that the functional of while loops is monotonic and continuity-preserving we obtain while loops as least fixpoints of their functionals. This enables programmers to construct imperative programs featuring

*Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
†Univ. Lille, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

H. Cheval, L. Leuştean, A. Sipoş (Eds.):
7th Symposium on Working Formal Methods (FROM 2023)
EPTCS 389, 2023, pp. 96–109, doi:10.4204/EPTCS.389.8

arbitrary while loops and provides them with tools for reasoning about loops. Specifically, the least-fixpoint property is used for proving that a while loop terminates if and only if there exists some finite amount of fuel for which it returns the desired result; hence, a termination proof can proceed by induction on the fuel value, once an adequate instantiation for this known-to-exists value is chosen.

Subsequently, we proceed to establish a Hoare logic system, which serves as a formal tool for proving the partial correctness of programs. In essence the Hoare logic provides a systematic and rigorous approach to program verification, where one defines preconditions and postconditions that govern the state of the program before and after its execution. Through these assertions one can verify that the program's execution leads to the desired outcomes, establishing its partial correctness. Here, again, the property of being a fixpoint is used for proving the soundness of the while loop's *Hoare triple*.

Finally, the least-fixpoint property of the partial functions being defined ensures that the functions, defined abstractly using order theory, are, as mathematical functions, the same as the ones that Coq would have generated, had it not been constrained by its logic into rejecting the functions as partial.

Outline In Section 2 we introduce the reader, termination, and state monads, which serve the purpose of writing imperative programs in the Coq proof assistant. Moving forward to Section 3 we present our method for defining possibly nonterminating recursive functions and demonstrate its application to the definition of while loops. In Section 4 we define a monadic Hoare logic and illustrate its effectiveness by applying it to a program that computes the length of a linked list. In Section 5 we address the issue of proving termination. We compare with related work in Section 6 and conclude in Section 7.

The Coq development corresponding to this paper is available at `https://tinyurl.com/2p93uwdj`.

2 Monads for Possibly Nonterminating Stateful Computation

We consider a subset of Gallina (the programming language of Coq) expressive enough for shallowly embedding possibly nonterminating imperative programs. In purely functional languages such as Gallina the usual approach is to encode imperative features with monads [11]. We use a combination of the termination, state, and reader monads. The first one is used for possibly nonterminating computations, the second one for stateful computations where the state may change, and the third one for state-aware computations that do not change the state, such as checking the condition of while loops.

A monad consists of a type constructor equipped with an operation usually called `ret` for trivial computation, and another one, usually called `bind`, for sequencing computations. Each particular monad also comes with specific operations. Assume a context where a type `T` is declared. The type constructor for the reader monad is defined as follows:

```
Definition reader (A: Type): Type := T -> A.
```

where `A` is a type for values returned by a computation. Intuitively, `reader` is a side-effect free function from states to computed values. Unlike the state monad, shown below, it only reads from states, without modifying them; hence the monad's name `reader`.

Trivial computation `ret` consists in ignoring the state:

```
Definition ret {A: Type}(a: A) : reader A := fun _ => a.
```

Placing the parameter `A` between curly brackets marks it as *implicit* - its value can therafter be automatically inferred, relieving users from the burden of having to instantiate it.

Sequencing of computations consists in passing the result of the first computation to the second one:

```
Definition rbind {A B: Type}(m: reader A)(f: A -> reader B): reader B :=
fun s => f (m s) s.
```

Finally, the reader monad is equipped with a primitive for reading the state:

```
Definition get: reader T := fun s => s.
```

The other monad used in the shallow embedding of an imperative language in Coq is a combination of termination and state monads, for which we use the `option` type constructor:

```
Inductive option (X: Type): Type := None: option X | Some: X -> option X.
```

where `None` encodes nontermination. Termination state monads are the programs of our imperative language, which is reflected in the name of its type constructor:

```
Definition program (A: Type): Type := T -> option (A * T).
```

The reader monad is a special case of the termination state monad. Thus we introduce a *coercion* that enables Coq to automatically convert a reader monad into a termination state monad when needed:

```
Coercion reader_to_program {A: Type}(m: reader A): program A :=
fun s => Some (m s, s).
```

The `ret` and the `get` primitive of the reader monad are thus automatically converted. But the sequencing of computations needs to be redefined in order the take into account the fact that the first of the sequenced computations might change the state, or might not terminate:

```
Definition bind {A B: Type}(m: program A)(f: A -> program B): program B :=
fun s => match m s with None => None | Some (a, s') => f a s' end.
```

The termination state monad comes with an additional primitive to change the state:

```
Definition put (s: T) : program unit := fun _ => Some (tt, s).
```

The `unit` type (inhabited by one term - the constant `tt`) is used in our functional-laaguage setting for modelling imperative programs that do not return anything, encoded by returning the dummy constant `tt`.

As a running example we consider an imperative program that computes the length of a linked list. First we need to specify the `State` on which the program works. We assume a machine with two positive integer registers and an unbounded memory whose adresses and stored values are also positive integers:

```
Record State: Type := {reg1: nat; reg2: nat; memory: nat -> nat}.
```

Then, using the primitives of our monad we write operations to read/write in registers and memory:

- `read_reg1`, `write_reg1` for reading, resp. writing in the first register, and similar functions for reading/writing in the second register. The second register is increased by one using `incr_reg2`;
- `do next <- read_addr curr` assigns to `next` the content of the address `curr` in memory. More generally, reading operations return a value that can be bound to an identifier using the standard "do" notation of monads, i.e., `do x <- m; f x` is a shortcut for `rbind m (fun x => f x)`;
- operations are sequenced by double semicolons: `m;;f` is a notation for `bind m (fun _ => f)`.

This almost allows us to write a program computing the length of a linked list. Not completely, because the program uses a `while` loop, which is not defined until later in the paper:

```
Definition length (addr: nat): program State nat :=
write_reg1 addr;;
write_reg2 0;;
while (do curr <- read_reg1; ret  (curr != 0))
  (
   incr_reg2;;
   do curr <- read_reg1;
   do next <- read_addr curr;
   write_reg1 next
  );;
do res <- read_reg2; ret res.
```

It is assumed that the linked list[1] of interest starts in memory at address `addr`. This address is written into the first register, then the second register (which is to contain the length of the list) is initialized to zero. Next, while the current address of the first register is not `null` (also encoded by zero), the second register is incremented and the first register is updated to point to the next element of the linked list. Finally, at the end of the `while` loop (if the end is ever reached), the value in the second register is the length of the list of interest, hence, it is the value returned by our function whenever it terminates.

What is still missing is, of course, the definition of the `while` loop. A first attempt uses recursion:

```
Fixpoint
 while{T: Type}(cond: reader T bool)(body: program T unit): program T unit:=
  do c <- cond;if c then body;;while cond body else reader_to_program (ret tt)
```

That is, a recursive function (introduced by the keyword `Fixpoint`) attempts to define a `while` loop (with condition `cond` and body `body` of appropriate types) by first checking the condition, and if the condition holds, executing the body then recursively the `while` loop; otherwise, doing nothing (which is encoded by `reader_to_program (ret tt)`). However, this function does not always terminate. For example, if the `while` loop is used to navigate a linked list, like in the case of the `length` function above, and the list is badly linked, i.e., it contains a loop, then the `while` loop does not terminate. Coq rejects this definition attempt, as it rejects any recursive function whose termination it cannot infer.

In the rest of the paper we show how possibly infinite while loops (and, in general, partial recursive function) can be accepted by Coq by encoding nontermination as evaluation to a special value.

3 Partial Recursive Functions in Coq

The `while` loop is a particular case of a partial recursive function. We first sketch how partial recursive functions can be encoded in Coq before providing details.

[1] For simplicity we consider linked lists where each element only contains the next element's address.

3.1 Outline of the Approach

Assume we want to define a partial recursive function `f` from type `A` to type `B`. A natural way to proceed is to give `f` the type `A -> option B`, where for any `a:A`, `f a = None` encodes the fact that `f` is undefined for the input `a`. In order to define a function we further need its *functional*, an abstract representation of the body of the function being defined. Let `F : (A -> option B)-> A -> option B` be the functional for `f`. We say that `f := F f` is the *fixpoint definition* of `f`. The interesting case we here solve is when fixpoint definitions are not accepted by Coq - just like in the case of the `while` function above.

We proceed as follows. We define an auxiliary function `f_fuel : nat -> A -> option B` with an additional natural-number parameter called the *fuel*, as the following recursive function, which is accepted by Coq because Coq "sees" that the `fuel` parameter strictly decreases at each recursive call:

```
Fixpoint f_fuel (fuel: nat) (a: A) : option B:=
match fuel with
|S fuel' => F(f_fuel fuel') a
    (*S is the successor function on natural numbers*)
|0 => None
end
```

If the functional `F` is *monotonic* then, based on results in order theory explained later in this section, the function `f_fuel` can be *lifted* to a *continuous* function in `conat -> option B` where `conat` is the type

```
Inductive conat: Type:= finite: nat -> conat | infinity
```

That is, the inhabitants of `conat` are natural numbers wrapped with the `finite` constructor, together with the constant `infinity`. Putting back the parameter `a:A` in the type we obtain a function `f_inf` of the type `conat -> A -> option B`. The results later in the section also ensure that, under an additional condition on `F` (preservation of continuity), the function `(f_inf infinity)` is the *least fixpoint* of `F`.

Recapitulating, we started with the intention of defining a function `f : A -> option B`, using its functional `F : (A -> option B)-> A -> option B`, via the fixpoint definition `f := F f`. We have assumed this is rejected by Coq. Per the results below we define `f:= f_inf infinity` and prove that is the least solution of the fixpoint equation `f = F f` - precisely the solution that Coq would have constructed had it accepted the definition `f := F f` - with the advantage that our definition *is* accepted.

3.2 Elements of Order Theory

The results in this subsection have been adapted from the textbook [2]. We have formalized them in Coq, hence, hereafter proofs are only sketched or omitted altogether. The examples are not only used for illustration purposes: they also serve as building blocks in our approach to partial recursive functions.

Definition 1 *A* pointed partial order *(PPO)* $(S, \preceq, \bot)$ *is a partially ordered set* $(S, \preceq)$ *together with a distinguished element* $\bot \in S$ *such that for all* $s \in S$, $\bot \preceq s$.

Example 1 The triple $(\mathbb{N}, \leq, 0)$ consisting of natural numbers $\mathbb{N}$, their usual order $\leq$, and the least natural number 0 form a PPO.

Example 2 For any set A, the triple $(A \cup \{\bot\}, \preceq, \bot)$ with $\bot \notin A$, and $\preceq$ being defined as the smallest relation on $A \cup \{\bot\}$ such that $\bot \preceq a$ and $a \preceq a$ for all $a \in A$, is a PPO called the *flat PPO of A*.

In Coq the flat PPO of a type `A` is encoded using the type `option A` where `None` plays the role of $\bot$.

Definition 2 *Given a PPO $(S, \preceq, \bot)$, a set $S' \subseteq S$ is* directed *if $S \neq \emptyset$ and for all $x, y \in S$ there exists $z \in S$ such that $x, y \preceq z$.*

Example 3 Any nonempty set of natural numbers in the PPO $(\mathbb{N}, \leq, 0)$ is directed.

The above example is a consequence of the more general fact that any nonempty sequence, i.e., totally ordered subset of elements in a PPO, is directed. Indeed, directed sets are generalizations of sequences.

Example 4 In a flat PPO $(A \cup \{\bot\}, \preceq, \bot)$, the directed sets are exactly: the singletons $\{x\}$ with $x \in A \cup \{\bot\}$, and the pairs of elements of the form $\{a, \bot\}$ with $a \in A$.

Definition 3 *A* Complete Partial Order *(CPO) is a PPO $(S, \preceq, \bot)$ with the additional property that any directed set $T \subseteq S$ has a* least upper bound*, denoted by $lub\, T$.*

Least upper bounds of directed sets are generalizations of limits of sequences.

Example 5 Consider the PPO $(\mathbb{N} \cup \{\infty\}, \leq, 0)$ with the order $\leq$ on natural numbers extended such that $\infty \leq \infty$ and $n \leq \infty$ for all $n \in \mathbb{N}$. Then, $(\mathbb{N} \cup \{\infty\}, \leq, 0)$ is a CPO. Indeed, in this totally ordered set all subsets $T \subseteq \mathbb{N} \cup \{\infty\}$ are directed, and $lub\, T$ is either:

- the maximum of T, if it exists
- ∞, if the maximum of T does not exist.

In Coq the set $\mathbb{N} \cup \{\infty\}$ shall be encoded as the type `conat` seen earlier in this section.

Example 6 In a PPO $(A \cup \{\bot\}, \preceq, \bot)$, the least upper bound of a singleton $\{x\}$ is x and the least upper bound of a pair $\{\bot, a\}$ is a. Those are the only directed sets in this PPO; hence, $(A \cup \{\bot\}, \preceq, \bot)$ is a CPO.

Informally, the notion of compactness below captures what it means for an element to be finite.

Definition 4 *In a CPO $(S, \preceq, \bot)$, an element $s^\circ \in S$ is* compact *whenever for all directed sets $T \subseteq S$, if $s^\circ \preceq lub\, T$ then there exists $t \in T$ such that $s^\circ \preceq t$.*

Example 7 In the CPO $(\mathbb{N} \cup \{\infty\}, \leq, 0)$ the compact elements are exactly the (finite) natural numbers.

Example 8 In the CPO $(A \cup \{\bot\}, \preceq, \bot)$ all the elements are compact.

Some CPOs are, in the following sense, completely determined by their compact elements:

Definition 5 *A CPO $(S, \preceq, \bot)$ having the set $S^\circ \subseteq S$ of compacts is* algebraic *if for all $s \in S$, the set $\{s^\circ \in S^\circ \mid s^\circ \preceq s\}$ is directed, and $s = lub\, \{s^\circ \in S^\circ \mid s^\circ \preceq s\}$.*

Example 9 The CPO $(\mathbb{N} \cup \{\infty\}, \leq, 0)$ is algebraic. Indeed, for all $n \in \mathbb{N} \cup \{\infty\}$, the set of compacts (natural numbers) $\{m \in \mathbb{N} \mid m \leq n\}$ is directed (as is any nonempty subset of $\mathbb{N} \cup \{\infty\}$). Moreover,

- if $n = \infty$, then the set $\{m \in \mathbb{N} \mid m \leq n\}$ coincides with $\mathbb{N}$, and $lub\, \mathbb{N} = \infty$;
- if $n \in \mathbb{N}$, then $lub\, \{m \in \mathbb{N} \mid m \leq n\} = n$.

Example 10 The flat CPO $(A \cup \{\bot\}, \preceq, \bot)$ is algebraic. Indeed:

- for all $a \in A$, the set $\{\bot, a\}$ of compacts in the $\preceq$ relation with a is directed, and $a = lub\, \{\bot, a\}$
- the set $\{\bot\}$ of compacts in the $\preceq$ relation with $\bot$ is directed and $lub\, \{\bot\} = \bot$.

We shall use the following notion, which relates a PPO to the compact elements of an algebraic CPO:

Definition 6 *Consider a PPO* $(S^\circ, \preceq^\circ, \bot^\circ)$ *and an algebraic CPO* $(T, \preceq, \bot)$ *whose set of compacts is* T°. *We say that* $(T, \preceq, \bot)$ *is an* embedding *of* $(S^\circ, \preceq^\circ, \bot^\circ)$ *if there exists an injection* $\iota : S^\circ \to T$ *such that:*

- $\iota \bot^\circ = \bot$*;*
- ι *is monotonic;*
- ι *maps* S° *to* T°*, written* $\iota S^\circ = T^\circ$.

The embedding, including the injection involved in it, is denoted by $\iota : (S^\circ, \preceq^\circ, \bot^\circ) \to (T, \preceq, \bot)$.

Example 11 The embedding $\kappa : (\mathbb{N}, \leq, 0) \to (\mathbb{N} \cup \{\infty\}, \leq, 0)$ is induced by the canonical inclusion κ of $\mathbb{N}$ into $\mathbb{N} \cup \{\infty\}$. The embedding of $(A \cup \{\bot\}, \preceq, \bot)$ into itself is also induced by the canonical inclusion.

Remark: not all embeddings are induced by canonical inclusions. In Coq, in general, they are not. This is because Coq is based on type theory, hence, one cannot just add a new element to a type; to do this one must create a new type and wrap the old type in a constructor (which, in Coq, is always an injection). An example of this is the representation of $\mathbb{N} \cup \{\infty\}$ as the type `conat` with a constructor `finite : nat -> conat`. With the appropriate order and bottom element, `conat` is an embedding of `nat` induced by the constructor `finite`. The only situations when canonical inclusion induces an embedding in Coq is when it coincides with the identity function, like in the embedding of $(A \cup \{\bot\}, \preceq, \bot)$ into itself.

Definition 7 *Assume an embedding* $\iota : (S^\circ, \preceq^\circ, \bot^\circ) \to (T, \preceq, \bot)$. *We denote by* $\iota^{-1} : T \to S^\circ$ *the (unique) function such that* $\iota^{-1}(\iota\, s^\circ) = s^\circ$ *for all* $s^\circ \in S^\circ$*, and* $\iota^{-1}\, t = \bot^\circ$ *for* $t \in T \setminus (\iota S^\circ)$.

Hence ι^{-1} is the inverse of ι on the compacts ιS° of T, and elsewhere it is given the (arbitrary) value $\bot$.

The next theorem is our main ingredient for defining and reasoning about partial recursive functions. It uses the following notion of continuity:

Definition 8 *Given two CPOs* $(T, \preceq, \bot)$ *and* $(T', \preceq', \bot')$*, a function* $f : T \to T'$ *is* continuous *if for any directed set* $S \subseteq T$*, its image* $(f\ S) \subseteq T'$ *is directed, and* $f\ (lub\ S) = lub(f\ S)$.

Theorem 1 *Assume two embeddings* $\iota_1 : (S_1^\circ, \preceq_1^\circ, \bot_1^\circ) \to (T_1, \preceq_1, \bot_1)$ *and* $\iota_2 : (S_2^\circ, \preceq_2^\circ, \bot_2^\circ) \to (T_2, \preceq_2, \bot_2)$ *and a monotonic function* $f^\circ : S_1^\circ \to S_2^\circ$*. Then there exists a unique continuous function* $f : T_1 \to T_2$ *such that* $f = \iota_2 \circ f^\circ \circ \iota_1^{-1}$ *— where* $\circ$ *is the standard notation for function composition.*

If the embeddings in Theorem 1 are canonical inclusions we have a simpler version of the above result:

Corollary 1 *Assume two embeddings* $\iota_1 : (S_1^\circ, \preceq_1^\circ, \bot_1^\circ) \to (T_1, \preceq_1, \bot_1)$ *and* $\iota_2 : (S_2^\circ, \preceq_2^\circ, \bot_2^\circ) \to (T_2, \preceq_2, \bot_2)$ *where* ι_1, ι_2 *are canonical inclusions. Then, for any any monotonic function* $f^\circ : S_1^\circ \to S_2^\circ$ *there exists a unique continuous function* $f : T_1 \to T_2$ *such that for all* $s^\circ \in S_1^\circ$*,* $f\ s^\circ = f^\circ s^\circ$.

3.3 Application to Partial Recursive Functions

We use the existence part of Theorem 1 in order to define partial recursive functions and the uniqueness part in order to prove that the defined functions are least fixpoints of their respective fixpoint equations.

The method has been formalized in Coq; we sketch it below in mathematical notation. Assume that we want to define a partial function $f : A \to (B \cup \{\bot\})$. We have have at our disposal the functional $F : (A \to (B \cup \{\bot\})) \to A \to (B \cup \{\bot\})$. The following assumptions on F are required:

- monotonicity: for all $f, f' : A \to (B \cup \{\bot\})$, if *for all* $a \in A$, $f\ a \preceq f'\ a$ then *for all* $a \in A$, $F\ f\ a \preceq F\ f'\ a$, where $\preceq$ is the flat order on $(B \cup \{\bot\})$.

- preservation of continuity: assume an arbitary function $g : (\mathbb{N} \cup \{\infty\}) \to A \to (B \cup \{\bot\})$. If, for each $a \in A$, the function $\lambda n \to g\ n\ a$ is continuous (as a function between $(\mathbb{N} \cup \{\infty\})$ and $(B \cup \{\bot\})$ organized as CPOs) then, for each $a' \in A$ the function $\lambda n \to F\ (g\ n)\ a'$ is continuous as well.

The method proceeds as a series of steps, grounded in the results from the previous subsection:

1. A function $f^\circ : \mathbb{N} \to A \to (B \cup \bot)$ is recursively defined by the equations: for all $a \in A$ and $m \in \mathbb{N}$, $f^\circ\ 0\ a = \bot$ and $f^\circ\ (m+1)\ a = F\ (f^\circ\ m)\ a$; intuitively, for all $m \in \mathbb{N}$, $(f^\circ\ m)$ constitute approximations of the function that we want to define, constrained by the finite amount of fuel $m \in \mathbb{N}$;
2. the *monotonicity* requirement on F ensures that, for all $a \in A$, the function $f^\circ_a = \lambda n \to (f^\circ\ n\ a)$ is monotonic as a function between $\mathbb{N}$ and $(B \cup \{\bot\})$ organized as PPOs;
3. the *existence* result of Theorem 1 ensures that, for all $a \in A$ that there exists a continuous function $f_a : (\mathbb{N} \cup \{\infty\}) \to (B \cup \{\bot\})$, satisfying $f_a\ m = f^\circ\ m\ a$ for all $m \in \mathbb{N}$; here, we have used the fact that, in the embeddings involved in our application of Theorem 1, the injections are the canonical inclusions (cf. Example 11), hence, one can apply the simpler version of the theorem — Corollary 1;
4. using the *uniqueness* result of the corollary, for all $a \in A$, any continuous function $f'_a : (\mathbb{N} \cup \{\infty\}) \to (B \cup \{\bot\})$ satisfying $f'_a\ m = f^\circ\ (m+1)\ a$ for all $m \in \mathbb{N}$, also satisfies $f'_a = f_a \circ (\lambda n \to n+1)$; here we have used the fact that the function $\lambda\ n \to n+1 : (\mathbb{N} \cup \{\infty\}) \to (\mathbb{N} \cup \{\infty\})$ is continuous, and that the composition of continuous functions is a continuous function as well;
5. for all $a \in A$, let $f'_a : (\mathbb{N} \cup \{\infty\}) \to (B \cup \{\bot\})$ be defined by $f'_a = \lambda n \to F\ (\lambda (x : A) \to f_x\ n)\ a$. Using the *continuity preservation* requirement on F, the continuity of f'_a reduces to the continuity of $\lambda n \to (\lambda (x : A) \to f_x\ n)\ a = \lambda n \to f_a\ n = f_a$; since f_a is continuous, f'_a is continuous as well;
6. moreover, using the definitions of f° (first item in this list), of f_a (item 2), and of f'_a (item 5): for all $m \in \mathbb{N}$, $f'_a\ m = F\ (\lambda x \to f_x\ m)\ a = F(\lambda x \to f^\circ\ m\ x)\ a = F\ (f^\circ\ m)\ a = f^\circ\ (m+1)\ a$; hence, (cf. item 4), $f'_a = f_a \circ (\lambda n \to n+1)$. This implies that for all $n \in (\mathbb{N} \cup \{\infty\})$, $f_a\ (n+1) = F\ (\lambda x \to f_x\ n)$;
7. let now $f : A \to (B \cup \{\bot\})$ defined, for all $a \in A$, by $f\ a = f_a\ \infty$. Then, using item 6 and $\infty = \infty + 1$: for all $a \in A$, $f\ a = f_a\ \infty = f_a\ (\infty + 1) = F\ (\lambda x \to f_x\ \infty)\ a = F\ (\lambda x \to f\ x)\ a = F\ f\ a$; that is, we have obtained the fixpoint equation $f = F\ f$. What remains to be proved is that f is its least solution;
8. for this, we inductively define a sequence of functions in $A \to (B \cup \{\bot\})$ by $F^0 = \lambda x \to \bot$ and, for all $m \in \mathbb{N}$, $F^{m+1} = F(F^m)$. Using the definition of f we prove the equality $f = lub\{F^n \mid n \in \mathbb{N}\}$, where the least upper bound is taken in the CPO of functions $A \to (B \cup \{\bot\})$ ordered pointwise. Finally, we use a result that says that if F is monotonic on a CPO (it is, in our case, by the *monotonicity* assumption) and $lub\{F^n \mid n \in \mathbb{N}\}$ is a fixpoint of F (it is, in our case, since $lub\{F^n \mid n \in \mathbb{N}\} = f$ and $f = F\ f$) then $lub\{F^n \mid n \in \mathbb{N}\}$ is the least fixpoint of F; i.e., f is the least fixpoint of F.

A comparison between the proposed approach for defining partial recursive functions and the standard one based on Kleene's fixpoint theorem is discussed in Section 6 dedicated to related works.

3.4 Instantiation to While Loops

The results from the previous subsection are now instantiated to `while` loops.

Recall from subsection 3.1 the failed attempt at defining `while` loops in Coq, and notice their type:

```
Fixpoint while{T:Type}(cond:reader T bool)(body:program T unit):program T unit:=
  do c <- cond; if c then body ;; while cond body else reader_to_program (ret tt)
```

In order to instantiate the method described in the previous subsection to `while` loops we first need to change their type to `A->option B` for appropriate `A`, `B`. Remembering that `program T unit` is defined as `T->option(unit*T)`, once the implicit parameter `T` is chosen, the type of `while` becomes

```
(reader T bool) -> (program T unit) -> T -> option(unit*T)
```

In order to obtain a type of the form `A->option B` we *uncurry* the above type to

```
((reader T bool)*(program T unit)*T) -> option(unit*T)
```

where * builds products between types. Next, we define a function `while':A->option B` where `A=(reader T bool)*(program T unit)*T` and `B=unit*T`. For this we first write the *functional* for the `while'` function as follows

```
Definition While'{T:Type}(W:((reader T bool)*(program T unit)*T)->option unit*T)
                         (p:(reader T bool)*(program T unit)*T): option unit*T :=
let (cond,body,s)  := decompose p in
(
 do c <- cond;
 if c then
 body;;(fun (s':T)=>W (cond, body, s')) (*after ;; a function on T is expected*)
 else reader_to_program (ret tt)
) s
```

(Notice how the parameter `p` was decomposed into three components.) After proving that `While'` is monotonic and preserves continuity, we obtain using the method in Subsection 3.3 the function `while': (reader T bool)*(program T unit)*T -> option unit*T` as the least fixpoint of `While'`.

What remains to be done is to *curry* the type `(reader T bool)*(program T unit)*T->option unit*T` to the expected type of the `while` function. When this is done, we obtain `while{T:Type}: (reader T bool)->(program T unit)->T->option unit*T` as the least fixpoint of the functional

```
Definition While{T:Type}(W:(reader T bool)->(program T unit)->T->option unit*T)
     (cond :(reader T bool))(body : (program T unit))(s :T)) : option unit*T :=
(do c <- cond; if c then body;;(W cond body) else reader_to_program (ret tt)) s
```

which concludes our construction of `while` loops in Coq. The next step is to provide users with means to reason about programs that contain such loops. This is the object of the next two sections. They shall be using the two following facts, which are consequences of `while` being the least fixpoint of its functional:

- an unfolding lemma, which is just another form of the fixpoint equation:

  ```
  Lemma while_unfold{T:Type}: forall(c: reader T bool)(b: program T unit),
  while c b =
        (do c' <- c; if c' then b;;while c b else reader_to_program (ret tt))
  ```

- a lemma stating that the `while` loop evaluates to `Some x` in a state if and only if there exists a fuel-constrained version of the loop that also evaluates to the same `Some x` in the same state:

  ```
  Lemma while_iff_while_fuel{T:Type}:
  forall (c:reader T bool)(b:program T unit)(s:T)(x:unit*T),
  while c b s = Some x <-> exists (fuel:nat), while_fuel fuel c b s = Some x.
  ```

  Since evaluation to `Some x` models termination, the lemma can also be read as"if a loop terminates, then it terminates in finitely many steps" - where the number of steps is upper-bounded by `fuel`.

4 Partial Correctness

In this section we define a monadic Hoare logic for partial correctness [7, 14]. Roughly speaking, partial correctness expresses the fact that a program returns the right answer whenever it terminates. In this paper, a program is a monadic computation. Remembering that `Prop` is the Coq type for logical statements, one writes the Hoare triple `{{P}} m {{Q}}` for the proposition `hoare_triple P m Q` defined in Coq by

```
Definition hoare_triple
  {T A : Type} (P : T -> Prop) (m : program T A) (Q : A -> T -> Prop) : Prop :=
forall s s' a, P s -> m s = Some (a, s') -> Q a s'.
```

That is, if the program `m: (program T A)` is in a state `s: T` such that the pre-condition `P: T -> Prop` holds for `s`, if the program terminates (encoded by the fact that the program returns `Some (a, s')`) then the pair `(a, s')` satisfies the postcondition `Q: A -> T -> Prop`.

There are Hoare triples for all monadic instructions, but the triple of interest is this paper is the one for the `while` loops. It states that: if the `body` of the loop preserves an invariant `I` as long as the condition `cond` of the loop is true, then the loop preserves the invariant whenever it terminates.

```
Lemma while_triple
  {T: Type}(cond: reader T bool)(body: program T unit)(I: T -> Prop):
{{ fun s => I s /\ cond s = true }} body {{ fun _ s' => I s' }} ->
{{ I }} while cond  body {{ fun _ s' => cond s' = false /\ I s' }}.
```

In order to prove `while_triple` we first prove a triple for fuel-constrained loops by induction on `fuel`:

```
Lemma while_fuel_triple
 {T:Type}(fuel:nat)(cond: reader T bool)(body: program T unit)(I: T -> Prop):
{{ fun s => I s /\ cond s = true }}body{{ fun _ s' => I s' }} ->
{{ I }} while_fuel fuel cond body  {{ fun _ s' => cond s' = false /\ I s' }}.
```

The lemma `while_fuel_triple` is used in the proof of `while_triple`, together with the lemmas `while_iff_while_fuel` and `while_unfold` shown at the end of the previous subsection.

Then, `while_triple` is used to prove the *weakest precondition* triple for our running example:

```
Lemma length_wp (addr: nat)(P: nat -> State -> Prop):
{{ fun s => forall len,
     Length s addr len ->
     P len {| reg1 := 0; reg2 := len; memory := s.(memory) |} }}
length addr
{{ P }}.
```

where `Length` is an inductively defined relation that says that in a state `s`, the list starting at address `addr` has length `len`. Having this predicate gives us an abstract manner of defining a linked list's length:

```
Inductive Length (s:State): nat -> nat -> Prop :=
| length_nil : forall addr, addr = 0 -> Length s addr 0
| length_cons :
  forall addr len,
  addr <> 0 -> Length s (s.(memory) addr) len -> Length s addr (S len).
```

In accordance to the laws of weakest precondition, `length_wp` says that the postcondition `P` must hold in the precondition when applied to the expected return value upon termination (the length `len`) and to the expected state upon termination `{|reg1 := 0; reg2 := len; memory := s.(memory)|}`.

As usual with Hoare logic, the crux of the proof is to find the right loop invariant to be fed to the lemma `while_triple`. Here, it is a generalization of the precondition in `length_wp`:

```
fun _ s => forall len,
  Length s s.(reg1) len ->
  P (len+s.(reg2)) {| reg1:= 0; reg2:= len + s.(reg2) ; memory:= s.(memory) |}).
```

With this choice of invariant the proof of `length_wp` is just a matter of unfolding definitions.

The interest of having proved a weakest precondition lies in its generality. As immediate corollaries of `length_wp` we obtain a first lemma that states that whenever `length addr` terminates, the register `reg2` contains the length of the linked list starting at address `addr`:

```
Lemma length_correct1 (s0: State)(addr: nat) :
{{ fun s => s = s0 }} length addr {{ fun _ s' => Length s0 addr s'.(reg2) }}.
```

And another lemma stating that: if the linked list starting at address `addr` has length `len`, then this is the value that will be returned by `length addr` whenever it terminates.

```
Lemma length_correct2 (len: nat)(addr: nat) :
{{ fun s => Length s addr len }} length addr {{ fun n _ => n = len }}.
```

5 Termination

In our approach, termination is modeled by evaluation to `Some` value. In order to succesfully prove termination we need to specify quite precisely the value to which a program evaluates. The key for the termination of the `length` program is the termination of its `while` loop, which is expressed as follows:

```
Lemma while_terminates:
forall len s addr, Length s addr len ->
forall n,
while do curr <- read_reg1; ret  (curr != 0)
(incr_reg2;; do curr <- read_reg1; do next <- read_addr curr; write_reg1 next)
{| reg1 := addr; reg2 := n; memory := s.(memory) |} =
Some (tt, {| reg1 := 0; reg2 := n + len; memory := s.(memory) |}).
```

That is, when called in a state where the first register `reg1` points to the beginning of the list and the second register `reg2` is initialized with some value `n`, the `while` loop ends in a state where the first register is `null` and the second register contains `n` plus the length of the list. The `memory` field of the state remains unchanged because the loop does not write in it. The return value `tt` is the unique inhabitant of `unit` and encodes the fact that `while` loops do not actually return anything relevant.

Lemma `while_terminates` is proved by induction on `n` and uses the lemma `while_unfold` to unfold the loop in the inductive step. It is then used to prove the termination of the `length` program:

```
Lemma length_terminates:
forall  s len addr,
Length s addr len ->
length addr s =
 Some (len,{|reg1 := 0; reg2 := len; memory := s.(memory)|}).
```

This says that the function call `length addr s` terminates with value `len` whenever, according to the inductive relation `Length`, the state `s` has a `memory` field where there is a well-formed linked list of length `len` starting at address `addr`. If the list were not well-formed, i.e. its links would form a loop, then the function would evaluate to `None`, which, in our method denotes non-termination.

6 Related Work

In domain theory [15] partial recursive functions are typically defined as least fixpoints of their functionals using *Kleene's fixpoint theorem*, which states that a functional has a least fixpoint whenever it is *continuous*, and that the least fixpoint is obtained by infinitely many iterations of the functional starting from a function defined nowhere. This theorem is very elegant and easy to prove. Our own version of a fixpoint theorem (perhaps less elegant than Kleene's theorem, and definitely harder to prove) has the same conclusion but requires that the functional be monotonic and *continuity-preserving*, which roughly means that, given a certain continuous function, the functional produces a continuous function. From our own experience and that of other authors (e.g., [3]) it appears that *using* Kleene's theorem, which requires proving continuity, is difficult in practice, even in the simplest cases. For example, it took us hundreds of lines of Coq code just to prove the continuity of the successor function on natural numbers extended with infinity. By contrast, using our version of the theorem requires a proof of preservation of continuity, which appear to be more manageable - for while-loops in a shallow embedding of an imperative programming language in Coq the proof of continuity-preservation is remarkably simple: ten lines of Coq code. One possible reason for why continuity-preservation is easier to prove for functionals than continuity is that that the latter refers to the *higher-order* functional itself, whereas the former concerns the argument of the functional, which is a simpler function, one order below the order of the functional.

Other authors have explored partial recursive functions in Coq. In [5] a partial recursive function's codomain is a *thunk* - a parameterized coinductive type that "promises" an answer as a value of its parameter, but may postpone this answer forever, yielding nontermination. However, the functions being defined now become *corecursive* functions, which are restricted in the Coq proof assistant. As a result, only *tail-recursive* functions can be defined with this approach. This was also noted in [6, Chapter 7.3].

The same author [6, Chapter 7.2] proposes an alternative for the codomain of a partial function: a *computation* is a type that associates to a natural-number approximation level an approximation of the intended function. Like us the author uses the Coq `option` type, where `None` stands for nontermination and `Some` for termination with a return value; and a monad of computations is designed so that computations can encode imperative features. However, [6, Chapter 7.2] requires that the functional of the function being defined be continuous, for a definition of continuity equivalent to that employed in domain theory; hence this approach is subject to the general difficulty of proving continuity of functionals.

Regarding the embedding of imperative languages in proof assistants for the purpose of program verification, an alternative to the shallow embedding that we here use is *deep* embedding, which consist in defining the syntax, operational semantics [9, 10], and program logic for the guest language in the logic of the host. Among the many projects we here cite the Iris project [1], a rich environment built around a concurrent programming language and the corresponding program logic - concurrent separation logic.

7 Conclusion and Future Work

Recursive functions in Coq need to terminate for the underlying logic to be sound. Coq typically ensures termination via an automatically-checkable structural decreasing of terms to which recursive calls apply. In more complex cases Coq can be helped by the user with termination proofs. The termination proof becomes a part of a function definition; a function is not defined until the termination proof is completed.

For various reasons falling under the general notion of separation of concerns it is desirable to separate function definitions from termination proofs. It is also useful to have functions that do not terminate on some inputs. This paper proposes a new approach that achieves these desirable features. Non-termination is simulated as evaluation to a special value interpreted as "undefined". Under mild conditions on the function's body encoded as a higher-order functional, a possibly non-terminating function is defined and proved to be the least fixpoint of its functional according to a certain definition order.

We instantiate the general approach to while-loops in an imperative language shallowly embedded in Coq. The shallow embedding is based on a combination of monads. The least-fixpoint property of the resulting while loops is a key property enabling termination and partial-correctness proofs on imperative programs containing them. The practicality of the approach is illustrated by proving partial correctness and termination properties on a program computing the length of linked lists. Partial correctness is expressed in Hoare logic and is proved in the standard manner, by having users provide a strong-enough invariant; and termination is proved by having users provide an upper bound for the number of iterations.

Future Work A promising line of future work is to extend our approach to defining partial *co*recursive function in Coq. The idea is that codomains of such functions would be encodings of coinductive types organized as algebraic CPOs, generalizing the option types that we here used for recursive functions. Initial experiments with defining some difficult corecursive functions that go beyond Coq's builtin corecursion mechanisms (a filter function on streams, a mirror function on Rose trees) are promising.

The function-definition mechanism presented in this paper critically depends on functionals preserving continuity. A deeper understanding of the relationship between continuity-preservation and continuity, and of the perimeter where the continuity-preservation property holds, is also left for future work.

A more practical future work direction is to apply the instance of our approach to imperative-program definition and verification. Our intention is to build on our experience with proving low-level programs manipulating linked lists [8]. An interesting logic to consider in this setting is separation logic [12], perhaps by drawing inspiration from the shallow embedding of separation logic in Coq presented in [13].

References

[1] *The Iris Project*. `https://iris-project.org/`.

[2] Roberto M. Amadio & Pierre-Louis Curien (1998): *Domains and lambda-calculi*. *Cambridge tracts in theoretical computer science* 46, Cambridge University Press, doi:10.1017/CBO9780511983504. Also available as an Inria research report: `https://hal.inria.fr/inria-00070008`.

[3] Y. Bertot & V. Komendantsky (2008): *Fixed point semantics and partial recursion in Coq*. In: *Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 15-17, 2008, Valencia, Spain*, pp. 89–96, doi:10.1145/1389449.1389461.

[4] Yves Bertot & Pierre Castéran (2004): *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series, Springer, doi:10.1007/978-3-662-07964-5.

[5] Venanzio Capretta (2005): *General recursion via coinductive types*. *Log. Methods Comput. Sci.* 1(2), pp. 1–18, doi:10.2168/LMCS-1(2:1)2005.

[6] Adam Chlipala (2013): *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press. Available at `http://mitpress.mit.edu/books/certified-programming-dependent-types`.

[7] C. A. R. Hoare (1969): *An Axiomatic Basis for Computer Programming*. *Commun. ACM* 12(10), pp. 576–580, doi:10.1145/363235.363259.

[8] Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud & Samuel Hym (2018): *Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base*. In: *18th International Workshop on Automated Verification of Critical Systems (AVOCS 2018)*, Oxford, United Kingdom, doi:10.14279/tuj.eceasst.76.1080. Available at `https://hal.science/hal-01816830`.

[9] Robbert Krebbers, Xavier Leroy & Freek Wiedijk (2014): *Formal C Semantics: CompCert and the C Standard*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, *Lecture Notes in Computer Science* 8558, Springer, pp. 543–548, doi:10.1007/978-3-319-08970-6_36.

[10] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Commun. ACM* 52(7), pp. 107–115, doi:10.1145/1538788.1538814.

[11] Eugenio Moggi (1989): *Computational Lambda-Calculus and Monads*. In: *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, IEEE Computer Society, pp. 14–23, doi:10.1109/LICS.1989.39155.

[12] John C. Reynolds (2005): *An Overview of Separation Logic*. In Bertrand Meyer & Jim Woodcock, editors: *Verified Software: Theories, Tools, Experiments, First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, *Lecture Notes in Computer Science* 4171, Springer, pp. 460–469, doi:10.1007/978-3-540-69149-5_49.

[13] Ilya Sergey: *Programs and Proofs: Mechanizing Mathematics with Dependent Types*, doi:10.5281/zenodo.4996238. Lecture notes with exercises.

[14] Wouter Swierstra (2009): *A Hoare Logic for the State Monad*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, *Lecture Notes in Computer Science* 5674, Springer, pp. 440–451, doi:10.1007/978-3-642-03359-9_30.

[15] Glynn Winskel (1993): *The formal semantics of programming languages - an introduction*. Foundation of computing series, MIT Press, doi:10.7551/mitpress/3054.001.0001.