Defining Corecursive Functions in Coq Using Approximations

Vlad Rusu
Inria, Lille, France

David Nowak
Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France

Abstract

We present two methods for defining corecursive functions that go beyond what is accepted by the builtin corecursion mechanisms of the Coq proof assistant. This gain in expressiveness is obtained by using a combination of axioms from Coq's standard library that, to our best knowledge, do not introduce inconsistencies but enable reasoning in standard mathematics. Both methods view corecursive functions as limits of sequences of approximations, and both are based on a property of productiveness that, intuitively, requires that for each input, an arbitrarily close approximation of the corresponding output is eventually obtained. The first method uses Coq's builtin corecursive mechanisms in a non-standard way, while the second method uses none of the mechanisms but redefines them. Both methods are implemented in Coq and are illustrated with examples.

2012 ACM Subject Classification Theory of computation → Functional constructs

Keywords and phrases corecursive function, productiveness, approximation, Coq proof assistant

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2022.12

Supplementary Material *Software (ECOOP 2022 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.8.2.2`

1 Introduction

Coq [1] is a proof assistant based on the Calculus of Inductive Constructions. Coinductive constructions (coinductive types, relations, and proofs, and corecursive functions) have been included in Coq's underlying theory [11]. However, these constructions are limited. Corecursive functions must conform to a syntactical *guardedness* criterion requiring that, up to standard reductions, calls to the function under definition occur directly under *constructors* of the coinductive representing the codomain of the function of interest. Such functions are total and by consequence are accepted by Coq.

The guardedness criterion is best illustrated by an example. Consider the set of streams S over a set A, which, intuitively, are infinite sequences of elements of A separated by the *constructor* $\_\cdot\_$. The *head* (resp. the *tail*) of a stream s is the first element of s (resp. the stream obtained from s by removing its first element). Consider also a predicate p on A, and the following function *filter*, which takes a stream $s \in S$ as input and aims at producing as output a stream that contains the elements of s that satisfy p. Since its output is an (infinite) stream the function does not terminate.

$$\mathit{filter}\, s := \text{if } p(\mathit{head}\, s) \text{ then } (\mathit{head}\, s) \cdot (\mathit{filter}\,(\mathit{tail}\, s)) \text{ else } \mathit{filter}\,(\mathit{tail}\, s)$$

The first self-call to *filter* in the function's body falls directly under a call to the constructor $\_\cdot\_$. It is therefore syntactically guarded by the constructor. In the second self-call, the constructor is not present; the second call is not syntactically guarded. Overall, the above function definition fails to satisfy the syntactical "guarded-by-constructors" criterion because of the second self-call.

To see why the syntactical guardedness criterion is important, consider a stream in $s \in S$ such that none of its elements satisfy p. Then, *filter s* is not a stream – none of the elements in the input are kept in the output. The unguarded call is responsible for this. Hence, *filter* is *not* a *total* function from S to S, and Coq's guardedness criteria rightfully reject it because Coq only accepts total functions.

However, by restricting its domain to *the set S' of streams s' such that infinitely many elements of s' satisfy p*, the *filter* function becomes a total function from S' to S. Intuitively, the guarded call, which copies an element in the input into the output, is called infinitely many times and thus produces an (infinite) stream. Such a function could, in principle, be accepted by Coq; however, Coq does not have automatic mechanisms to realize this. Its builtin syntactical criteria are automatic, sound (i.e. all functions that fulfill them are total), but restricted since they reject some total functions.

Let us take a closer look at the argument for the totality of *filter* restricted to S'. Consider an arbitrary stream $s' \in S'$. At the beginning, i.e., before *filter* s' starts computing, obviously, nothing is known about the value of output. This continues to be the case while the function processes successive elements of s' that do not satisfy p, because such elements are not kept in the output. However, *eventually*, an element of s', say, a, which does satisfy p, is encountered. It is kept in the output, which becomes $a \cdot (filter\,(tail\ \ldots))$, i.e., a stream about which *something* is known: its first element. Thus, one starts with a situation for which nothing is known about the output, and, *eventually*, the first element of the output becomes known. By repeating these observations one can see that, *eventually*, any finite prefix of the output becomes known. By viewing a finite prefix of a stream as an approximation of the stream in question, and by interpreting a longer prefix as a closer approximation of a stream than a shorter prefix, we can rephrase the argument for the totality of our function as: *for each input, an arbitrarily close approximation of the corresponding output is eventually produced.* This condition is called *productiveness*, and it is the condition that our function (and, in general, corecursive functions) needs to fulfill in order to be total. We note that in the literature about corecursive functions this condition (under various formulations) is well known, to the point that it has become folklore; but, to our best knowledge, it has not yet been formalized.

What is, then, the relation between guardedness and productiveness? To see this, consider the function $filter'\,s :=$ if $p(head\,s)$ then $(head\,s) \cdot (filter'\,(tail\,s))$ else $dummy \cdot filter'\,(tail\,s)$, in which the second self-call is now also guarded by the constructor $\_ \cdot \_$ by having elements of the input that do not satisfy the predicate replaced by some dummy value in the output. The effect of this *guarded* definition is that for each input, the *next* call produces a closer approximation of the output. Since "next" is a particular case of "eventually", the overall effect of guardedness is to ensure productiveness and therefore totality, in a syntactical (thus, automatically checkable) and conservative way.

Contributions

In this paper we propose a formal definition of productiveness that captures the corresponding intuitive notion, and two methods for defining corecursive functions in which productiveness is a key ingredient. Essentially, productiveness restricts the manner in which a sequence of approximating functions converges to the function under definition, and the two methods offer two different ways for building the sequence of approximating functions. Both methods enable the definition of corecursive functions beyond what Coq accepts by default. Both methods have been implemented in Coq and are illustrated by examples. Their additional expressiveness is obtained thanks to axioms from Coq's standard library, which, to our best knowledge, do not introduce inconsistencies. The difference between the methods lies in the amount of Coq builtin coinductive features they reuse: the first method reuses them extensively, while the second method uses none but redefines them.The Coq development is available at `https://project.inria.fr/ecoop2022/`.

The rest of the paper is organized as follows. A theoretical part (Sections 2-4) presents a formal notion of productiveness and our two corecursive function-definition methods in a language-agnostic manner. We emphasize that knowing Coq is not necessary for understanding the theory. Section 5 gives details of the Coq implementation that are not visible in the theory but are essential in the implementation, such as the combination of axioms imported from the standard library that enable reasoning in standard mathematics. Section 6 concludes and discusses related and future work.

2 A formal notion of productiveness

We start with some basic definitions used in the rest of the paper. Consider a set C and a partial order $\preceq$ on C. We denote by $\prec$ the relation defined by $t \prec t'$ iff $t \preceq t'$ and $t \neq t'$.

▶ **Definition 1.** *A sequence $(s_i)_{i\in\mathbb{N}}$ of elements of C is*

- increasing *whenever for all $i \in \mathbb{N}$, $s_i \preceq s_{i+1}$;*
- strictly increasing*, whenever for all $i \in \mathbb{N}$, $s_i \prec s_{i+1}$;*
- stabilizing to $c \in C$ *whenever there exist $m \in \mathbb{N}$ such that for all $i \geq m$, $s_i = c$, and* stabilizing *whenever it is stabilizing to some $c \in C$;*
- ascending *whenever it is increasing and non-stabilizing.*

▶ Remark. A sequence is ascending iff it is increasing and has a strictly increasing subsequence.

The following is one of the several existing definitions of a complete partial order (CPO) :

▶ **Definition 2.** *A CPO consists of a set C, a partial order $\preceq$ on C, and an element $\bot \in C$ satisfying $\forall t \in T$, $\bot \preceq t$, such that that any increasing sequence of elements of T has a least upper bound.*

We call the least upper bound of an increasing sequence $(s_n)_{n\in\mathbb{N}}$ the *limit* of the sequence, hereafter denoted by $lim[(s_n)_{n\in\mathbb{N}}]$.

▶ **Example 3.** Any set A can be organized as a CPO $(A \cup \{\bot_A\}, \preceq_A, \bot_A)$ by choosing some value $\bot_A \notin A$ and by defining $\preceq_A$ as the smallest relation on $A \cup \{\bot_A\}$ satisfying $\bot_A \preceq_A a$ for all $a \in A$ and $a' \preceq_A a'$ for all $a' \in A \cup \{\bot_A\}$. The properties of orders (reflexivity, anti-symmetry, transitivity) hold trivially. Any increasing sequence $(a_n)_{n\in\mathbb{N}}$ stabilizes to some $a \in A \cup \{\bot_A\}$, and the limit of the sequence is the value to which the sequence stabilizes. This CPO is called the *flat CPO of A*.

In the rest of the paper the maximal elements of a CPO with respect to its order shall play an important role: that of "well-defined corecursive values". This view is consistently held ahead in the paper.

The following definition is our formal notion of productiveness. It restricts the manner in which a sequence of functions "converges" to a given function.

▶ **Definition 4.** *Given a sequence of functions $(f_n)_{n\in\mathbb{N}}$ having the same domain D and codomain C, such that the codomain is organized as a CPO $(C, \preceq, \bot)$, we say that the sequence $(f_n)_{n\in\mathbb{N}}$* productively converges *whenever for all $x \in D$, the sequence $(f_n\, x)_{n\in\mathbb{N}}$ is increasing and its limit $lim[(f_n\, x)_{n\in\mathbb{N}}]$ is maximal w.r.t. the order $\preceq$. The* limit *of the sequence $(f_n)_{n\in\mathbb{N}}$ is by definition the function $f : D \to C$ such that for all $x \in D$, $f\, x = lim[(f_n\, x)_{n\in\mathbb{N}}]$. We call $(f_n)_{n\in\mathbb{N}}$ the sequence of approximating functions for the limit function f.*

▶ Remark. The image of the domain D by functions f constructed as in Definition 4 is included in the set of maximal elements of C, which, as said earlier, play the role of well-defined corecursive values. This justifies us calling "corecursive" the limits of productively converging sequences $(f_n)_{n\in\mathbb{N}}$.

▶ Remark. We now justify why Definition 4 captures the informal definition of productiveness. For each $x \in D$, the increasing sequence $(f_n\, x)_{n\in\mathbb{N}}$ is either stabilizing or non-stabilizing. The values $x \in D$ for which $(f_n\, x)_{n\in\mathbb{N}}$ stabilizes are inputs on which f terminates. The values $x \in D$ for which $(f_n\, x)_{n\in\mathbb{N}}$ does not stabilize are such that the increasing sequence $(f_n\, x)_{n\in\mathbb{N}}$ is *ascending*: it has a strictly increasing subsequence $(f_{n_i}\, x)_{i\in\mathbb{N}}$ such that for all $i \in \mathbb{N}$, $f_{n_i}\, x \prec f_{n_{i+1}}\, x$, i.e., $f_{n_i}\, x$ and $f_{n_{i+1}}\, x$ both are approximations of the sequence's limit $f\, x$, but $f_{n_{i+1}}\, x$ is a strictly *closer* approximation of $f\, x$ than $f_{n_i}\, x$. Thus, $(f_n\, x)_{n\in\mathbb{N}}$ produces, as n grows, arbitrarily close approximations of the output $f\, x$. This captures the intuition of productiveness: the ability to eventually produce, for each input, arbitrarily close (and, in case of termination, exact) approximations of the corresponding output.

The next two sections present two methods for obtaining CPOs and corecursive functions as limits of sequences of approximating functions. The first method reuses as much as possible Coq's builtin mechanisms for corecursion. The second one replaces these mechanisms by other constructions.

3 First method

In this approach the carrier set of the CPO being defined is the set of terms of a type coinductively defined by Coq and the limits of increasing sequences in the CPO are Coq builtin corecursive functions. The approximating sequences for the corecursive functions under definition use a functional for the function in question. We illustrate the approach by defining the filter function on streams.

3.1 CPOs as coinductive types

▶ **Example 5.** The set S of streams (a.k.a infinite lists) over a base set $A \cup \{\bot_A\}$ can be organized as a CPO as follows. First, the flat CPO $(A \cup \{\bot_A\}, \preceq_A, \bot_A)$ is built as in Example 3. Then, the set S is defined to be the set of terms of a certain coinductive type, which, conceptually, are built by applying the following rule a countably infinite number of times : $a \cdot s \in S$ whenever $a \in (A \cup \{\bot_A\})$ and $s \in S$. This simultaneously defines the *constructor* function $\_\cdot\_ : (A \cup \{\bot_A\}) \times S \to S$.

Then, we define the constant stream $\bot \in S$ as the stream satisfying the equation $\bot = \bot_A \cdot \bot$. This is an example of a corecursive definition, which is accepted by Coq, since the occurrence of $\bot$ in the right-hand side is guarded by (a direct call to) the constructor $\_\cdot\_$. On the set S we define the functions *head* and *tail* by $head(a \cdot s) = a$ and $tail(a \cdot s) = s$. We also define the nth element of a stream by induction: $nth\, 0\, s = head\, s$ and $nth\,(n+1)\, s = nth\, n\,(tail\, s)$. Regarding the order relation $\preceq$, it is the relation on S defined "pointwise", by $s_1 \preceq s_2$ iff for all $n \in \mathbb{N}$, $nth\, n\, s_1 \preceq_A nth\, n\, s_2$.

Then, we define the limit $lim[(s_n)_{n\in N}]$ of an increasing sequence of streams $(s_n)_{n\in N}$ by $lim[(s_n)_{n\in N}] = (lim_A[(head\, s_n)_{n\in N}]) \cdot (lim[(tail\, s_n)_{n\in \mathbb{N}}])$. That is, the head of the limit of a sequence of streams is the limit (in $A \cup \{\bot_A\}$) of the heads of the streams in the sequence, and the tail of the limit is the limit (in S) of the tails of the streams in the sequence. This is another example of a corecursive function, and one that can be defined in Coq using the tool's builtin constructions for corecursion, because in the right-hand side of $lim[(s_n)_{n\in N}] = (lim_A[(head\, s_n)_{n\in N}]) \cdot (lim[(tail\, s_n)_{n\in \mathbb{N}}])$ the call to *lim* is guarded by the constructor $\_ \cdot \_$. In order to prove that the defined limit is indeed the least upper bound one can, e.g., reduce that property to a "pointwise" one, i.e., first proving that for all $m \in \mathbb{N}$, $nth\, m\, (lim[(s_n)_{n\in N})] = (lim_A[(nth\, m\ s_n)_{n\in N}])$ and then using the fact that lim_A computes least upper bounds in the flat CPO of A. Finally, we note that the limits of ascending sequences of streams over $A \cup \{\bot_A\}$ are streams over A (i.e., they do not contain any $\bot_A$), and are maximal with respect to $\preceq$.

▶ Remark. As illustrated by the above example, the maximal elements in CPOs play the role of "well defined" corecursive values, since they do not contain $\bot$ subterms, themselves interpreted as "undefined". The ascending sequences "push away" $\bot$ subterms, to the effect that, in their limit, all such subterms have been eliminated. Since $\bot$ is interpreted as "undefined", terms containing $\bot$ are "partially defined", and "pushing $\bot$ away" amounts to producing "better defined" values.

▶ Remark. The ability to define a CPO using Coq's builtin mechanisms relies on the ability of those mechanisms to accept the definitions of limits as corecursive functions. This works for many interesting coinductive datatypes (streams, colists, possibly infinite binary trees, . . .) but not in general. For coinductive datatypes that are mutually dependent with inductive datatypes, the limits may require corecursive functions that contain self-calls guarded not by constructors of the coinductive datatype, but by recursive functions on the inductive datatype. Such "improperly guarded" functions are rejected by Coq. A second method presented ahead in the paper deals with such difficult cases.

▶ Remark. The construction in Example 5 is not the only way to organize streams over a set A into a CPO. Another possibility is to define the set of streams S as the set obtained by applying the rules $\bot \in S$ and $a \cdot s \in S$ if $a \in A$ and $s \in S$ for a finite or a countably infinite number of times. In this definition $\bot$ is a constructor (unlike $\bot$ in Example 5 where it was a defined function). The order relation and the notion of limit are also slightly different. We chose the construction in Example 5 because it has fewer technical complications: for example, the *head* function is total in Example 5, but partial in the alternative construction, which makes it more complicated to define.

3.2 Approximating sequences using functionals

This method assumes a functional $F : (D \to C) \to D \to C$ for the function of interest. The functional may be obtained, e.g., from an attempt to define the function $f : D \to C$ of interest directly in Coq, via a statement of the form $f := F f$. It is, of course, assumed that the attempt failed – i.e., it failed the guardedness criteria – otherwise one would just define f directly in Coq.

▶ **Example 6.** Consider the set S of streams over $A \cup \{\bot_A\}$ as in Example 5 and assume a predicate $p : A \cup \{\bot_A\} \to \{true, false\}$ such that $p\, \bot_A = false$. Let D be the subset of S consisting of streams s over A, such that $p\,(nth\, n\, s) = true$ for infinitely many $n \in \mathbb{N}$. The following pseudocode statement is an attempt to define the *filter* function over D, which computes the substream of values satisfying p:

$$filter\, s := \text{if } p(head\, s) \text{ then } (head\, s) \cdot (filter\,(tail\, s)) \text{ else } filter\,(tail\, s)$$

Equivalently, the function could be defined by $filter := F\, filter$ where F is the pseudocode for the functional below, which takes a function as input and produces an (anonymous) function as output:

$$F\, f := \lambda s.\ \text{if } p(head\, s) \text{ then } (head\, s) \cdot (f\,(tail\, s)) \text{ else } f\,(tail\, s)$$

These definitions, when translated to Coq syntax, are rejected by the tool, because they fail the guardedness criterion: the call to *filter* in the "else" case is not guarded by the constructor $\_ \cdot \_$.

We show below an alternative way in which a functional F can be used for uniquely defining the function, say, f, of interest, while ensuring that the fixpoint equation $f = F\,f$ holds. Assume a CPO $(C, \preceq, \bot)$. We extend the order $\preceq$ from C to functions $D \to C$ by $f_1 \preceq f_2$ iff $f_1\,x \preceq f_2\,x$ for all $x \in D$.

▶ **Definition 7.** *A functional* $F : (D \to C) \to D \to C$ *is* increasing *if for all* $f_1, f_2 : D \to C$, $f_1 \preceq f_2$ *implies* $F f_1 \preceq F f_2$.

▶ **Example 8.** The functional F from the previous example is increasing. Indeed, consider two functions $f_1, f_2 : D \to S$, with D, S as in the example in question (in particular, S is organized as a CPO as in Example 5) and assume $f_1 \preceq f_2$. We have to show that $F\,f_1\,s \preceq F\,f_2\,s$ for all $s \in D$. If $p\,(\mathit{head}\,s) = \mathit{true}$ then $F\,f_1\,s = (\mathit{head}\,s) \cdot f_1\,(\mathit{tail}\,s)$ and $F\,f_2\,s = (\mathit{head}\,s) \cdot f_2\,(\mathit{tail}\,s)$; and $F\,f_1\,s \preceq F\,f_2\,s$ because $f_1 \preceq f_2$ implies in particular $f_1\,(\mathit{tail}\,s) \preceq f_2\,(\mathit{tail}\,s)$, and then $(\mathit{head}\,s) \cdot f_1\,(\mathit{tail}\,s) \preceq (\mathit{head}\,s) \cdot f_2\,(\mathit{tail}\,s)$ holds thanks to the definition of $\preceq$. If $p\,(\mathit{head}\,s) = \mathit{false}$ then $F\,f_1\,s = f_1\,(\mathit{tail}\,s)$, $F\,f_2\,s = f_2\,(\mathit{tail}\,s)$, and $F\,f_1\,s \preceq F\,f_2\,s$ is just $f_1\,(\mathit{tail}\,s) \preceq f_2\,(\mathit{tail}\,s)$, established above.

Assume again a CPO $(C, \preceq, \bot)$ and a functional $F : (D \to C) \to D \to C$. Let $\perp\!\!\!\perp : D \to C$ be the constant function such that $\perp\!\!\!\perp\ x = \bot$, for all $x \in D$, and let $F^n : (D \to C) \to D \to C$ be the functional inductively defined by $F^0 f = f$ and, for all $n \in \mathbb{N}$, $F^{n+1} f = F(F^n f)$.

▶ **Definition 9.** *A functional* $F : (D \to C) \to D \to C$ *is productive whenever it is increasing and the sequence of functions* $(F^n \perp\!\!\!\perp)_{n \in \mathbb{N}}$ *productively converges (cf. Definition 4).*

Calling *productive* a functional satisfying the above definition is justified by the fact that it generates a sequence of functions that productively converges. Its limit is characterized by the following theorem.

▶ **Theorem 10.** *If a functional* F *is productive then* $\mathit{lim}[(F^n \perp\!\!\!\perp)_{n \in \mathbb{N}}]$ *is the unique fixpoint of* F.

Proof. Let the type of the functional be $(D \to C) \to D \to C$. By Definition 4, for all $x \in D$, the sequence $(F^n \perp\!\!\!\perp\ x)_{n \in \mathbb{N}}$ is increasing and its limit is maximal w.r.t. $\preceq$. Hence, for all $x \in D$, $\mathit{lim}[(F^n \perp\!\!\!\perp\ x)_{n \in \mathbb{N}}]$ exists, and let $f : D \to C$ be defined by $f\,x = \mathit{lim}[(F^n \perp\!\!\!\perp\ x)_{n \in \mathbb{N}}]$ for all $x \in D$.

We first show that f is a fixpoint of F, i.e., $f = Ff$, which amounts to proving that for all $x \in D, f\,x = F\,f\,x$. We fix an arbitrary $x \in D$. By definition of f, $f\,x$ is maximal, hence, in order to prove $f\,x = F\,f\,x$ it is enough to prove $f\,x \preceq F\,f\,x$. Moreover $f\,x$ is the least upper bound of $(F^n \perp\!\!\!\perp\ x)_{n \in \mathbb{N}}$, hence, in order to show that $f\,x \preceq F\,f\,x$ it is enough to prove that $F\,f\,x$ is an upper bound of the sequence $(F^n \perp\!\!\!\perp\ x)_{n \in \mathbb{N}}$. This is proved by case analysis: For $n = 0$, $F^0 \perp\!\!\!\perp\ x = \bot \preceq F\,f\,x$, and, for $n > 0$, we have that for all $y \in D$, $F^{n-1} \perp\!\!\!\perp\ y \preceq f\,y$ because $f\,y$ is an upper bound for the sequence $(F^k \perp\!\!\!\perp\ y)_{k \in \mathbb{N}}$, which, since F is increasing, implies that for all $y \in D$, $F\,(F^{n-1} \perp\!\!\!\perp)\,y = F^n \perp\!\!\!\perp\ y \preceq F\,f\,y$. Setting $y := x$ in the previous relation proves that $F\,f\,x$ is an upper bound of the sequence $(F^n \perp\!\!\!\perp\ x)_{n \in \mathbb{N}}$ also in the case $n > 0$, and the proof of the fact that f is a fixpoint of F is completed.

Next, we show that f is the only fixpoint of F. Assume a solution f' of the fixpoint equation; we show $f = f'$. Note that it is enough to show $f \preceq f'$, i.e., $f\,x \preceq f'\,x$ for all $x \in D$, because from the latter by the maximality of $f\,x$ we obtain $f\,x = f'\,x$ for all $x \in D$, i.e., the desired $f = f'$.

Moreover, in order to prove that $f\,x \preceq f'\,x$ for all $x \in D$, it is enough to prove that $f'\,x$ is an upper bound for the sequence $(F^n \perp\!\!\perp\; x)_{n\in\mathbb{N}}$, because by definition $f\,x$ is the least upper bound of the sequence. Hence, what we have to prove is that for all $n \in N$, (for all $x \in D$, $F^n \perp\!\!\perp\; x \preceq f'\,x$), which is done by induction on n. The base case $n = 0$ is trivial, as it amounts to showing that for all $x \in D$, $\perp \preceq f'\,x$. In the inductive step, we have the inductive hypothesis that for all $x \in D$, $F^n \perp\!\!\perp\; x \preceq f'\,x$. By using the fact that F is increasing we obtain that for all $x \in D$, $F^{n+1} \perp\!\!\perp\; x = F\,(F^n \perp\!\!\perp)\,x \preceq F\,f'\,x = f'\,x$, which proves the inductive step. That was what remained to prove; the proof of the theorem is complete. ◀

The productiveness condition is more convenient to establish via the following sufficient conditions.

▶ **Definition 11.** *A CPO $(C, \preceq, \perp)$ is a CPO+ if each ascending sequence has a maximal limit.*

▶ **Example 12.** Per the observation at the end of Example 5, the CPO of streams is a CPO+.

▶ **Lemma 13.** *Assume a CPO+ $(C, \preceq, \perp)$ having the set of maximal elements $K \subseteq C$, and a functional $F : (D \to C) \to D \to C$. Then, F is productive whenever it is increasing and, for all $x \in D$:*

- *either there exists $n \in \mathbb{N}$ such that $F^n \perp\!\!\perp\; x \in K$;*
- *or, for all $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ with $n < m$ such that $F^n \perp\!\!\perp\; x \prec F^m \perp\!\!\perp\; x$.*

Proof. By Definitions 4 and 9, we have to show that for all $x \in D$, the sequence $(F^n \perp\!\!\perp\; x)_{n\in\mathbb{N}}$ has a limit in K. We first prove that the sequence is increasing, i.e., for all $n \in \mathbb{N}$, by induction on n. In the base case $n = 0$, we have, for each $x \in D$, $F^0 \perp\!\!\perp\; x = \perp\!\!\perp\; x = \perp \preceq F^1 \perp\!\!\perp\; x$, which settles this case. For the inductive step, we assume that for each $x \in D$, $F^n \perp \perp\; x \preceq F^{n+1} \perp\!\!\perp\; x$ and prove that, again for each $x \in D$, $F^{n+1} \perp\!\!\perp\; x \preceq F^{n+2} \perp\!\!\perp\; x$. We have $F^{n+1} \perp\!\!\perp\; x = F\,(F^n \perp\!\!\perp)\,x$ and since F is increasing, using the induction hypothesis $F\,(F^n \perp\!\!\perp)\,x \preceq F\,(F^{n+1} \perp\!\!\perp)\,x = F^{n+2} \perp\!\!\perp\; x$ holds for each $x \in D$, which proves the induction step and the fact that the sequence is increasing.

Hence, the sequence $(F^n \perp\!\!\perp\; x)_{n\in\mathbb{N}}$ has a limit; we just have to show the limit is in K.

- if there exists $n \in \mathbb{N}$ such that $F^n \perp\!\!\perp\; x \in K$, then, since the sequence is increasing and by definition of maximality, for all $m \geq n$, $F^m \perp\!\!\perp\; x = F^n \perp\!\!\perp\; x \in K$; and the limit is $F^n \perp\!\!\perp\; x \in K$ as required.
- if, for all $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ with $n < m$ such that $F^n \perp\!\!\perp\; x \prec F^m \perp\!\!\perp\; x$: we first note that each $F^n \perp\!\!\perp\; x$ must be in $C \setminus K$ (otherwise the hypothesis for this case is contradicted). Hence, the sequence has a strictly increasing subsequence, or, equivalently, the sequence is ascending. Since we have assumed that $(C, \preceq, \perp)$ is a CPO+ the limit of the sequence of interest is, again, in K. ◀

▶ **Example 14.** We prove using Lemma 13 that the functional $F : (D \to S) \to D \to S$ for the filter function from Example 6 is productive. We have already established that it is increasing and that the CPO $(S, \preceq, \perp)$ is a CPO+. We prove the condition at the second item the statement of Lemma 13, i.e., *for all $x \in D$ and $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ with $n < m$ such that $F^n \perp\!\!\perp\; x \prec F^m \perp\!\!\perp\; x$*, which amounts to finding a strictly increasing sequence of natural numbers $(n_i)_{i\in\mathbb{N}}$ such that $F^{n_i} \perp\!\!\perp\; x \prec F^{n_{i+1}} \perp\!\!\perp\; x$ for all $i \in \mathbb{N}$. This is where we use the fact that D is the set of streams x for which, given a predicate $p : (A \cup \{\perp_A\}) \to \{\mathit{true}, \mathit{false}\}$ on the base type of S with $p \perp_A = \mathit{false}$, it holds that $p\,(\mathit{nth}\,n\,x) = \mathit{true}$ for infinitely many $n \in \mathbb{N}$. We first prove by induction on n that $F^n \perp\!\!\perp = \mathit{ffilter}\,n$ where $\mathit{ffilter} = \lambda n.\lambda x.(\mathit{if}\ n = 0\ \mathit{then} \perp \mathit{else}\ (\mathit{if}\ p(\mathit{head}\,x)\ \mathit{then}\ (\mathit{head}\,x) \cdot (\mathit{ffilter}\,(n-1)\,(\mathit{tail}\,x))\ \mathit{else}\ \mathit{ffilter}\,(n-1)\,(\mathit{tail}\,x)))$ is a recursive, "finite approximation" of the corecursive *filter* function that we are trying to define. Then, we notice that if $p(\mathit{head}\,x) = \mathit{true}$ then $\mathit{ffilter}\,(n+1)\,x = (\mathit{head}\,x)\cdot(\mathit{ffilter}\,n\,(\mathit{tail}\,x))$, and if $p(\mathit{head}\,x) = \mathit{false}$ then $\mathit{ffilter}\,(n+1)\,x = \mathit{ffilter}\,n\,(\mathit{tail}\,x)$. That is, for the positions in the sequence x where p holds, the output of *ffilter* "grows", and for the positions where p does not hold, the output of *ffilter* stays the same. Finally, for all $i \in \mathbb{N}$, let n_i be i-th position where p holds in x; this gives us the strictly increasing sequence $(n_i)_{i\in\mathbb{N}}$ such that $F^{n_i} \perp\!\!\!\perp\ x = \mathit{ffilter}\,n_i\,x \prec \mathit{ffilter}\,n_{i+1}\,x = F^{n_{i+1}} \perp\!\!\!\perp\ x$ for all $i \in \mathbb{N}$. Hence, using Lemma 13 we have established that the functional $F = \lambda\,f.\lambda\,s.\ \mathit{if}\ p(\mathit{head}\,s)\ \mathit{then}\ (\mathit{head}\,s)\cdot(f\,(\mathit{tail}\,s))\ \mathit{else}\ f\,(\mathit{tail}\,s)$ is productive. Using Theorem 10 we obtain that F has a unique fixpoint; we call *filter* the fixpoint in question. The fixpoint equation states that $\mathit{filter}\,s = \mathit{if}\ p(\mathit{head}\,s)\ \mathit{then}\ (\mathit{head}\,s)\cdot(\mathit{filter}\,(\mathit{tail}\,s))\ \mathit{else}\ \mathit{filter}\,(\mathit{tail}\,s)$ for all $x \in D$. We note that D being the set of streams having infinitely many positions satisfying the filtering predicate is essential: outside this domain the functional F is not productive and one cannot use Theorem 10 as above to define the filter function.

Summarizing, what we have obtained in the present section is a method by which corecursive functions can be defined in Coq – details about the Coq implementation are given in Section 5 – even when the functions are not directly accepted by Coq because they do not satisfy Coq's builtin criteria for corecursive definitions. A function defined using our approach is abstract (it involves limits of ascending sequences in a certain CPO), but is the unique one satisfying the equation induced by its functional. We use the term "validation" for the process by which one can gain confidence that a given definition is the adequate one; one can reasonably claim that uniquely satisfying its fixpoint equation is the best validation possible for a corecursive function.

Finally, we note that from the user's point of view, by using our approach one gets the same result that one would have gotten if Coq had directly accepted the corecursive definition. Our definitions are not executable because they use axioms – i.e., the term $\mathit{filter}\,s$ is not automatically reduced to the term $\mathit{if}\ p(\mathit{head}\,s)\ \mathit{then}\ (\mathit{head}\,s)\cdot(\mathit{filter}\,(\mathit{tail}\,s))\ \mathit{else}\ \mathit{filter}\,(\mathit{tail}\,s)$ by Coq – but, in order to avoid nontermination, such reductions are not performed in Coq-builtin corecursive definitions either: one still has to prove a fixpoint equation and manually perform, e.g., rewriting with it in order to reduce it.

4 Second method

When the technique presented for in the previous section fails, we need to replace Coq's builtin mechanisms for coinduction, which no longer fulfill their role, by other constructions.

4.1 CPOs built by completion

The main idea is to start from the "finite subset" of the intended CPO and from an order relation on the given subset, and to "complete" them with values that are the equivalence classes of ascending sequences, according to a certain equivalence relation. We illustrate the notions introduced in this section by giving an alternative construction of a CPO of streams, different from the construction based on Coq's builtin mechanisms seen in the previous section. We also show an example where the present construction of a CPO is essential because Coq's builtin mechanisms for coinduction fail.

▶ **Definition 15.** *Given a set C° and an order $\preceq^\circ$ on it, a* measure *on $(C^\circ, \preceq^\circ)$ is a function $\mu : C^\circ \to \mathbb{N}$ such that for all $x, y \in C^\circ$, $x \prec^\circ y$ implies $\mu\,x < \mu\,y$.*

That is, that the measure is compatible with the relation $\prec^\circ$. It is then easy to prove that a measure is also compatible with $\preceq^\circ$: $x \preceq^\circ y$ implies $\mu\, x \leq \mu\, y$.

▶ **Example 16.** Consider the set L of finite lists over a base set A, inductively defined by the rules $nil \in L$ and $a \cdot l \in L$ whenever $a \in A$ and $l \in L$. Define an order on L by $l_1 \preceq^L l_2$ iff l_1 is a prefix of l_2. Then, the function *length* mapping each list to its length is a measure on $(L, \preceq^L)$.

▶ Remark. For ascending sequences $(s_n)_{n\in\mathbb{N}}$, the sequence $(\mu\, s_n)_{n\in\mathbb{N}}$ is a sequence of natural numbers that tends to infinity. This is the main reason why we chose natural numbers as measure values.

▶ **Definition 17.** *Two sequences $(s_n)_{n\in\mathbb{N}}$ and $(s'_n)_{n\in\mathbb{N}}$ of elements of C° are in the $\sim$ relation whenever for all $N \in \mathbb{N}$ there exist $n \in \mathbb{N}$ and $x \in C^\circ$ such that $x \preceq^\circ s_n$, $x \preceq^\circ s'_n$, and $\mu\, x \geq N$.*

A common predecessor of two elements is, in our approach, an "under-approximation" of the two elements. Thus, two sequences are in the $\sim$ relation whenever there there is a sequence of pointwise "under-approximations" of two sequences, whose measures tend to infinity. In some sense, the pointwise "difference" between the sequences intuitively tends to "nothing"[1]. In order to show that $\sim$ restricted to ascending sequences is an equivalence, the following property of an order is required:

▶ **Definition 18.** *An order $\sqsubseteq$ on a set A is weakly total whenever for all $a \in A$, the restriction of $\sqsubseteq$ to the set $\{a' \in A | a' \sqsubseteq a\}$ is total.*

▶ **Example 19.** The prefix order $\preceq^L$ on lists over a set A is weakly total: when l_1 and l_2 are both prefixes of a given list l then, if $length\, l_1 \leq length\, l_2$, $l_1 \preceq^L l_2$ holds, otherwise, $l_2 \preceq^l l_1$ holds. If A contains two elements $a_1 \neq a_2$, the order is not total, as $[a_1; a_2]$ and $[a_2; a_1]$ are incomparable.

▶ **Lemma 20.** *Assuming a set C° and a weakly total order $\preceq^\circ$ on C°, the restriction of the relation $\sim$ from Definition 17 to ascending sequences of elements of C° is an equivalence relation.*

Proof. For reflexivity, we use the fact that the sequence $(\mu\, s_n)_{n\in\mathbb{N}}$ of measures of an ascending sequence $(s_n)_{n\in\mathbb{N}}$ tends to infinity, hence, for each N there is $n \in \mathbb{N}$ such that $\mu\, s_n \geq N$, and we take $x := s_n$ in Definition 17 to show $(s_n)_{n\in\mathbb{N}} \sim (s_n)_{n\in\mathbb{N}}$. For symmetry, it is enough to note that Definition 17 is a symmetrical statement in $(s_n)_{n\in\mathbb{N}}$, $(s'_n)_{n\in\mathbb{N}}$. For transitivity assume $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$ and $(s'_n)_{n\in\mathbb{N}} \sim (s''_n)_{n\in\mathbb{N}}$. Fix an arbitrary $N \in \mathbb{N}$. By Definition 17, there exist $m, m' \in \mathbb{N}$ and $y, y' \in C^\circ$ such that $y \preceq^\circ s_m$, $y \preceq^\circ s'_m$, $y' \preceq^\circ s'_{m'}$, $y' \preceq^\circ s''_{m'}$, and $\mu\, y$, $\mu\, y' \geq N$. Since the sequences are increasing, we have $y, y' \preceq^\circ s'_{(max\, m\, m')}$ and since $\preceq^\circ$ is weakly total, $y \preceq^\circ y'$ or $y' \preceq^\circ y$. Assume $y \preceq^\circ y'$. Then, for the arbitrarily chosen N, we set $n := (max\, m\, m')$ and $x := y$ in Definition 17 and, since the sequences are increasing, we obtain $(s_n)_{n\in\mathbb{N}} \sim (s''_n)_{n\in\mathbb{N}}$. The other case $(y' \preceq^\circ y)$ is similar. ◀

The next lemma gives a useful sufficient condition for the equivalence of ascending sequences.

[1] This intuition can be formalized using a notion of distance, thus turning C° into a metric space. We have tried but discarded that approach because is complicates matters (one now has an order, a distance, and a measure, which have to satisfy certain properties) without any other benefit that perhaps a better intuition for the notion of equivalence.

▶ **Lemma 21.** *Given two ascending sequences $(s_n)_{n\in\mathbb{N}}$ and $(s'_n)_{n\in\mathbb{N}}$, if for all $k \in \mathbb{N}$ there exists $m \in \mathbb{N}$ such that $s_k \preceq s'_m$, then $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$.*

Proof. Fix an arbitrary $N \in \mathbb{N}$. Since $(s_n)_{n\in\mathbb{N}}$ is ascending, there exists $k \in \mathbb{N}$ such that $\mu\, s_k \geq N$. From the hypothesis we obtain $m \in \mathbb{N}$ such that $s_k \preceq^\circ s'_m$. Let $n := (max\, k\, m)$ and $x := s_k$. Since the sequences are increasing, $x \preceq s_n$, $x \preceq^\circ s'_n$, and from $\mu\, s_k \geq N$ we obtain $\mu\, x \geq N$. Hence, for all $N \in \mathbb{N}$ there are $n \in \mathbb{N}$, $x \in C^\circ$ such that $x \preceq^\circ s_n$, $x \preceq^\circ s'_n$, $\mu\, x \geq N$. By Def. 17, $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$. ◀

▶ Remark. The reverse implication in Lemma 21 does not hold in general: there exists sequences $(s_n)_{n\in\mathbb{N}}$ and $(s'_n)_{n\in\mathbb{N}}$ such that for all $n, m \in \mathbb{N}$, s_n and s'_m are incomparable, yet $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$ because the sequences have in common another sequence that pointwise under-approximates them and whose measure tends to infinity, i.e., they obey Definition 17. The latter is the proper definition of equivalence: if instead we had taken *for all $k \in \mathbb{N}$ there exists $m \in \mathbb{N}$ such that $s_k \preceq s'_m$* as in Lemma 21 we would be distinguishing certain sequences – namely, those that have a common sequence of under-approximations whose sizes tend to infinity, yet are pointwise incomparable – that should not be distinguished, because pointwise the difference between them becomes "negligible".

▶ **Definition 22.** *Assuming a set C° and a weakly total order relation $\preceq^\circ$ on C°, the* completion *of the set and its order to a set C and an order $\preceq$ on C are defined as follows:*

- *$C = C^\circ \cup K$, where K is the set of equivalence classes modulo $\sim$ of ascending sequences of elements in C°:*
- *$\preceq$ is the smallest relation on C satisfying*
  - *for all $x, y \in C^\circ$, $x \preceq y$ if $x \preceq^\circ y$;*
  - *for all $x, y \in K$, $x \preceq y$ if $x = y$;*
  - *for all $x \in C^\circ$ and $y \in K$, $x \preceq y$ if for all $(s_n)_{n\in\mathbb{N}} \in y$, there exists $m \in \mathbb{N}$ such that $x \preceq^\circ s_m$.*

This definition deserves a few comments. First, K is defined as equivalence classes of *ascending* sequences because, on the one hand, the sequences have to be increasing because they need to have limits – as we shall see, the set K will be a set of limits – and, on the other hand, they are non-stabilizing because if one sequence were stabilizing to a value, e.g., $v \in C^\circ$ then the limit (also v) of the sequence being also in K would imply a nonempty intersection of C° and K, which we wish to avoid. Second, the relation $\preceq$ is an order relation (this is established by Lemma 23 below). It is a conservative extension of $\preceq^\circ$, and elements in K are in the order iff they are equal. Combined with the fact that there is no situation in which $x \preceq y$ for $x \in K$ and $y \in C^\circ$, we obtain that the elements in K are maximal w.r.t. $\preceq$. Like in the case of the CPO of streams in an earlier example, the maximal elements play the role of "well-defined corecursive values". Finally, the third case defining the relation $\preceq$ requires an explanation. An element x (in C°) is in the order with an equivalence class y of ascending sequences (in K) whenever each sequence in the class "overtakes" x at some position $m \in \mathbb{N}$ according to the base relation $\preceq^\circ$. Combined with the fact that $(s_n)_{n\in\mathbb{N}}$ is increasing, this implies that the sequence overtakes x for all positions $n \geq m$. Several results hereafter (Lemma 24, Theorem 26, Theorem 32) critically depend on the proposed definition of the $\preceq$ relation.

▶ **Lemma 23.** *Assume a measure μ on $(C^\circ, \preceq^\circ)$ like in Definition 15, with $\preceq^\circ$ a weakly total order. Then, with C and $\preceq$ being the completions of C° and $\preceq^\circ$ respectively, given in Definition 22, the relation $\preceq$ on C is an order.*

Proof. Reflexivity is trivial since $\preceq$ amounts to $\preceq^\circ$ on C° and to equality on K, both of which are reflexive. For anti-symmetry, we note that it reduces to the anti-symmetry of $\preceq^\circ$, because the nontrivial remaining case has the form "$x \preceq y$ and $y \preceq x$ for $x \in C^\circ$ and $y \in K$ implies $x = y$", which holds because its premise $y \preceq x$ is impossible. Let us now consider transitivity, thus, $x \preceq y$ and $y \preceq z$. There are only four possibilities when those relations can hold:

1. $x, y, z \in C^\circ$, in which case the transitivity of $\preceq$ reduces to that of $\preceq^\circ$;
2. $x, y \in C^\circ$ and $z \in K$, which implies $x \preceq^\circ y$ and, given the definition of $y \preceq z$ for x an element and z a equivalence class of ascending sequences, from the fact that any sequence in z overtakes y at some position, we obtain thanks to $x \preceq^\circ y$ that the sequence also overtakes x at the same position, which implies $x \preceq z$ and settles this case;
3. $x \in C^\circ$ and $y, z \in K$: then, $y \preceq z$ implies $y = z$, and transitivity follows easily;
4. $x, y, z \in K$, in which case the transitivity of $\preceq$ follows from that of equality. ◀

The following lemma gives a useful alternative definition for the order $\preceq$ in a particular case.

▶ **Lemma 24.** *For all $x \in C^\circ$ and ascending sequences $(s_n)_{n\in\mathbb{N}}$ of elements of C°, $x \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$ iff there exists $m \in \mathbb{N}$ such that $x \preceq s_m$.*

Proof. By Definition 22, $x \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$ means: for all $(s'_n)_{n\in\mathbb{N}} \in [(s_n)_{n\in\mathbb{N}}]_\sim$, there exists $m \in \mathbb{N}$ such that $x \preceq^\circ s'_n$. The "only if" direction is trivial since obviously $(s_n)_{n\in\mathbb{N}} \in [(s_n)_{n\in\mathbb{N}}]_\sim$. We thus focus on the "if" direction. By hypothesis, there exists $m \in \mathbb{N}$ such that $x \preceq^\circ s_m$. Choose an arbitrary $(s'_n)_{n\in\mathbb{N}} \in [(s_n)_{n\in\mathbb{N}}]_\sim$, i.e., $(s'_n)_{n\in\mathbb{N}} \sim (s_n)_{n\in\mathbb{N}}$. By Definition 17, there exists $x' \in C^\circ$ and $m' \in \mathbb{N}$ such that $x' \preceq^\circ s_{m'}$, $x' \preceq^\circ s'_{m'}$ and $\mu\, x' > \mu\, x$. Since the sequences are increasing, we obtain $x, x' \preceq^\circ s_{(max\, m,\, m')}$. From the latter and the weak totality of $\preceq^\circ$ we obtain $x \preceq^\circ x'$ or $x' \preceq^\circ x$. But $x' \preceq^\circ x$ contradicts the established $\mu\, x' > \mu\, x$. Hence, $x \preceq^\circ x'$ and then $x \preceq^\circ s'_{m'}$ follows from $x' \preceq^\circ s'_{m'}$ by transitivity. Summarizing, for the arbitrarily chosen sequence $(s'_n)_{n\in\mathbb{N}} \in [(s_n)_{n\in\mathbb{N}}]_\sim$ we found $m' \in \mathbb{N}$ such that $x \preceq^\circ s'_{m'}$ř. But this is $x \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$ by definition; which proves the lemma. ◀

▶ **Definition 25.** *Given a set C° and weakly total order $\preceq^\circ$ on C°, consider the completion of C° to C and of $\preceq^\circ$ to $\preceq$ as in Definition 22. For an increasing sequence $(s_n)_{n\in\mathbb{N}}$ of elements of C, we define $lim[(s_n)_{n\in\mathbb{N}}]$ as follows:*

- *if the sequence stabilizes at a value, say, $v \in C$, then $lim[(s_n)_{n\in\mathbb{N}}] = v$;*
- *otherwise, the sequence does not stabilize, which implies that for all $n \in \mathbb{N}$, $s_n \in C^\circ$, and we define $lim[(s_n)_{n\in\mathbb{N}}] = [(s_n)_{n\in\mathbb{N}}]_\sim$, i.e., the equivalence class of the sequence w.r.t. the relation $\sim$.*

Note that in the second case of the above definition it is essential that the ascending sequence $(s_n)_{n\in\mathbb{N}}$ be composed of elements of C° because $\sim$ is only an equivalence for such sequences.

▶ **Theorem 26.** *Assume a measure on $(C^\circ, \preceq^\circ)$ like in Definition 15, with $\preceq^\circ$ a weakly total order. Then, with C and $\preceq$ being the completions of C° and $\preceq^\circ$ respectively, given in Definition 22, and with the limits of increasing sequences introduced in Definition 25, the triple $(C, \preceq, \bot)$ is a CPO.*

Proof. In order to prove the theorem we have to prove that the limits of increasing sequences proposed in Definition 25 are least upper bounds. Consider an increasing sequence $(s_n)_{n\in\mathbb{N}}$ of elements of C.

- if the sequence stabilizes to some value $v \in C$ then the proposed limit v is an upper bound for the (increasing) sequence. To show that it is the least such bound, assume another upper bound w; then, in particular, $v \preceq w$ because v is an element of the sequence.

- if the sequence does not stabilize then it is ascending, and as already observed before, $s_n \in C^\circ$ for all $n \in \mathbb{N}$, and the proposed limit is the equivalence class $[(s_n)_{n\in\mathbb{N}}]_\sim$.
  - We first show that $[(s_n)_{n\in\mathbb{N}}]_\sim$ is an upper bound for $(s_n)_{n\in\mathbb{N}}$: $s_k \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$ for all $k \in \mathbb{N}$. We apply Lemma 24 with $x := s_k$: there exists $m := k$ such $s_k \preceq s_m$, which implies $s_k \preceq [(s_n)_{n\in\mathbb{N}}]_\sim$.
  - Then, we show that $[(s_n)_{n\in\mathbb{N}}]_\sim$ is the least upper bound for $(s_n)_{n\in\mathbb{N}}$. Assume any upper bound $w \in C$, thus, $s_k \preceq w$ for all $k \in \mathbb{N}$. Suppose first that $w \in C^\circ$. Since $(s_n)_{n\in\mathbb{N}}$ is ascending, it has a strictly increasing subsequence $(s_{n_i})_{i\in\mathbb{N}}$. Now, w is also an upper bound for the subsequence, hence, $s_{n_i} \preceq w$ for all $i \in \mathbb{N}$, and due to the properties of the measure, $\mu\, s_{n_i} \preceq \mu\, w$ for all $i \in \mathbb{N}$. But this is impossible, since the sequence of measures of a strictly increasing sequence is a strictly increasing sequence of natural numbers, which tends to infinity. Hence, $w \in C^\circ$ is impossible. It follows that $w \in K$, i.e., $w = [(s'_n)_{n\in\mathbb{N}}]_\sim$ for some ascending sequence $(s'_n)_{n\in\mathbb{N}}$. From our hypothesis $s_k \preceq w$ for all $k \in \mathbb{N}$, we obtain that for all $k \in \mathbb{N}$ there exists $m \in \mathbb{N}$ such that $s_k \preceq^\circ s'_m$. Using Lemma 21, $(s_n)_{n\in\mathbb{N}} \sim (s'_n)_{n\in\mathbb{N}}$, i.e., $[(s_n)_{n\in\mathbb{N}}]_\sim = [(s'_n)_{n\in\mathbb{N}}]_\sim = w$ and in particular $[(s_n)_{n\in\mathbb{N}}]_\sim \preceq w$. Since the upper bound w was chosen arbitrarily, we have proved that $[(s_n)_{n\in\mathbb{N}}]_\sim$ is the least upper bound for $(s_n)_{n\in\mathbb{N}}$. The proof of the theorem is complete. ◀

▶ **Example 27.** Going back to the example of finite lists, their prefix order, and the measure defined by lengths of lists, the constructions in this section enable us to build a CPO of lists and streams. The streams are not defined by Coq corecursive functions (as in the earlier construction in Section 3) but by equivalence classes of ascending sequences of lists. One important difference in practice is that, unlike the approach in Section 3, the constructor $\_ \cdot \_$ is not directly available for streams, and the functions *head* and *tail* do not have simple definitions. All three functions can be defined, and the standard relations between them can be proved, with some effort; but having them readily available as in the approach from Section 3 is preferable. We now give an example where that approach fails.

▶ **Example 28.** The set T of Rose trees over a set A is coinductively definable in Coq by the rules $\bot \in T$ and $tree\, a\, l \in T$ whenever $a \in A$ and l is a list over T. Note the mixture of coinduction and induction: the trees are defined coinductively, but their definition relies on inductively defined lists.

When $t = tree\, a\, l$ we define $label\, t = a$ and $forest\, t = l$; when $t = \bot$ we define $label\, t = \bot_A$ (the least element in the flat CPO of A) and let $forest\, t$ be the singleton $[\bot_A]$. Assuming an order $\preceq^T$ on T, the limit of an increasing sequence $(t_n)_{n\in\mathbb{N}}$ of Rose trees would naturally be defined as $lim[(t_n)_{n\in\mathbb{N}}] = \bot$ if $t_n = \bot$ for all $n \in \mathbb{N}$ and $lim[(t_n)_{n\in\mathbb{N}}] = tree\, (lim_A[(label\, t_n)_{n\in\mathbb{N}}])\, (map\, lim\, (forest\, t_n)_{n\in\mathbb{N}})$ otherwise. This corecursive definition of limits is not guarded by constructors, since the corecursive call to *lim* occurs under the *map* function, which is not a constructor but a defined function. Hence, the definition of limits of increasing sequences of Rose trees is rejected by Coq, and without limits there is no CPO.

▶ **Example 29.** One can define and organize the set $T = F \cup R$, with F the set of finite trees and R that of Rose trees, in a CPO using the approach described in this section. We first define the finite trees F over A inductively, by the rules $\bot \in F$ and $tree\, a\, l \in F$ whenever $a \in A$ and l is a list over F. The measure function is the tree's height, recursively defined by $height\, \bot = 0$ and $height\, (tree\, a\, l) = 1 + max\, (map\, height\, l)$ where max computes the maximum value in a list of natural numbers. The order relation is based on the following recursive function, whose effect is to "cut" a given finite tree t at given depth n:

$cut = \lambda n.\lambda t.\textit{if}\ n = 0\ \textit{or}\ t = \bot\ \textit{then}\ \bot\ \textit{else}\ \textit{tree}\,(\textit{label}\,t)\,(\textit{map}\,(\textit{cut}\,(n-1))\,(\textit{forest}\,t))$; we then define the order relation $\preceq^F$ by $t_1 \preceq^F t_2$ whenever $t_1 = \textit{cut}(\textit{height}\,t_1)\,t_2$. The set R of Rose trees consists of equivalence classes of ascending sequences of finite trees. We have proved in Coq that all the requirements presented earlier in this section for obtaining a CPO for $T = F \cup R$ hold.

By contrast, a perhaps more natural definition of the "prefix order" $\preceq'^F$ by $t_1 \preceq'^F t_2$ *whenever* $t_1 = \bot$ *or* $t_1 = \textit{tree}\,a\,l_1$, $t_2 = \textit{tree}\,a\,l_2$, $\textit{length}\,l_1 = \textit{length}\,l_2$ *and for all* $n < \textit{length}\,l_1$, $\textit{nth}\,n\,l_1 \preceq'^F \textit{nth}\,n\,l_2$ fails to meet the critical weak totality requirement (Definition 18). Indeed, e.g., for $t', t'' \neq \bot$, $t = \textit{tree}\,a\,[t', t'']$, $t_1 = \textit{tree}\,a\,[t', \bot]$ and $t_2 = \textit{tree}\,a\,[\bot, t'']$ satisfy $t_1 \preceq'^F t$ and $t_2 \preceq'^F t$, yet t_1 and t_2 are incomparable. Without weak totality there is no sequence equivalence and ultimately no CPO[2].

4.2 Approximating sequences without functionals

In Section 3.2 the approximating sequence $(f_n)_{n\in\mathbb{N}}$ for defining a function was defined using a functional, which used functions over streams (such as the constructor $\_ \cdot \_$) that were readily available in Coq, due to the fact that the CPO for streams had been defined as a builtin Coq coinductive type. However, in the case of CPOs built by completion, such constructors are no longer available. One can try to replace them by defined functions, but this may turn out to be excessively difficult. For instance, in the CPO of Example 29, extending the constructor *tree* from finite trees F to a fully defined function $\textit{tree} : A \to \textit{list}\,T \to T$, with $T = F \cup R$, is difficult: each of the trees in its second argument of type $\textit{list}\,T$ may be a finite tree in F or a Rose tree in R – an equivalence class of ascending sequences of elements in F. Even when all elements in the list are equivalence classes, it is not clear how the result – again, an equivalence class of ascending sequence of elements in F – can be built.

Hence, we have to make do without constructors or defined functions replacing them. This severely limits the functionals that one may write, making the functional-based definition of corecursive functions from Section 3.2 essentially useless. Example 31 below illustrates this issue. In this section we present an approach that does not require a functional, but does require a "finite version" f° of the corecursive function f under definition, which moreover has to satisfy a productiveness requirement.

▶ **Definition 30.** *Assume two CPOs* $(D, \preceq_D, \bot_D)$ *and* $(C\ \preceq_C, \bot_C)$ *defined as in Theorem 26, thus, their base sets are decomposed as* $C = C^\circ \cup K_C$ *and* $D = D^\circ \cup K_D$*. Then, a function* $f^\circ : D^\circ \to C^\circ$ *is* productive *whenever, for all increasing sequences* $(x_n)_{n\in\mathbb{N}}$ *of elements in* D° *that have a limit in* K_D*, the sequence* $(f^\circ x_n)_{n\in\mathbb{N}}$ *of elements in* C° *is also increasing and has a limit in* K_C*.*

▶ Remark. Definition 30 of a productive function implies the function is also increasing. It also implies that the function maps ascending sequences to ascending sequences. Calling such a function *productive* is justified by the fact that it generates a sequence of functions that productively converges according to Definition 4. This sequence is built as follows: for all $x \in K_D$, choose an arbitrary ascending sequence $(x_n)_{n\in\mathbb{N}} \in x$; and set $(f_n\,x) = (f^\circ x_n)$ for all $n \in \mathbb{N}$. Then, $(f_n)_{n\in\mathbb{N}}$ productively converges according to Definition 4: for all $x \in K_D$, $(f_n\,x)_{n\in\mathbb{N}}$ is increasing and its limit is in K_C.

[2] Our earlier attempt with metric spaces also required a weakly total order for obtaining a proper notion of distance.

▶ **Example 31.** In the CPO of finite and Rose trees from Example 29, the sets C° and D° from the above definition are both the set F of finite trees. Consider the following recursive endofunction of F: $mirror^\circ = \lambda t. if\, t = \bot\ then\ \bot\ else\ tree\,(label\, t)(map\, mirror^\circ\ (rev\,(forest\, t)))$, where rev is the function that computes the reverse of a list. As its name indicates, the function computes the "mirror image" of finite trees. We have defined this function in Coq and have proved that it is productive according to Definition 30, using the fact that the $mirror^\circ$ function preserves the $height$ of its argument.

Note how the functional for $mirror^\circ$: $\lambda\phi\, t. if\, t = \bot\ then\, \bot\ else\ tree\,(label\, t)(map\, \phi\,(rev\,(forest\, t)))$ uses the constructor *tree*. Writing the corresponding functional for a full *mirror* function for both finite and Rose trees would require a corresponding defined function $tree : A \to list\, T \to T$. As stated above, such a function is hard to define. Hence our alternative solution avoiding these issues.

The following theorem states that productive functions map equivalent ascending sequences in their domain to equivalent ascending sequences in their codomain.

▶ **Theorem 32.** *In the context of Definition 30, let $\sim_D$ denote the equivalence relation on ascending sequences in the CPO $(D, \preceq_D, \bot_D)$ (cf. Definition 17, Lemma 20). Let $\sim_C$ denote the corresponding equivalence in $(C\ \preceq_C, \bot_C)$. Then, for any pair of equivalent ascending sequences $(s_n)_{n\in\mathbb{N}} \sim_D (s'_n)_{n\in\mathbb{N}}$ and any productive function $f^\circ : D^\circ \to C^\circ$, we have the equivalence $(f^\circ s_n)_{n\in\mathbb{N}} \sim_C (f^\circ s'_n)_{n\in\mathbb{N}}$.*

Proof. By Definition 22 and Theorem 26 the equivalence class $[(s_n)_{n\in\mathbb{N}}]_{\sim_D}$ is the least upper bound of $(s_n)_{n\in\mathbb{N}}$ and the equivalence class $[(s'_n)_{n\in\mathbb{N}}]_{\sim_D}$ is the least upper bound of $(s'_n)_{n\in\mathbb{N}}$. Since the two sequences are equivalent, we have the equality $[(s_n)_{n\in\mathbb{N}}]_{\sim_D} = [(s'_n)_{n\in\mathbb{N}}]_{\sim_D}$. In particular, it follows that $[(s'_n)_{n\in\mathbb{N}}]_{\sim_D}$ is an upper bound for $(s_n)_{n\in\mathbb{N}}$, thus for all $n \in \mathbb{N}$, $s_n \preceq_D [(s'_n)_{n\in\mathbb{N}}]_{\sim_D}$ and by Lemma 24, (i): *for all $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ such that $s_n \preceq^\circ_D s'_m$.* Since f° is productive, it is also increasing, and thus from (i) we obtain (ii): *for all $n \in \mathbb{N}$, there exists $m \in \mathbb{N}$ such that $f^\circ s_n \preceq^\circ_C f^\circ s'_m$.* Using Lemma 21 we obtain $(f^\circ s_m)_{m\in\mathbb{N}} \sim_C (f^\circ s'_m)_{m\in\mathbb{N}}$, which proves the lemma. ◀

The above result enables us to define functions $f : K_D \to C$ as limits of approximating sequences $(f_n : K_D \to C)_{n\in\mathbb{N}}$. The definition of each of the functions f_n below depends on an arbitrary choice for a representative in its argument (which is an equivalence class), however, thanks to the above theorem, the limit (i.e. the defined function f) does not depend on that choice. The functions are built as in the Remark following Definition 30: for all $x \in K_D$, choose an arbitrary ascending sequence $(x_n)_{n\in\mathbb{N}} \in x$; and set $(f_n\, x) = (f^\circ x_n)$ for all $n \in \mathbb{N}$. We have observed that $(f_n)_{n\in\mathbb{N}}$ productively converges according to Definition 4; its limit is $[(f^\circ x_n)_{\in\mathbb{N}}]_{\sim_C} \in K_C$, which, by above theorem, is independent of the choice for $(x_n)_{n\in\mathbb{N}} \in x$. Hence, the limit $f := \lambda x. lim[(f_n\, x)_{n\in\mathbb{N}}]$ is also independent on the initial choice.

Of course, the natural question that arises regards validation: do the functions thus defined match the intention of the user? Unlike the case of functional-based corecursive functions in an earlier section, we do not have a functional and a fixpoint equation as validation mechanisms. A certain degree of confidence in the definition of f is already obtained from the (assumed) confidence in f° – as we have $f([(x_n)_{n\in\mathbb{N}}]_{\sim_D}) = [(f^\circ x_n)_{n\in\mathbb{N}}]_{\sim_C}$ – and from the independence of choice of representative from Theorem 32. The confidence can be improved by proving properties of f that the user expects.

▶ **Example 33.** By applying the above process to the $mirror^\circ$ function from Example 31, and noting that in this case $K_D = K_C = R$ (the set of Rose trees), we obtain a well-defined function $mirror : R \to R$. To increase confidence in this function we prove that the *mirror* function "reverses" positions in Rose trees. A position p is a finite sequence of natural numbers, and in a given tree t it indicates the label (in the set $A \cup \bot_A$, if we consider trees over A) obtained by navigating in the tree, starting from the root and choosing children indexed by the successive numbers in p. Let $pos\, p\, t$ be the label in question (or $\bot_A$, since the position may "overflow"). A function pos_rev is also defined, which like pos takes a position p and a tree t and navigates the tree from root to children, but unlike pos, the children are chosen "backwards" (counting back starting from the last child) instead of forwards. We have then proved that for all positions p and trees t (finite or Rose), $pos\, p\, (mirror\, t) = pos\_rev\, p\, t$, meaning that, intuitively, $mirror$ "reverses" all positions in the tree. The proofs were performed by first defining finite versions pos° and pos_rev° for the new functions involved, then proving $pos^\circ p\, (mirror^\circ f) = pos\_rev^\circ p\, f$ for finite trees f, and finally proving that the corresponding property on Rose trees reduces to that on finite trees whose height is large enough (here, larger the length of the list p). The Coq proofs are available in the companion Coq development.

5 Implementation

The corecursive function-definition methods presented in Sections 2–4 have been implemented in the Coq proof assistant. The implementation has two motivations. The first one is ensuring that the results are sound, i.e., no case has been forgotten in a proof, and no assumption was left implicit. This is a standard motivation for using a proof assistant. The second motivation aims at providing Coq itself with stronger mechanisms for corecursive definitions than the builtin ones available in the tool. This is achieved at the cost of assuming several axioms from Coq's standard library; we state which axioms were used, where, and for what purpose. To our best knowledge the combination of axioms we imported from the standard library does not introduce inconsistencies (cf. [8, Chapter 12]).

Understanding the rest of this section requires knowledge about Coq's inductive and coinductive types, recursive and corecursive definitions, and its module system.

5.1 Sequences

Some notions are used by both methods. The main concept is that of sequences over a given type, encoded as functions from the natural numbers to the type in question. The fact that an element belongs to a sequence (parameterized by a given type) is also defined using an existential quantifier.

```
Definition Seq {A:Type}:Type := nat -> A
Definition sin{A:Type}(a:A)(q: Seq(A:=A)):Prop := exists i, a = q i.
```

Then, given a relation `R` (i.e., a binary predicate, of type `A->A->Prop`), the various kinds of sequences from Definition 1 (increasing, strictly increasing, stabilizing, ascending) are defined. Next, the fact that a value `lub_val` is the least upper bound (w.r.t. a relation `R`) of a sequence `q` is defined as

```
Definition lub{A:Type} (R:A-> A-> Prop) (lub_val: A) (q:Seq(A:=A)) :=
(forall a, sin a q -> R a lub_val) /\
(forall lub_val',(forall a,sin a q-> R a lub_val')-> R lub_val lub_val').
```

The definition of `lub` is only relevant for order relations `R`, and will only be used for such relations. For the first method these definitions are enough. The second method requires the property noted in the Remark following Definition 1: if `R` is an order, then a sequence is ascending if and only if it is increasing and has a strictly increasing sequence. This one-line property required quite a few intermediary lemmas in order to be formally proved, using classical logic and the following axiom:

```
Axiom constructive_indefinite_description: forall (A:Type) (P:A->Prop),
(exists x, P x) -> {x:A | P x}.
```

This axiom, from Coq's standard library, enables one to "choose" an element `P` satisfying a predicate `P` just based on the knowledge that `P` is satisfiable. In informal mathematical reasoning this is often implicitly assumed. In a Coq formal development, however, it has to be explicitly assumed. Here we use it in order to turn a total relation into a function having the relation in question as its graph:

```
Lemma functional_choice: forall (A B:Type) (R:A->B->Prop),
(forall x:A,exists y:B, R x y)->(exists f:A->B,forall x:A, R x (f x)).
```

The `constructive_indefinite_description` axiom occurs several times in the Coq development.

▶ Remark. We have not formalized Definition 4 of productive convergence of sequences of functions. That definition is useful in the paper for a unified presentation of the two methods and for giving the intuitive notion of productiveness a mathematical meaning. In Coq these motivations do not apply.

5.2 First method

This method reuses Coq's builtin coinduction mechanisms for organizing coinductive types as CPOs.

5.2.1 Stream CPO

The Coq definition for the stream CPO closely follows the approach outlined in Example 5. First, the flat CPO over a given type `A` (cf. Example 3) is encoded using Coq's `option` type. A relation `leo` on this type is also defined, which we prove to be an order relation, having `None` as the bottom element:

```
Inductive option(A:Type): Type :=  None: option A | Some: A -> option A.
Inductive leo{A:Type} : option A -> option A -> Prop :=
|leo_none: forall a, leo None a
|leo_some: forall a, leo (Some a) (Some a)
```

In Example 3, `None` was denoted by $\bot_A$ and `leo` was denoted by the infix symbol $\_ \preceq_A \_$. Then, a lemma states that for each increasing sequence in the `leo` order, there exists a least upper bound:

```
Lemma leo_lub{A:Type} :
forall (q:Seq (A:=option A)),increasing leo q-> exists b,lub leo b q.
```

The least upper bound of a sequence is obtained using `constructive_indefinite_description`:

```
Definition limF{A:Type}(q:Seq(A:=option A))(H:increasing leo q):=
  constructive_indefinite_description _ (leo_lub q H).
```

Next, a stream over a type `T` is obtained by applying the constructor `scons` to an element in `T` and another stream over `T`. The stream `bot`, which is an infinite repetition of `None`, is also defined.

```
CoInductive Stream{T:Type} :=  scons : T -> Stream -> Stream.
CoFixpoint bot{T:Type}: Stream(T:=T) :=  scons None bot.
```

In Example 5 the constructor `scons` is denoted by an infix operation $\_ \cdot \_$ and `bot` is denoted by $\bot$. The head (`hd`) and tail (`tl`) of a stream are also defined, in the expected manner.

Next comes the order relation on streams. In Example 5 the order $\_ \preceq \_$ was defined pointwise. We here give an alternative, coinductive definition, and prove that the two definitions are equivalent.

```
CoInductive les{T:Type} :Stream(T:=T)-> Stream (T:=T)-> Prop :=
les_def:forall a b s s', leo a b-> les s s'-> les(scons a s)(scons b s').
```

Next, the limit of an increasing sequence of streams over the flat CPO of a given set is defined by:

```
CoFixpoint lim{A:Type}(q:Seq(A:=Stream(T:=option A)))(H:increasing les q)
:= scons
 (proj1_sig(limF(fun n => hd(q n))(increasing_smap_hd q H)))
 (lim(fun n => tl(q n))(increasing_smap_tl q H)).
```

The function is parameterized by base type `A` of the underlying flat CPO. The type of the argument `q` is a sequence of streams over the flat order of `A`. The function takes a second argument: a proof that the sequence is increasing w.r.t. the order `les` of streams. The function returns a stream, built with `scons`, whose head is the limit (`limF`, in the flat CPO) of a stream that consists in mapping the head of streams to the sequence `q`, and whose tail is the (corecursively called) limit of the sequence of streams obtained by mapping the tail of streams to the sequence `q`. There are also some proof terms being used for ensuring that the various sequences whose limits are being invoked are increasing. Finally, we prove that the proposed limit is the actual least upper bound of an increasing sequence:

```
Lemma lim_lub{A:Type}(q:Seq(A:=Stream(T:=option A)))(H:increasing les q):
lub les (lim q H) q.
```

We also formalize the main artifact in the first method for corecursive function definition – Theorem 10, which says that a productive functional has a unique fixpoint. For productiveness we use the more convenient sufficient conditions given by Lemma 13. These conditions are placed in a Coq *module type*, which can be seen as an interface that other modules need to implement in order to benefit from the results implied by the conditions (here, the function definition method embodied in Theorem 10).

5.2.2 The filter function on streams

The proposed functional for the filter function for streams over a type `A` is written as follows:

```
Definition Filter(f:S->Stream(T:=option A))(s: S): Stream(T:=option A):=
if P (head s) then
scons (head s) (f (tail s))
else f (tail s).
```

where `S:= {s:Stream(T:=option A)|forall n,exists m,n<=m` $\wedge$ `P(nth m q)=true}` is the subtype of streams that have an infinite number of elements satisfying the filtering predicate `P: option A-> bool`, and `head`, `tail` are the restrictions of the `hd`, resp. `tl` functions on streams to the subtype `S`. We prove the conditions in Lemma 13, which enables us to use the Coq formalization of Theorem 10 and to define a function `filter` satisfying the two following theorems:

```
Theorem filter_fix: forall  s, bisim (filter s)(Filter (filter s))
Theorem filter_fix_unique: forall f,(forall s,bisim (f s) (Filter f s))->
 forall s,bisim  (filter s)(f s).
```

The theorems state the existence and uniqueness of `filter` as the unique fixpoint of `Filter`... except for the fact that instead of the expected equality we get bisimulation, coinductively defined as follows:

```
CoInductive bisim{T: Type}:Stream(T:=T) -> Stream(T:=T) -> Prop :=
|bisim_def: forall a s1 s2, bisim s1 s2 -> bisim(scons a s1)(scons a s2).
```

In the presentation of the first function-definition method from Section 3 we allowed ourselves, for simplicity of notation, to use equality instead of bisimulation. When translating informal mathematical reasoning to Coq such notation abuses and other similar approximations are revealed. Bisimulation is the natural equality between streams; by contrast, the standard equality of Coq is too strong. We note that, after having proved that bisimulation is a congruence relation, by importing a certain library (`Setoid`) one can perform rewriting with the fixpoint "equation" `filter_fix` in Coq.

Other examples

The companion Coq development also contains a definition of the stream of Fibonacci numbers, which, like the filter function is not accepted by Coq's builtin coinduction mechanisms. There is also a construction for a CPO of *colists*, which can be seen as the union of finite lists and streams. Accordingly, colists have a constructor `nil` for the empty colist, in addition to `bot` and `scons` like in the above definition of streams. The filter function on colists, defined as the unique fixpoint of a certain functional, turns out to be quite different from the filter function of streams: it is total on the subtype of "well-formed" colists (those that do not contain `bot`) and uses a non-executable "oracle" to determine whether its current argument is such that none of its elements satisfy the filtering predicate. If this is the case, the function returns `nil`, otherwise, it behaves like the filter function for streams.

5.3 Second method

Unlike the first method, in which each individual coinductive type has to be organized as a CPO, the second method provides a generic construction of CPOs, if some assumptions are met. Particular CPOs can be defined as instances of the generic notions, by providing definitions and lemmas that instantiate the assumptions. Corecursive functions between CPOs can then be defined.

5.3.1 Generic CPO

The generic construction of CPOs requires a set `Cc` (C°, in Section 4.1), a least element, and an order relation `ordc` (for $\preceq^\circ$), which must be weakly total. There is also a measure `mu` (for μ) compatible with the strict order. These requirements are gathered in a Coq *module type*:

```
Parameter Cc: Type.
Parameter bot: Cc.
Parameter ordc: Cc-> Cc-> Prop.
Parameter bot_is_least: forall x,ordc bot x.
Parameter ordc_refl: forall x,ordc x x.
Parameter ordc_trans: forall x y z,ordc x y->ordc y z->ordc x z.
Parameter ordc_antisym: forall x y,ordc x y->ordc  y x->x=y.
Parameter ordc_wtot: forall x y z,ordc x z->ordc y z->ordc x y\/ordc y x.
Parameter mu: Cc -> nat.
Parameter mu_sordc: forall x y,ordc x y-> x<>y-> mu x<mu y.
```

The type `Cc` is "extended" to a type `C` by adding equivalence classes of ascending sequences of elements in `Cc`, and the order `ordc` is extended to a relation `ord`, which is then proved to be an order:

```
Inductive C:Type :=
|elt:forall (e:Cc),C
|cls:forall (ec:EqClass),C.
Inductive ord : C-> C-> Prop :=
|elt_elt: forall  e1 e2,ordc e1 e2 -> ord (elt e1)(elt e2)
|elt_cls: forall e ec,
    (forall t, in t ec-> exists n, ordc e (nth t n) )->
    ord (elt e)(cls ec)
|cls_cls: forall ec,ord (cls ec)(cls ec).
```

More information about our encoding of equivalence classes is given at the end of this section. The type `C` is obtained by "wrapping" elements in `Cc` into a constructor `elt` and equivalence classes into a constructor `cls`. The relation `ord` has three cases, corresponding the three cases by which $\preceq^{\circ}$ is extended to $\preceq$ in Definition 22. Then, the limit of an increasing sequence of elements in `C` is defined:

```
Definition lim (s:Seq(T:=C))(H:increasing ord s):C :=
  match (excluded_middle_informative (stabilizing s)) with
  | left stab =>
    let (c, _) := constructive_indefinite_description _ stab in c
    |right nostab =>
     (cls (class_of (exist _ (fun n => extract_elt (s n))
                  (conj (extract_elt_incr _ Hinc nostab)
                          (incr_nostab_nostab _ Hinc nostab)))))end.
```

Like in Definition 25, the code for `lim` needs to decide whether its argument `s` is stabilizing or not. This is not decidable, because a decision procedure would have to examine a whole infinite sequence. We make it decidable by proving a theorem called `excluded_middle_informative` stating that every proposition is decidable: `forall P,{P}+{~P}` – a consequence of classical logic and `constructive_indefinite_description`. Applying that theorem to `(stabilizing s)` leads to two cases: if the sequence is stabilizing (with `stab` being a proof of stabilization) then the value to which it stabilizes is "fetched" by `constructive_indefinite_description`. If the sequence is not stabilizing (with `nostab` being a proof of non-stabilization) then, intuitively the equivalence class of `s` should be returned – except for the fact that `s` is a sequence over `C` and we only have equivalence classes of ascending sequences over `Cc`. Various wrappers, conversion operations, and proof terms are used to produce the adequate equivalence class. Of course, none of these details were visible in the mathematical definition of the limit (Definition 25), but in Coq all the details are exposed. The proof of the fact that `lim` actually computes the least upper bound of its argument amounts to a similar exposure and management of many details, none of which is visible in the mathematical statements – Theorem 26 and its proof.

On equivalence classes

There is no universally accepted way for expressing equivalence classes modulo a given equivalence relation in Coq. One option, supported by the tool's standard library, is to use *setoids*, which are a triple consisting of a type, a binary relation on the type, and a proof that the relation is an equivalence. This approach is mainly used to obtain a generalized rewriting, using the setoid's equivalence relation (which moreover needs to be proved to be a congruence for the contexts under which rewriting is desired) instead of equality. For example, rewriting using bisimulation of streams falls in this category. However, in the present context, we just need equivalence classes for their own sake. Rewriting is not an issue, and using the powerful but complicated machinery of setoids did not seem cost-effective. We therefore opted for a more direct approach that uses axioms from the standard library: `constructive_indefinite_description` for obtaining a representative of a class; functional extensionality (two functions are equal iff they are pointwise equal) and propositional extensionality (propositional equality coincides with equivalence) for proving that if two elements are in the equivalence relation they are in the same equivalence class. In standard mathematics these properties are implicitly assumed, but in Coq they have to be explicitly assumed since they are not provable otherwise.

5.3.2 The CPO of finite and Rose trees

In order to obtain this CPO the parameters of the generic CPO (the type `Cc`, the relation `ordc`, the function `mu`, and the various constraints relating them) have to be instantiated with actual definitions and lemmas. This essentially amounts to encoding the content of Example 29 in Coq. The hardest part was establishing that the relation `ordc` is transitive; several nontrivial lemmas about cutting trees at given heights had to be proved. Perhaps the most difficult part of all the development effort was to convince ourselves that weak totality of the order is a crucial requirement, and therefore to abandon the apparently natural "prefix order", also defined in Example 29, which does not have this property.

5.3.3 The mirror function

Defining a function using the second method is composed of a generic part, which assumes two generic CPOs and a function between their "finite parts" that has to satisfy a productiveness constraint (Definition 30). Accordingly, in Coq we write a module type where such a function and its productiveness requirement are assumed. Any module that implements that module type gains access to the corecursive function definition method described at the end of Section 4.2. A recursive `fmirror` function between finite trees is written in such a module, and by the generic mechanism described above, this function is transformed into a corecursive function `mirror` between Rose trees.

Finally, to gain confidence in the obtained definition we define functions `fpos` and `fpos_rev` (cf. Example 33) that compute labels at given positions in finite trees; transform these functions into `pos` and `pos_rev` that do the corresponding operations on Rose trees; and prove the following lemma:

```
Lemma mirror_pos: forall p t, pos p (mirror t) = pos_rev p t.
```

6 Conclusion, related work, and future work

This paper presents two methods for defining corecursive functions that go beyond the guarded-by-constructor setting available in the Coq proof assistant. The first method reuses the dedicated coinduction mechanisms available in Coq, which works as long as the underlying coinductive datatypes are not mutually dependent with inductive types. The second method is not subject to this restriction, as it does not rely on Coq's coinduction mechanisms but redefines them, at the cost of some additional work. Both methods have in common the interpretation of maximal values in CPOs as well-defined corecursive values, and they both rely on a mathematical notion of *productiveness* that captures the corresponding intuitive notion of productiveness (the ability of a function to eventually generate, for each input, an arbitrarily close approximation of the corresponding output). Both methods are implemented in Coq and are illustrated by defining corecursive functions that Coq's dedicated mechanisms reject. This gain in expressiveness is obtained at the cost of using axioms from the standard library of Coq, which are known not to introduce inconsistencies: using them amounts to losing constructiveness, but gaining access to standard mathematical reasoning. Both methods were presented independently of Coq; especially the second one, which *is* independent from Coq's builtin mechanisms for corecursion, could be implemented in other proof assistants. An interesting target is Lean [14], a dependently-typed language and proof assistant that includes the additional feature of *quotient types* that would naturally encode equivalence classes in the second method.

The methods we propose transform a problem currently without solution (defining corecursive functions that do not satisfy the guardedness condition) into a problem that is solvable: defining and reasoning about functions that approximate the function under definition. In practice the approximating functions are recursive, as can be seen from the examples in the paper (Examples 14 and 31) and from the additional ones in the companion Coq development. Now, if for a given corecursive function the corresponding approximating recursive functions are difficult to reason about, then applying our methods may be difficult. However, most of the difficulty does not arise from the methods, but from the intrinsic complexity of the corecursive function being defined.

Comparison with related work

We start with classical results and with their applications for the purpose of defining functions. Kleene's fixpoint theorem [19, Chapter 5] can be used to define functions as *least* fixpoints of *continuous* functionals over CPOs. A functional is continuous if it commutes with least upper bounds. The least fixpoint is the least upper bound of an increasing sequence of functions, obtained by iterating the functional starting from the constant "bottom" function. This has been formalized and used for defining recursive functions in Coq [5]. Unsurprisingly, they use the same kinds of axioms as we do.

In our first method we use the same iteration as in Kleene's fixpoint theorem to obtain a fixpoint, but require *productiveness* instead of continuity; and we obtain a unique fixpoint, not just a least fixpoint. The stronger fixpoint result, and the fact that productiveness is a natural requirement for corecursive functions, suggest that our method is well-adapted for the purpose of defining such functions.

Our second method has similarities with the classical construction of the real numbers based on equivalence classes of Cauchy sequences of rational numbers [10, Appendix A]. However, Cauchy sequences over a base set require the base set to be organized as a metric space, with a distance function satisfying certain properties. An approach for defining corecursive functions based on Cauchy sequences is mentioned in [13]. They use another classical result (Banach's fixpoint theorem [2]) to define corecursive functions as unique fixpoints of *eventually contracting* functionals. By contrast, we organize the base set as a CPO, use ascending sequences instead of Cauchy sequences, and (in the second method) do not use functionals, but a "finite version" of the corecursive function under definition, which has to satisfy a certain productiveness requirement to ensure a proper definition.

We now present related work about corecursion in proof assistants and similar formalisms. In Coq, corecursive function definitions have to satisfy a guardedness-by-constructors criterion. This criterion ensures a strong version of productiveness, namely, that each evaluation step produces a strictly closer approximation of the final result than the previous steps. By contrast, productiveness only requires that *eventually* a strictly closer approximation is obtained. In some cases, a function that is productive but unguarded can be transformed into an equivalent, guarded function. This has been done for the filter function on streams in [3] and generalized in [4] to other unguarded functions. Their idea is to use an ad-hoc predicate stating that the definition under study is, in some sense, productive. However, their approach does not use a general, formal notion of productiveness, nor does it handle the case where corecursive calls are guarded by some non-constructor function, like the mirror function for Rose trees presented in this paper. Our approach is not subject to these limitations. In other related work, a constructive version of the CPO of streams in Coq is mentioned in [16] in the context of a coinductive formalization of Kahn networks. However, the author does not use her formalization of CPO to extend the class of corecursive stream functions admissible by Coq.

We note that *coinductive proofs* in Coq, which by default are subject to the same syntactical requirements as corecursive functions, can be performed using more relaxed, semantical requirements by using *parameterized coinduction* implemented in the *Paco* extension of Coq [12]. We have tried to adapt parameterized coinduction to corecursive function definition, but have given up because we found that it is not adaptable. Parameterized coinduction works for coinductive proofs, because, there, *witness terms* do not matter – any term of the right type will do. By contrast, in corecursive functions, witness terms do matter, since they expresses what the function is supposed to compute.

Agda [20] is also a dependently-typed programming language and proof assistant that offers support for corecursive function definition. In the core tool there is a guardedness checker similar to that of Coq, but more liberal as it allows, e.g., the definition of two mutually dependent functions, one of which is recursive and the other one, corecursive. This enables it to accept the definition of the mirror function on Rose trees, which Coq does not accept. Extensions to Agda with sized types [18] provide users with a uniform, automatic way of handling termination and productiveness, based on type annotations written by the user. The current implementation of sized types in Agda is unsound (cf. [20, chapter Safe Agda] and `https://github.com/agda/agda/issues/2820`).

Isabelle/HOL [21] is another major proof assistant which supports corecursive function definition. A guardedness criterion (there called *primitive corecursion*) similar to that of Coq and Agda is implemented [6], based on *bounded natural functors*, a conservative extension of Higher Order Logic. The framework has further been extended to accept function definitions that go beyond primitive corecursion [7]. Isabelle/HOL now accepts function definitions where corecursive calls can be guarded by functions other than constructors, provided the functions are proved to be *friendly* (essentially, a friendly function needs to destruct at most one constructor of input to produce one constructor of output). Unguarded corecursive calls, such as those in the filter function on streams, are also accepted, provided they are proved to eventually produce a constructor of output. Like in our approach, all proof obligations are the responsibility of the user. They have the additional advantage of using no supplementary axioms, as those of Higher Order Logic are expressive enough.

Beyond generic proof assistants, support for corecursion also exists in tools targeting particular languages. For example, Dafny is a specification and verification language dedicated to the C# language, which has support for corecursive function definition [15], based on a guardedness criterion similar to those existing in the already mentioned tools. Coinductive proofs are also supported.

Finally, beyond the area of formal verification, it is very worth mentioning the Haskell functional language, which offers support for corecursive function definition by means of lazy evaluation.

Future work

We have encountered corecursive functions that are productive yet do not obey the guarded-by-constructors criterion in our planned future work. The Prelude dataflow synchronous programming language [9] has a flow sampling construction whose semantics is best described using a filter function on colists (which we have defined in the companion Coq development as an instance of our first method). This opens the way to a mechanized semantics of Prelude in Coq, which would then enable program verification and semantically correct code generation for the language. While formalizing in Coq the paper [17] about the semantics of dataflow languages we have encountered unguarded corecursive functions on streams that can also be defined using our first method. More speculative future work includes a comparison and possible cross-fertilization of our approach with the sized-type approach of Agda and the bounded-natural-functor approach of Isabelle/HOL.

References

1 *The Coq Proof Assistant.* URL: https://coq.inria.fr/.

2 S. Banach. Sur les opérations dans les ensembles abstraits et leur applications aux équations intégrales. *Fundam. Math.*, 3:133–181, 1922.

3 Y. Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In *Typed Lambda Calculi and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*, volume 3461 of *Lecture Notes in Computer Science*, pages 102–115, 2005.

4 Y. Bertot and E. Komendantskaya. Inductive and coinductive components of corecursive functions in Coq. In *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science, CMCS 2008, Budapest, Hungary, April 4-6, 2008*, volume 203 of *Electronic Notes in Theoretical Computer Science*, pages 25–47, 2008.

5 Y. Bertot and V. Komendantsky. 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*, pages 89–96, 2008.

6 J. Biendarra, J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. *Defining (Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL.* URL: https://isabelle.in.tum.de/doc/datatypes.pdf.

7 J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. *Defining Nonprimitively (Co)recursive Functions in Isabelle/HOL.* URL: https://isabelle.in.tum.de/dist/Isabelle2021/doc/corec.pdf.

8 A. Chlipala. *Certified Programming with Dependent Types.* MIT Press, 2013.

9 J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints.* PhD thesis, Institut Supérieur de l'Aéronautique et de l'Espace, Toulouse, France, 2009.

10 S. R. Ghorpade and B. V. Limaye. *A Course in Calculus and Real Analysis.* Undergraduate Texts in Mathematics. Springer, 2018.

11 E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.

12 C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 193–206. ACM, 2013.

13 D. Kozen and N. Ruozzi. Applications of metric coinduction. *Log. Methods Comput. Sci.*, 5(3), 2009.

14 *The Lean Proof Assistant.* URL: https://leanprover.github.io/.

15 K. Rustan M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program verifier. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 382–398. Springer, 2014.

16 C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. Available at `https://www.lri.fr/~paulin/PUBLIS/paulin07kahn.pdf`.

17 T. Uustalu and V. Vene. The essence of dataflow programming. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.

18 N. Veltri and N. van der Weide. Guarded recursion in agda via sized types. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

19 G. Winskel. *The Formal Semantics of Programming Languages, an introduction.* MIT Press, 1993.

20 *The Agda Proof Assistant.* `https://agda.readthedocs.io/en/v2.6.2/getting-started/what-is-agda.html`.

21 *The Isabelle/HOL Proof Assistant.* URL: `https://isabelle.in.tum.de/`.