



**HAL**  
open science

# Defining Corecursive Functions in Coq Using Approximations

Vlad Rusu, David Nowak

► **To cite this version:**

Vlad Rusu, David Nowak. Defining Corecursive Functions in Coq Using Approximations. ECOOP, Jun 2022, Berlin, Germany. 10.4230/LIPIcs.ECOOP.2022.12 . hal-03671876

**HAL Id: hal-03671876**

**<https://inria.hal.science/hal-03671876>**

Submitted on 18 May 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# 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

## 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.

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

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



© Vlad Rusu and David Nowak;

licensed under Creative Commons License CC-BY 4.0

36th European Conference on Object-Oriented Programming (ECOOP 2022).

Editors: Karim Ali and Jan Vitek; Article No. 12; pp. 12:1–12:22



Leibniz International Proceedings in Informatics

LIPIcs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

## 12:2 Defining Corecursive Functions in Coq Using Approximations

45 have automatic mechanisms to realize this. Its builtin syntactical criteria are automatic, sound (i.e. all  
46 functions that fulfill them are total), but restricted since they reject some total functions.

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

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

### 70 Contributions.

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

82 The rest of the paper is organized as follows. A theoretical part (Sections 2-4) presents a formal  
83 notion of productiveness and our two corecursive function-definition methods in a language-agnostic  
84 manner. We emphasize that knowing Coq is not necessary for understanding the theory. Section 5  
85 gives details of the Coq implementation that are not visible in the theory but are essential in the  
86 implementation, such as the combination of axioms imported from the standard library that enable  
87 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  $\leq$  on  $C$ . We denote by  $<$  the relation defined by  $t < t'$  iff  $t \leq t'$  and  $t \neq t'$ .

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

- increasing whenever for all  $i \in \mathbb{N}$ ,  $s_i \leq s_{i+1}$ ;
- strictly increasing, whenever for all  $i \in \mathbb{N}$ ,  $s_i < s_{i+1}$ ;
- 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  $\leq$  on  $C$ , and an element  $\perp \in C$  satisfying  $\forall t \in T, \perp \leq t$ , 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  $(A \cup \{\perp_A\}, \leq_A, \perp_A)$  by choosing some value  $\perp_A \notin A$  and by defining  $\leq_A$  as the smallest relation on  $A \cup \{\perp_A\}$  satisfying  $\perp_A \leq_A a$  for all  $a \in A$  and  $a' \leq_A a'$  for all  $a' \in A \cup \{\perp_A\}$ . The properties of orders (reflexivity, anti-symmetry, transitivity) hold trivially. Any increasing sequence  $(a_n)_{n \in \mathbb{N}}$  stabilizes to some  $a \in A \cup \{\perp_A\}$ , 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  $(C, \leq, \perp)$ , 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  $\leq$ . The limit of the sequence  $(f_n)_{n \in \mathbb{N}}$  is by definition the function  $f : D \rightarrow C$  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  $i \in \mathbb{N}$ ,  $f_{n_i} x < f_{n_{i+1}} x$ , 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.

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

### 133 **3 First method**

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

#### 138 **3.1 CPOs as coinductive types**

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

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

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

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

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

173 ▶ **Remark.** The construction in Example 5 is not the only way to organize streams over a set  $A$  into a  
 174 CPO. Another possibility is to define the set of streams  $S$  as the set obtained by applying the rules  
 175  $\perp \in S$  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  
 176 definition  $\perp$  is a constructor (unlike  $\perp$  in Example 5 where it was a defined function). The order  
 177 relation and the notion of limit are also slightly different. We chose the construction in Example 5  
 178 because it has fewer technical complications: for example, the *head* function is total in Example 5,  
 179 but partial in the alternative construction, which makes it more complicated to define.

### 180 3.2 Approximating sequences using functionals

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

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

189 
$$\text{filter } s := \text{if } p(\text{head } s) \text{ then } (\text{head } s) \cdot (\text{filter } (\text{tail } s)) \text{ else } \text{filter } (\text{tail } s)$$

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

192 
$$Ff := \lambda s. \text{if } p(\text{head } s) \text{ then } (\text{head } s) \cdot (f(\text{tail } s)) \text{ else } f(\text{tail } s)$$

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

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

198 ▶ **Definition 7.** A functional  $F : (D \rightarrow C) \rightarrow D \rightarrow C$  is increasing if for all  $f_1, f_2 : D \rightarrow C$ ,  $f_1 \leq f_2$   
 199 implies  $Ff_1 \leq Ff_2$ .

200 ▶ **Example 8.** The functional  $F$  from the previous example is increasing. Indeed, consider two  
 201 functions  $f_1, f_2 : D \rightarrow S$ , with  $D, S$  as in the example in question (in particular,  $S$  is organized as  
 202 a CPO as in Example 5) and assume  $f_1 \leq f_2$ . We have to show that  $Ff_1 s \leq Ff_2 s$  for all  $s \in D$ .  
 203 If  $p(\text{head } s) = true$  then  $Ff_1 s = (\text{head } s) \cdot f_1(\text{tail } s)$  and  $Ff_2 s = (\text{head } s) \cdot f_2(\text{tail } s)$ ; and  $Ff_1 s \leq$   
 204  $Ff_2 s$  because  $f_1 \leq f_2$  implies in particular  $f_1(\text{tail } s) \leq f_2(\text{tail } s)$ , and then  $(\text{head } s) \cdot f_1(\text{tail } s) \leq$   
 205  $(\text{head } s) \cdot f_2(\text{tail } s)$  holds thanks to the definition of  $\leq$ . If  $p(\text{head } s) = false$  then  $Ff_1 s = f_1(\text{tail } s)$ ,  
 206  $Ff_2 s = f_2(\text{tail } s)$ , and  $Ff_1 s \leq Ff_2 s$  is just  $f_1(\text{tail } s) \leq f_2(\text{tail } s)$ , established above.

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

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

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

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

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

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

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

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

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

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

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

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

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

244 **Proof.** By Definitions 4 and 9, we have to show that for all  $x \in D$ , the sequence  $(F^n \perp x)_{n \in \mathbb{N}}$  has a  
 245 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  
 246 base case  $n = 0$ , we have, for each  $x \in D$ ,  $F^0 \perp x = \perp x = \perp \leq F^1 \perp x$ , which settles this case. For the  
 247 inductive step, we assume that for each  $x \in D$ ,  $F^n \perp x \leq F^{n+1} \perp x$  and prove that, again for each  $x \in D$ ,  
 248  $F^{n+1} \perp x \leq F^{n+2} \perp x$ . We have  $F^{n+1} \perp x = F(F^n \perp) x$  and since  $F$  is increasing, using the induction  
 249 hypothesis  $F(F^n \perp) x \leq F(F^{n+1} \perp) x = F^{n+2} \perp x$  holds for each  $x \in D$ , which proves the induction  
 250 step and the fact that the sequence is increasing.

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

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

259 ▶ **Example 14.** We prove using Lemma 13 that the functional  $F : (D \rightarrow S) \rightarrow D \rightarrow S$  for the filter  
 260 function from Example 6 is productive. We have already established that it is increasing and that the  
 261 CPO  $(S, \leq, \perp)$  is a CPO+. We prove the condition at the second item the statement of Lemma 13, i.e.,  
 262 for all  $x \in D$  and  $n \in \mathbb{N}$ , there exists  $m \in \mathbb{N}$  with  $n < m$  such that  $F^n \perp x < F^m \perp x$ , which amounts  
 263 to finding a strictly increasing sequence of natural numbers  $(n_i)_{i \in \mathbb{N}}$  such that  $F^{n_i} \perp x < F^{n_{i+1}} \perp x$  for  
 264 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  
 265  $p : (A \cup \{\perp_A\}) \rightarrow \{true, false\}$  on the base type of  $S$  with  $p \perp_A = false$ , it holds that  $p(\text{nth } n \ x) = true$   
 266 for infinitely many  $n \in \mathbb{N}$ . We first prove by induction on  $n$  that  $F^n \perp = \text{ffilter } n$  where  $\text{ffilter} =$   
 267  $\lambda n. \lambda x. (\text{if } n = 0 \text{ then } \perp \text{ else } (\text{if } p(\text{head } x) \text{ then } (\text{head } x) \cdot (\text{ffilter } (n-1) (\text{tail } x)) \text{ else } \text{ffilter } (n-1) (\text{tail } x)))$   
 268 is a recursive, “finite approximation” of the corecursive *filter* function that we are trying to define.  
 269 Then, we notice that if  $p(\text{head } x) = true$  then  $\text{ffilter } (n+1) \ x = (\text{head } x) \cdot (\text{ffilter } n (\text{tail } x))$ , and if  
 270  $p(\text{head } x) = false$  then  $\text{ffilter } (n+1) \ x = \text{ffilter } n (\text{tail } x)$ . That is, for the positions in the sequence  $x$   
 271 where  $p$  holds, the output of *ffilter* “grows”, and for the positions where  $p$  does not hold, the output  
 272 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  
 273 the strictly increasing sequence  $(n_i)_{i \in \mathbb{N}}$  such that  $F^{n_i} \perp x = \text{ffilter } n_i \ x < \text{ffilter } n_{i+1} \ x = F^{n_{i+1}} \perp x$  for all  
 274  $i \in \mathbb{N}$ . Hence, using Lemma 13 we have established that the functional  $F = \lambda f. \lambda s. \text{if } p(\text{head } s) \text{ then}$   
 275  $(\text{head } s) \cdot (f (\text{tail } s)) \text{ else } f (\text{tail } s)$  is productive. Using Theorem 10 we obtain that  $F$  has a unique  
 276 fixpoint; we call *filter* the fixpoint in question. The fixpoint equation states that *filter*  $s = \text{if } p(\text{head } s) \text{ then}$   
 277  $(\text{head } s) \cdot (\text{filter } (\text{tail } s)) \text{ else } \text{filter } (\text{tail } s)$  for all  $x \in D$ . We note that  $D$  being the set of streams  
 278 having infinitely many positions satisfying the filtering predicate is essential: outside this domain the  
 279 functional  $F$  is not productive and one cannot use Theorem 10 as above to define the filter function.

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

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

## 294 4 Second method

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

### 297 4.1 CPOs built by completion

298 The main idea is to start from the “finite subset” of the intended CPO and from an order relation on  
 299 the given subset, and to “complete” them with values that are the equivalence classes of ascending  
 300 sequences, according to a certain equivalence relation. We illustrate the notions introduced in this  
 301 section by giving an alternative construction of a CPO of streams, different from the construction



## 12:8 Defining Corecursive Functions in Coq Using Approximations

302 based on Coq’s builtin mechanisms seen in the previous section. We also show an example where the  
 303 present construction of a CPO is essential because Coq’s builtin mechanisms for coinduction fail.

304 ► **Definition 15.** Given a set  $C^\circ$  and an order  $\leq^\circ$  on it, a measure on  $(C^\circ, \leq^\circ)$  is a function  $\mu : C^\circ \rightarrow \mathbb{N}$   
 305 such that for all  $x, y \in C^\circ$ ,  $x <^\circ y$  implies  $\mu x < \mu y$ .

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

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

311 ► **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  
 312 that tends to infinity. This is the main reason why we chose natural numbers as measure values.

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

315 A common predecessor of two elements is, in our approach, an “under-approximation” of the two  
 316 elements. Thus, two sequences are in the  $\sim$  relation whenever there is a sequence of pointwise  
 317 “under-approximations” of two sequences, whose measures tend to infinity. In some sense, the  
 318 pointwise “difference” between the sequences intuitively tends to “nothing”<sup>1</sup>. In order to show that  $\sim$   
 319 restricted to ascending sequences is an equivalence, the following property of an order is required:

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

322 ► **Example 19.** The prefix order  $\leq^L$  on lists over a set  $A$  is weakly total: when  $l_1$  and  $l_2$  are both  
 323 prefixes of a given list  $l$  then, if  $length\ l_1 \leq length\ l_2$ ,  $l_1 \leq^L l_2$  holds, otherwise,  $l_2 \leq^L l_1$  holds. If  $A$   
 324 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.

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

327 **Proof.** For reflexivity, we use the fact that the sequence  $(\mu s_n)_{n \in \mathbb{N}}$  of measures of an ascending  
 328 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  
 329 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  
 330 Definition 17 is a symmetrical statement in  $(s_n)_{n \in \mathbb{N}}$ ,  $(s'_n)_{n \in \mathbb{N}}$ . For transitivity assume  $(s_n)_{n \in \mathbb{N}} \sim (s'_n)_{n \in \mathbb{N}}$   
 331 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  $y, y' \in C^\circ$   
 332 such that  $y \leq^\circ s_m$ ,  $y \leq^\circ s'_m$ ,  $y' \leq^\circ s'_m$ ,  $y' \leq^\circ s''_m$ , and  $\mu y, \mu y' \geq N$ . Since the sequences are increasing,  
 333 we have  $y, y' \leq^\circ s'_{(\max m m')}$  and since  $\leq^\circ$  is weakly total,  $y \leq^\circ y'$  or  $y' \leq^\circ y$ . Assume  $y \leq^\circ y'$ . Then, for  
 334 the arbitrarily chosen  $N$ , we set  $n := (\max m m')$  and  $x := y$  in Definition 17 and, since the sequences  
 335 are increasing, we obtain  $(s_n)_{n \in \mathbb{N}} \sim (s''_n)_{n \in \mathbb{N}}$ . The other case ( $y' \leq^\circ y$ ) is similar. ◀

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

337 ► **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  $m \in \mathbb{N}$   
 338 such that  $s_k \leq s'_m$ , then  $(s_n)_{n \in \mathbb{N}} \sim (s'_n)_{n \in \mathbb{N}}$ .

<sup>1</sup> This intuition can be formalized using a notion of distance, thus turning  $C^\circ$  into a metric space. We have tried but discarded that approach because it 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.

339 **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  $\mu s_k \geq N$ .  
 340 From the hypothesis we obtain  $m \in \mathbb{N}$  such that  $s_k \leq^\circ s'_m$ . Let  $n := (\max k m)$  and  $x := s_k$ . Since  
 341 the sequences are increasing,  $x \leq s_n$ ,  $x \leq^\circ s'_n$ , and from  $\mu s_k \geq N$  we obtain  $\mu x \geq N$ . Hence, for all  
 342  $N \in \mathbb{N}$  there are  $n \in \mathbb{N}$ ,  $x \in C^\circ$  such that  $x \leq^\circ s_n$ ,  $x \leq^\circ s'_n$ ,  $\mu x \geq N$ . By Def. 17,  $(s_n)_{n \in \mathbb{N}} \sim (s'_n)_{n \in \mathbb{N}}$ . ◀

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

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

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

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

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

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

- 382 1.  $x, y, z \in C^\circ$ , in which case the transitivity of  $\leq$  reduces to that of  $\leq^\circ$ ;

## 12:10 Defining Corecursive Functions in Coq Using Approximations

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

389

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

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

393 **Proof.** By Definition 22,  $x \leq [(s_n)_{n \in \mathbb{N}}]_{\sim}$  means: for all  $(s'_n)_{n \in \mathbb{N}} \in [(s_n)_{n \in \mathbb{N}}]_{\sim}$ , there exists  $m \in \mathbb{N}$   
394 such that  $x \leq^\circ s'_m$ . The “only if” direction is trivial since obviously  $(s_n)_{n \in \mathbb{N}} \in [(s_n)_{n \in \mathbb{N}}]_{\sim}$ . We thus  
395 focus on the “if” direction. By hypothesis, there exists  $m \in \mathbb{N}$  such that  $x \leq^\circ s_m$ . Choose an  
396 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  
397  $m' \in \mathbb{N}$  such that  $x' \leq^\circ s_{m'}$ ,  $x' \leq^\circ s'_{m'}$  and  $\mu x' > \mu x$ . Since the sequences are increasing, we obtain  
398  $x, x' \leq^\circ s_{(\max m, m')}$ . From the latter and the weak totality of  $\leq^\circ$  we obtain  $x \leq^\circ x'$  or  $x' \leq^\circ x$ . But  
399  $x' \leq^\circ x$  contradicts the established  $\mu x' > \mu x$ . Hence,  $x \leq^\circ x'$  and then  $x \leq^\circ s'_{m'}$  follows from  
400  $x' \leq^\circ s'_{m'}$  by transitivity. Summarizing, for the arbitrarily chosen sequence  $(s'_n)_{n \in \mathbb{N}} \in [(s_n)_{n \in \mathbb{N}}]_{\sim}$  we  
401 found  $m' \in \mathbb{N}$  such that  $x \leq^\circ s'_{m'}$ . But this is  $x \leq [(s_n)_{n \in \mathbb{N}}]_{\sim}$  by definition; which proves the lemma. ◀

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

- 405 ■ *if the sequence stabilizes at a value, say,  $v \in C$ , then  $\lim[(s_n)_{n \in \mathbb{N}}] = v$ ;*
- 406 ■ *otherwise, the sequence does not stabilize, which implies that for all  $n \in \mathbb{N}$ ,  $s_n \in C^\circ$ , and we*  
407 *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  $\sim$ .*

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

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

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

- 415 ■ *if the sequence stabilizes to some value  $v \in C$  then the proposed limit  $v$  is an upper bound for the*  
416 *(increasing) sequence. To show that it is the least such bound, assume another upper bound  $w$ ;*  
417 *then, in particular,  $v \leq w$  because  $v$  is an element of the sequence.*
- 418 ■ *if the sequence does not stabilize then it is ascending, and as already observed before,  $s_n \in C^\circ$  for*  
419 *all  $n \in \mathbb{N}$ , and the proposed limit is the equivalence class  $[(s_n)_{n \in \mathbb{N}}]_{\sim}$ .*
  - 420 ■ *We first show that  $[(s_n)_{n \in \mathbb{N}}]_{\sim}$  is an upper bound for  $(s_n)_{n \in \mathbb{N}}$ :  $s_k \leq [(s_n)_{n \in \mathbb{N}}]_{\sim}$  for all  $k \in \mathbb{N}$ . We*  
421 *apply Lemma 24 with  $x := s_k$ : there exists  $m := k$  such  $s_k \leq s_m$ , which implies  $s_k \leq [(s_n)_{n \in \mathbb{N}}]_{\sim}$ .*
  - 422 ■ *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*  
423  *$w \in C$ , thus,  $s_k \leq w$  for all  $k \in \mathbb{N}$ . Suppose first that  $w \in C^\circ$ . Since  $(s_n)_{n \in \mathbb{N}}$  is ascending, it has*  
424 *a strictly increasing subsequence  $(s_{n_i})_{i \in \mathbb{N}}$ . Now,  $w$  is also an upper bound for the subsequence,*

425 hence,  $s_{n_i} \leq w$  for all  $i \in \mathbb{N}$ , and due to the properties of the measure,  $\mu s_{n_i} \leq \mu w$  for all  $i \in \mathbb{N}$ .  
 426 But this is impossible, since the sequence of measures of a strictly increasing sequence is a  
 427 strictly increasing sequence of natural numbers, which tends to infinity. Hence,  $w \in C^\circ$  is  
 428 impossible. It follows that  $w \in K$ , i.e.,  $w = [(s'_n)_{n \in \mathbb{N}}]_{\sim}$  for some ascending sequence  $(s'_n)_{n \in \mathbb{N}}$ .  
 429 From our hypothesis  $s_k \leq w$  for all  $k \in \mathbb{N}$ , we obtain that for all  $k \in \mathbb{N}$  there exists  $m \in \mathbb{N}$  such  
 430 that  $s_k \leq^\circ s'_m$ . 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  
 431 particular  $[(s_n)_{n \in \mathbb{N}}]_{\sim} \leq w$ . Since the upper bound  $w$  was chosen arbitrarily, we have proved that  
 432  $[(s_n)_{n \in \mathbb{N}}]_{\sim}$  is the least upper bound for  $(s_n)_{n \in \mathbb{N}}$ . The proof of the theorem is complete.

433

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

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

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

452 ► **Example 29.** One can define and organize the set  $T = F \cup R$ , with  $F$  the set of finite trees and  $R$   
 453 that of Rose trees, in a CPO using the approach described in this section. We first define the finite  
 454 trees  $F$  over  $A$  inductively, by the rules  $\perp \in F$  and  $tree\ a\ l \in F$  whenever  $a \in A$  and  $l$  is a list over  $F$ .  
 455 The measure function is the tree's height, recursively defined by  $height\ \perp = 0$  and  $height\ (tree\ a\ l) =$   
 456  $1 + max\ (map\ height\ l)$  where *max* computes the maximum value in a list of natural numbers. The  
 457 order relation is based on the following recursive function, whose effect is to “cut” a given finite tree  $t$   
 458 at given depth  $n$ :  $cut = \lambda n.\lambda t.\text{if } n = 0 \text{ or } t = \perp \text{ then } \perp \text{ else } tree\ (label\ t)\ (map\ (cut\ (n - 1))\ (forest\ t))$ ;  
 459 we then define the order relation  $\leq^F$  by  $t_1 \leq^F t_2$  whenever  $t_1 = cut\ (height\ t_1)\ t_2$ . The set  $R$  of Rose  
 460 trees consists of equivalence classes of ascending sequences of finite trees. We have proved in Coq  
 461 that all the requirements presented earlier in this section for obtaining a CPO for  $T = F \cup R$  hold.

462 By contrast, a perhaps more natural definition of the “prefix order”  $\leq'^F$  by  $t_1 \leq'^F t_2$  whenever  
 463  $t_1 = \perp$  or  $t_1 = tree\ a\ l_1$ ,  $t_2 = tree\ a\ l_2$ ,  $length\ l_1 = length\ l_2$  and for all  $n < length\ l_1$ ,  $nth\ n\ l_1 \leq'^F nth\ n\ l_2$   
 464 fails to meet the critical weak totality requirement (Definition 18). Indeed, e.g., for  $t', t'' \neq \perp$ ,  
 465  $t = tree\ a\ [t', t'']$ ,  $t_1 = tree\ a\ [t', \perp]$  and  $t_2 = tree\ a\ [\perp, t'']$  satisfy  $t_1 \leq'^F t$  and  $t_2 \leq'^F t$ , yet  $t_1$  and  $t_2$  are  
 466 incomparable. Without weak totality there is no sequence equivalence and ultimately no CPO<sup>2</sup>.

<sup>2</sup> Our earlier attempt with metric spaces also required a weakly total order for obtaining a proper notion of distance.

467 **4.2 Approximating sequences without functionals**

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

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

484 **► Definition 30.** *Assume two CPOs  $(D, \leq_D, \perp_D)$  and  $(C \leq_C, \perp_C)$  defined as in Theorem 26, thus,*  
 485 *their base sets are decomposed as  $C = C^\circ \cup K_C$  and  $D = D^\circ \cup K_D$ . Then, a function  $f^\circ : D^\circ \rightarrow C^\circ$  is*  
 486 *productive whenever, for all increasing sequences  $(x_n)_{n \in \mathbb{N}}$  of elements in  $D^\circ$  that have a limit in  $K_D$ ,*  
 487 *the sequence  $(f^\circ x_n)_{n \in \mathbb{N}}$  of elements in  $C^\circ$  is also increasing and has a limit in  $K_C$ .*

488 **► Remark.** Definition 30 of a productive function implies the function is also increasing. It also  
 489 implies that the function maps ascending sequences to ascending sequences. Calling such a function  
 490 *productive* is justified by the fact that it generates a sequence of functions that productively converges  
 491 according to Definition 4. This sequence is built as follows: for all  $x \in K_D$ , choose an arbitrary  
 492 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  
 493 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  $K_C$ .

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

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

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

506 **► Theorem 32.** *In the context of Definition 30, let  $\sim_D$  denote the equivalence relation on ascending*  
 507 *sequences in the CPO  $(D, \leq_D, \perp_D)$  (cf. Definition 17, Lemma 20). Let  $\sim_C$  denote the corresponding*  
 508 *equivalence in  $(C \leq_C, \perp_C)$ . Then, for any pair of equivalent ascending sequences  $(s_n)_{n \in \mathbb{N}} \sim_D (s'_n)_{n \in \mathbb{N}}$*   
 509 *and any productive function  $f^\circ : D^\circ \rightarrow C^\circ$ , we have the equivalence  $(f^\circ s_n)_{n \in \mathbb{N}} \sim_C (f^\circ s'_n)_{n \in \mathbb{N}}$ .*

510 **Proof.** By Definition 22 and Theorem 26 the equivalence class  $[(s_n)_{n \in \mathbb{N}}]_{\sim_D}$  is the least upper bound  
 511 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

512 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  
 513  $[(s'_n)_{n \in \mathbb{N}}]_{\sim_D}$  is an upper bound for  $(s_n)_{n \in \mathbb{N}}$ , thus for all  $n \in \mathbb{N}$ ,  $s_n \leq_D [(s'_n)_{n \in \mathbb{N}}]_{\sim_D}$  and by Lemma 24,  
 514 (i): for all  $n \in \mathbb{N}$ , there exists  $m \in \mathbb{N}$  such that  $s_n \leq_D^{\circ} s'_m$ . Since  $f^{\circ}$  is productive, it is also increasing,  
 515 and thus from (i) we obtain (ii): for all  $n \in \mathbb{N}$ , there exists  $m \in \mathbb{N}$  such that  $f^{\circ} s_n \leq_C^{\circ} f^{\circ} s'_m$ . Using  
 516 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.  $\blacktriangleleft$

517 The above result enables us to define functions  $f : K_D \rightarrow C$  as limits of approximating sequences  
 518  $(f_n : K_D \rightarrow C)_{n \in \mathbb{N}}$ . The definition of each of the functions  $f_n$  below depends on an arbitrary choice for  
 519 a representative in its argument (which is an equivalence class), however, thanks to the above theorem,  
 520 the limit (i.e. the defined function  $f$ ) does not depend on that choice. The functions are built as in the  
 521 Remark following Definition 30: for all  $x \in K_D$ , choose an arbitrary ascending sequence  $(x_n)_{n \in \mathbb{N}} \in x$ ;  
 522 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  
 523 to Definition 4; its limit is  $[(f^{\circ} x_n)_{n \in \mathbb{N}}]_{\sim_C} \in K_C$ , which, by above theorem, is independent of the choice  
 524 for  $(x_n)_{n \in \mathbb{N}} \in x$ . Hence, the limit  $f := \lambda x. \text{lim}[(f_n x)_{n \in \mathbb{N}}]$  is also independent on the initial choice.

525 Of course, the natural question that arises regards validation: do the functions thus defined match  
 526 the intention of the user? Unlike the case of functional-based corecursive functions in an earlier  
 527 section, we do not have a functional and a fixpoint equation as validation mechanisms. A certain  
 528 degree of confidence in the definition of  $f$  is already obtained from the (assumed) confidence in  $f^{\circ}$  —  
 529 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  
 530 from Theorem 32. The confidence can be improved by proving properties of  $f$  that the user expects.

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

## 546 5 Implementation

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

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

557 **5.1 Sequences**

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

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

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

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

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

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

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

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

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

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

594 **5.2 First method**

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

596 **5.2.1 Stream CPO**

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

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

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

```
608 Lemma leo_lub{A:Type} :  
609 forall (q:Seq (A:=option A)),increasing leo q-> exists b,lub leo b q.  
610  
611
```

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

```
613 Definition limF{A:Type}(q:Seq(A:=option A))(H:increasing leo q):=  
614 constructive_indefinite_description _ (leo_lub q H).  
615  
616
```

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

```
619 CoInductive Stream{T:Type} := scon : T -> Stream -> Stream.  
620 CoFixpoint bot{T:Type}: Stream(T:=T) := scon None bot.  
621  
622
```

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

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

```
627 CoInductive les{T:Type} :Stream(T:=T)-> Stream (T:=T)-> Prop :=  
628 les_def:forall a b s s', leo a b-> les s s'-> les(scons a s)(scons b s').  
629  
630
```

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

```
632 CoFixpoint lim{A:Type}(q:Seq(A:=Stream(T:=option A)))(H:increasing les q)  
633 := scon  
634 (proj1_sig(limF(fun n => hd(q n))(increasing_smap_hd q H)))  
635 (lim(fun n => tl(q n))(increasing_smap_tl q H)).  
636  
637
```

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

```
646 Lemma lim_lub{A:Type}(q:Seq(A:=Stream(T:=option A)))(H:increasing les q):  
647 lub les (lim q H) q.  
648  
649
```

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

## 655 5.2.2 The filter function on streams

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



## 12:16 Defining Corecursive Functions in Coq Using Approximations

```
657 Definition Filter(f:S->Stream(T:=option A))(s: S): Stream(T:=option A):=
658   if P (head s) then
659     scon (head s) (f (tail s))
660   else f (tail s).
```

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

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

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

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

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

### 685 Other examples

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

### 695 5.3 Second method

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

#### 700 5.3.1 Generic CPO

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

```

704 Parameter Cc: Type.
705 Parameter bot: Cc.
706 Parameter ordc: Cc-> Cc-> Prop.
707 Parameter bot_is_least: forall x,ordc bot x.
708 Parameter ordc_refl: forall x,ordc x x.
709 Parameter ordc_trans: forall x y z,ordc x y->ordc y z->ordc x z.
710 Parameter ordc_antisym: forall x y,ordc x y->ordc y x->x=y.
711 Parameter ordc_wtot: forall x y z,ordc x z->ordc y z->ordc x y\/ordc y x.
712 Parameter mu: Cc -> nat.
713 Parameter mu_sordc: forall x y,ordc x y-> x<>y-> mu x<mu y.
714
715

```

716 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:

```

718 Inductive C:Type :=
719 |elt:forall (e:Cc),C
720 |cls:forall (ec:EqClass),C.
721 Inductive ord : C-> C-> Prop :=
722 |elt_elt: forall e1 e2,ordc e1 e2 -> ord (elt e1)(elt e2)
723 |elt_cls: forall e ec,
724 (forall t, in t ec-> exists n, ordc e (nth t n) )->
725 ord (elt e)(cls ec)
726 |cls_cls: forall ec,ord (cls ec)(cls ec).
727
728

```

729 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  $\leq^\circ$  is extended to  $\leq$  in Definition 22. Then, the limit of an increasing sequence of elements in  $C$  is defined:

```

734 Definition lim (s:Seq(T:=C))(H:increasing ord s):C :=
735   match (excluded_middle_informative (stabilizing s)) with
736   | left stab =>
737     let (c, _) := constructive_indefinite_description _ stab in c
738   | right nostab =>
739     (cls (class_of (exist _ (fun n => extract_elt (s n))
740       (conj (extract_elt_incr _ Hinc nostab)
741         (incr_nostab_nostab _ Hinc nostab))))))end.
741
742

```

743 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:  $\text{forall } P, \{P\} + \{\sim P\}$  — 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.

758 **On equivalence classes**

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

774 **5.3.2 The CPO of finite and Rose trees**

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

782 **5.3.3 The mirror function**

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

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

793  
 794 `Lemma mirror_pos: forall p t, pos p (mirror t) = pos_rev p t.`  
 795

796 **6 Conclusion, related work, and future work**

797 This paper presents two methods for defining corecursive functions that go beyond the guarded-  
 798 by-constructor setting available in the Coq proof assistant. The first method reuses the dedicated  
 799 coinduction mechanisms available in Coq, which works as long as the underlying coinductive  
 800 datatypes are not mutually dependent with inductive types. The second method is not subject to this

801 restriction, as it does not rely on Coq’s coinduction mechanisms but redefines them, at the cost of  
 802 some additional work. Both methods have in common the interpretation of maximal values in CPOs  
 803 as well-defined corecursive values, and they both rely on a mathematical notion of *productiveness* that  
 804 captures the corresponding intuitive notion of productiveness (the ability of a function to eventually  
 805 generate, for each input, an arbitrarily close approximation of the corresponding output). Both  
 806 methods are implemented in Coq and are illustrated by defining corecursive functions that Coq’s  
 807 dedicated mechanisms reject. This gain in expressiveness is obtained at the cost of using axioms from  
 808 the standard library of Coq, which are known not to introduce inconsistencies: using them amounts to  
 809 losing constructiveness, but gaining access to standard mathematical reasoning. Both methods were  
 810 presented independently of Coq; especially the second one, which *is* independent from Coq’s builtin  
 811 mechanisms for corecursion, could be implemented in other proof assistants. An interesting target is  
 812 Lean [14], a dependently-typed language and proof assistant that includes the additional feature of  
 813 *quotient types* that would naturally encode equivalence classes in the second method.

814 The methods we propose transform a problem currently without solution (defining corecursive  
 815 functions that do not satisfy the guardedness condition) into a problem that is solvable: defining  
 816 and reasoning about functions that approximate the function under definition. In practice the  
 817 approximating functions are recursive, as can be seen from the examples in the paper (Examples  
 818 14 and 31) and from the additional ones in the companion Coq development. Now, if for a given  
 819 corecursive function the corresponding approximating recursive functions are difficult to reason about,  
 820 then applying our methods may be difficult. However, most of the difficulty does not arise from the  
 821 methods, but from the intrinsic complexity of the corecursive function being defined.

## 822 Comparison with related work

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

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

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

843 We now present related work about corecursion in proof assistants and similar formalisms. In Coq,  
 844 corecursive function definitions have to satisfy a guardedness-by-constructors criterion. This criterion  
 845 ensures a strong version of productiveness, namely, that each evaluation step produces a strictly closer  
 846 approximation of the final result than the previous steps. By contrast, productiveness only requires  
 847 that *eventually* a strictly closer approximation is obtained. In some cases, a function that is productive

848 but unguarded can be transformed into an equivalent, guarded function. This has been done for the  
 849 filter function on streams in [3] and generalized in [4] to other unguarded functions. Their idea is to  
 850 use an ad-hoc predicate stating that the definition under study is, in some sense, productive. However,  
 851 their approach does not use a general, formal notion of productiveness, nor does it handle the case  
 852 where corecursive calls are guarded by some non-constructor function, like the mirror function for  
 853 Rose trees presented in this paper. Our approach is not subject to these limitations. In other related  
 854 work, a constructive version of the CPO of streams in Coq is mentioned in [16] in the context of a  
 855 coinductive formalization of Kahn networks. However, the author does not use her formalization of  
 856 CPO to extend the class of corecursive stream functions admissible by Coq.

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

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

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

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

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

## 889 Future work

890 We have encountered corecursive functions that are productive yet do not obey the guarded-by-  
 891 constructors criterion in our planned future work. The Prelude dataflow synchronous programming  
 892 language [9] has a flow sampling construction whose semantics is best described using a filter function  
 893 on colists (which we have defined in the companion Coq development as an instance of our first  
 894 method). This opens the way to a mechanized semantics of Prelude in Coq, which would then enable

895 program verification and semantically correct code generation for the language. While formalizing  
 896 in Coq the paper [17] about the semantics of dataflow languages we have encountered unguarded  
 897 corecursive functions on streams that can also be defined using our first method. More speculative  
 898 future work includes a comparison and possible cross-fertilization of our approach with the sized-type  
 899 approach of Agda and the bounded-natural-functor approach of Isabelle/HOL.

## 900 ——— References ———

- 901 **1** *The Coq Proof Assistant*. <https://coq.inria.fr/>.
- 902 **2** S. Banach. Sur les opérations dans les ensembles abstraits et leur applications aux équations intégrales.  
 903 *Fundam. Math.*, 3:133 – 181, 1922.
- 904 **3** Y. Bertot. Filters on coinductive streams, an application to Eratosthenes' sieve. In *Typed Lambda Calculi*  
 905 *and Applications, 7th International Conference, TLCA 2005, Nara, Japan, April 21-23, 2005, Proceedings*,  
 906 volume 3461 of *Lecture Notes in Computer Science*, pages 102–115, 2005.
- 907 **4** Y. Bertot and E. Komendantskaya. Inductive and coinductive components of corecursive functions in  
 908 Coq. In *Proceedings of the Ninth Workshop on Coalgebraic Methods in Computer Science, CMCS 2008,*  
 909 *Budapest, Hungary, April 4-6, 2008*, volume 203 of *Electronic Notes in Theoretical Computer Science*,  
 910 pages 25–47, 2008.
- 911 **5** Y. Bertot and V. Komendantsky. Fixed point semantics and partial recursion in Coq. In *Proceedings of the*  
 912 *10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming,*  
 913 *July 15-17, 2008, Valencia, Spain*, pages 89–96, 2008.
- 914 **6** J. Biendarra, J. C. Blanchette, M. Desharnais, L. Panny, A. Popescu, and D. Traytel. *Defining*  
 915 *(Co)datatypes and Primitively (Co)recursive Functions in Isabelle/HOL*. [https://isabelle.in.tum.](https://isabelle.in.tum.de/doc/datatypes.pdf)  
 916 [de/doc/datatypes.pdf](https://isabelle.in.tum.de/doc/datatypes.pdf).
- 917 **7** J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. *Defining Nonprimitively*  
 918 *(Co)recursive Functions in Isabelle/HOL*. [https://isabelle.in.tum.de/dist/Isabelle2021/](https://isabelle.in.tum.de/dist/Isabelle2021/doc/corec.pdf)  
 919 [doc/corec.pdf](https://isabelle.in.tum.de/dist/Isabelle2021/doc/corec.pdf).
- 920 **8** A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2013.
- 921 **9** J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*.  
 922 PhD thesis, Institut Supérieur de l'Aéronautique et de l'Espace, Toulouse, France, 2009.
- 923 **10** S. R. Ghorpade and B. V. Limaye. *A Course in Calculus and Real Analysis*. Undergraduate Texts in  
 924 Mathematics. Springer, 2018.
- 925 **11** E. Giménez. Codifying guarded definitions with recursive schemes. In *Types for Proofs and Programs,*  
 926 *International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of  
 927 *Lecture Notes in Computer Science*, pages 39–59. Springer, 1994.
- 928 **12** C.-K. Hur, G. Neis, D. Dreyer, and V. Vafeiadis. The power of parameterization in coinductive proof. In  
 929 *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*  
 930 *'13, Rome, Italy - January 23 - 25, 2013*, pages 193–206. ACM, 2013.
- 931 **13** D. Kozen and N. Ruozzi. Applications of metric coinduction. *Log. Methods Comput. Sci.*, 5(3), 2009.
- 932 **14** *The Lean Proof Assistant*. <https://leanprover.github.io/>.
- 933 **15** K. Rustan M. Leino and M. Moskal. Co-induction simply - automatic co-inductive proofs in a program  
 934 verifier. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014,*  
 935 *Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 382–398. Springer, 2014.
- 936 **16** C. Paulin-Mohring. A constructive denotational semantics for Kahn networks in Coq. Available at  
 937 <https://www.lri.fr/~paulin/PUBLIS/paulin07kahn.pdf>.
- 938 **17** T. Uustalu and V. Vene. The essence of dataflow programming. In *Programming Languages and Systems,*  
 939 *Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780  
 940 of *Lecture Notes in Computer Science*, pages 2–18. Springer, 2005.
- 941 **18** N. Veltri and N. van der Weide. Guarded recursion in agda via sized types. In *4th International*  
 942 *Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019,*  
 943 *Dortmund, Germany*, volume 131 of *LIPICs*, pages 32:1–32:19. Schloss Dagstuhl - Leibniz-Zentrum für  
 944 Informatik, 2019.

## 12:22 Defining Corecursive Functions in Coq Using Approximations

- 945 **19** G. Winskel. *The Formal Semantics of Programming Languages, an introduction*. MIT Press, 1993.
- 946 **20** *The Agda Proof Assistant*. [https://agda.readthedocs.io/en/v2.6.2/getting-started/  
947 what-is-agda.html](https://agda.readthedocs.io/en/v2.6.2/getting-started/what-is-agda.html).
- 948 **21** *The Isabelle/HOL Proof Assistant*. <https://isabelle.in.tum.de/>.