Replace this file with `prentcsmacro.sty` for your meeting,
or with `entcsmacro.sty` for your meeting. Both can be
found at the ENTCS Macro Home Page.

Inductive and Coinductive Components of Corecursive Functions in Coq

Yves Bertot [1,2]

Project MARELLE
INRIA Sophia Antipolis
France

Ekaterina Komendantskaya[1,3]

Project MARELLE
INRIA Sophia Antipolis
France

Abstract

In Constructive Type Theory, recursive and corecursive definitions are subject to syntactic restrictions which guarantee termination for recursive functions and productivity for corecursive functions. However, many terminating and productive functions do not pass the syntactic tests. Bove proposed in her thesis an elegant reformulation of the method of accessibility predicates that widens the range of terminative recursive functions formalisable in Constructive Type Theory. In this paper, we pursue the same goal for productive corecursive functions. Notably, our method of formalisation of coinductive definitions of productive functions in Coq requires not only the use of ad-hoc predicates, but also a systematic algorithm that separates the inductive and coinductive parts of functions.

Keywords: Coq, Induction, Coinduction, Productiveness, Guardedness, Accessibility Predicates.

1 Introduction

The proof assistant Coq [44] is an implementation of the Calculus of Inductive Constructions [23] extended with inductive [40] and coinductive [31] types. Implementations of coinductive types were first suggested by Coquand in [22] and implemented in Coq by Gimenez [31]. Coq has proved to be an effective tool for working with different kinds of final coalgebras, such as the final coalgebra of streams [12], the final coalgebra of infinite binary trees [12], and some others. For instance, the machinery of Coq was used to define algebraic structures on real numbers [28,38,11].

[1] The work was funded by the INRIA CORDI post-doctoral program and the ANR project "A3Pat" ANR-05-BLAN-0146.

[2] Email: `yves.bertot@inria.fr`

[3] Email: `ekaterina.komendantskaya@inria.fr`

The specification language of Coq makes it possible to model the types and programs of typed programming languages. Typed programming languages usually provide a few basic types and a mechanism that allows the definition of *inductive* data types. When defining an inductive data type, we need to introduce constructors to generate the elements of the new type. A very well-known inductive data type is the type of natural numbers, defined using two constructors: 0 and S.

In Coq, one can define coinductive types in the same fashion as inductive types, using a few basic constructors that are also related to destructors through the pattern-matching construct. One can also use destructors when he wants to emphasise the duality relative to inductive constructors. We will illustrate this in Section 2.

A key notion in typed (functional) programming is the notion of *recursion* by which an object being defined refers to itself. Functions defined over inductive types are recursive by nature. E.g., most of the functions defined on natural numbers need to be defined recursively. In Section 2, we will discuss in more detail the syntax of such functions in Coq.

Associated with recursion, there is a crucial question of termination. In general, there is no guarantee that a recursive function will always terminate. A solution to this problem is to use only *structurally recursive* definitions. A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument. This guarantees that the recursion terminates. Constructive Type Theory in general, and Coq in particular, impose the structurally recursive condition on every defined recursive function. And thus, all functions are guaranteed to terminate in Coq, as we further explain in Sections 3 and 4.

Definitions where the recursive calls are not required to be on structurally smaller arguments, that is, where the recursive calls can be performed on any argument are called *general recursive* definitions. Many important and well known algorithms are not structurally recursive but general recursive. Although many general recursive algorithms can be proved to terminate, there is no *syntactic condition* that guarantees their termination and thus, general recursive algorithms have no direct formalisation in Constructive Type Theory and in Coq. Several solutions to the problem of encoding general recursive functions have been suggested in [1,9,20]. Other methods include the use of *accessibility predicates* [2,4,39] and *ad-hoc predicates* [14], the two latter methods are surveyed in Section 4. Most of these methods ultimately rely on structural recursion.

Already in the case of inductive types, there exists a difference between the class of functions satisfying the *semantic condition* of termination, and the class of functions satisfying the *syntactic condition* of structural recursion. It is significant that, when working with coinductive types in Coq, we find the notions of productivity [24,43] and guardedness [22,31] dual to those of termination and structural recursion. This is analysed in Sections 3 and 4. Similarly to the inductive case, the guardedness condition bans Coq formalisations for many useful productive functions. This problem was tackled (e.g.) in [1,9,29,30] for type theory and in [36] for HOL.

A particular application of an ad-hoc predicate for defining corecursive filter function on streams in Coq first appeared in [10], where it was used to formalise

Eratosthenes' sieve. Already in that example, the filter function was decomposed into inductive and coinductive components, in order to become guarded. A similar result, also for filter functions on streams, was described by M. Niqui [37,38].

However, neither [10] nor [37,38] included the systematic description of the method of separating inductive and coinductive components of productive values in general case. The general description of how the ad-hoc predicates can be constructed for corecursive functions was missing, too.

In this paper we venture to generalise the results obtained in [10], and describe the method of formalising productive (non-guarded) values in Coq for any given function and for any given data type. In Section 4 we describe the class of functions that are covered by this general method; and give counterexamples of functions that are not.

In Sections 5 and 6, we give a general method of separating the inductive and coinductive components of productive coinductive values in Coq. In particular, Section 5 is devoted to the inductive component and gives general characterisation of the method of building ad-hoc predicates for formalising productive values in Coq. In Section 6, we characterise the coinductive component.

Notably, there are two predicates, `eventually` and `infinite`, that are essential for characterisations of inductive and coinductive components, respectively. The similar "eventually" and "infinite" were first introduced as temporal modalities in [42], and their coalgebraic specification was given by Jacobs in [34]. We show how to formalise lemmas relating `eventually` and `infinite`, and use these lemmas to tie together the inductive and coinductive components of productive functions.

In Section 7, we prove the "recursive equation lemmas" establishing that our formalisations of the productive functions are correct. Prior to this paper, such lemmas have never been established; in particular, they were missing in [10,37,38].

Finally, in Section 8 we conclude and outline the further work to be done.

2 Inductive and Coinductive Types in Coq

In this section, we will give a short exposition of how inductive and coinductive types are defined and used in Coq. We will introduce several running examples. The related work of developing the theory of corecursive definitions was done in HOL and mechanised using Isabelle [41]. For a more detailed introduction to Coq, see [12].

As we have already mentioned in the introduction, inductive data types are defined by introducing a few basic constructors that generate the elements of the new type.

Definition 2.1 The definition of the inductive type of natural numbers is built using two constructors `O` and `S`:

```
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.
```

After the inductive type is defined, one can define its inhabitants and functions on it. Most functions defined on the inductive type must be defined recursively, that is, by describing values for different patterns of the constructors and by allowing calls to the same function on variables taken from the patterns. Thus, division by two on natural numbers is computed by a recursive function `div2`:

$$\begin{cases} \mathrm{div2}(0) = 0 \\ \mathrm{div2}(\mathrm{S}\ 0) = 0 \\ \mathrm{div2}(\mathrm{S}(\mathrm{S}\ n')) = \mathrm{S}(\mathrm{div2}\ n'). \end{cases}$$

And this can be modelled in Coq as follows.

Definition 2.2

```
Fixpoint div2 n : nat :=
  match n with
  | O => 0
  | S O => 0
  | S (S n') => S (div2 n')
  end.
```

It is essential that recursive functions are defined over arguments of inductive types. For instance, natural numbers are given as arguments in the above definition.

We can use the inductive type of natural numbers and the defined function `div2` to obtain a new recursive function. The function of discrete logarithm of base 2 for natural numbers is computed by a recursive function `log` satisfying the equation

$$\begin{cases} \log(\mathrm{S}\ 0) = 0 \\ \log(\mathrm{S}(\mathrm{S}\ n)) = \mathrm{S}(\log \mathrm{S}(\mathrm{div2}\ n)). \end{cases}$$

But this, as we further explain in Section 4, cannot be modelled directly in Coq. However, we can define the inductive predicate characterising the arguments for which the discrete logarithm is well-defined:

Definition 2.3

```
Inductive log_domain : nat -> Prop :=
| log_domain_1 : log_domain 1
| log_domain_2 :
 forall p: nat, log_domain (S (div2 p)) -> log_domain (S (S p)).
```

It was observed in [35], that induction gives rise to initial algebras, while coinduction gives rise to final coalgebras; and the basic duality between algebras and coalgebras can be expressed as *construction* versus *observation*. Let us have a closer look at how this idea is realised in Coq.

The following is the definition of a coinductive type of infinite streams, built using one constructor `SCons`.

Definition 2.4 The type of streams is given by

```
CoInductive str (A:Set) : Set := SCons: A -> str A ->  str A.
```

Typically, a stream has the form `SCons a s`, where a is an element of some set A, and s is a stream. There exists a common convention to write `a :: s` for `SCons a s`. Although syntactically the above definition is very similar to the definitions of the inductive types, this coinductive definition supports well the dichotomy *construction* versus *observation*: given an infinite stream, we can only observe its head, and pass on to its tail. The tail will be infinite, too; and only its first element can be observed next.

Definition 2.5 The coinductive function `repeat` takes as argument an element a of some set A and yields a stream where a is repeated indefinitely:

```
CoFixpoint repeat (a: A): str A := SCons a (repeat a).
```

Notably, we do not have to impose any type requirements on arguments of the function, but we require the produced values to be of coinductive type.

Properties of coinductive data often need to be expressed with coinductive predicates. To prove some properties of infinite streams, we use the method of observation. For example, to prove that the two lists are *bisimilar*, we must observe that their first elements are the same, and continue the process with the next.

Definition 2.6 Bisimilarity is expressed in the definition of the following coinductive type:

```
CoInductive bisimilar_s:  str A -> str A -> Prop :=
|bisim: forall (a : A) (s s' : str A), bisimilar_s s s' ->
                    bisimilar_s (SCons A a s)(SCons A a s').
```

The definition of `bismilar_s` corresponds to the conventional notion of bisimilarity as given, e.g. in [35]. Lemmas and theorems analogous to the *coinductive proof principle* of [35] are proved in Coq and can be found in [12].

Infinite streams are not the only kind of data that is handled by coinductive machinery of Coq. We can work with different types of infinite data types, such as infinite binary trees or infinite expression trees, see also [3]. *Expression trees* are trees in which every node has one or two children. The nodes of these trees are labelled with elements of sets A and B, and we will call them A-nodes and B-nodes, respectively. We will denote expression trees by $E(A, B)$. The expression trees were extensively used in formalising real number arithmetic, see [38,25].

Definition 2.7 We coinductively define the expression trees in Coq:

```
CoInductive ETrees (A B : Set) : Set :=
| A_node : A -> ETrees A B -> ETrees A B
| B_node : B -> ETrees A B -> ETrees A B  -> ETrees A B.
```

We define a bisimilarity relation `bisimilar_t` for this type in [13]. Also, in [13] we show that `bisimilar_s` and `bisimilar_t` are equivalence relations.

We have seen that in Coq, inductive types are domains of recursive functions and coinductive types are codomains of corecursive functions. We have also observed that syntactically, the definitions of inductive and coinductive types in Coq follow one and the same scheme.

3 Termination and Productivity

In this section, we will discuss two computational concepts that depend on the recursive nature of *inductive* and *coinductive* definitions, and those are of *termination* and *productivity*.

In Coq, as in any other type-theoretic theorem provers (HOL, PVS, and others, see [7]), all computations must terminate. Because propositions are represented by types and proofs by programs, according to *Curry-Howard* isomorphism [6,32,27], we cannot allow non-terminating proofs, as they may lead to inconsistency. There is a technical reason for the termination requirement, too: to decide type-checking of dependent types, we need to reduce type expressions to normal form ([7] is a very good survey of proof techniques used in type theory).

Example 3.1 The function `div2` from Definition 2.2 is terminative.

Given any natural number excluding 0 as an input, the function `log` described in Section 2 is terminative.

For corecursive functions there is a dual notion to that of termination - productivity. The notion of termination is used to ensure totality of functions on *finite* objects (initial algebras [35]); while productivity is used to ensure totality of functions on *infinite objects* (final coalgebras). The infinite objects that we are going to use as running examples through the paper are streams and expression trees as defined in 2.4 and 2.7.

The notion of productivity was first defined in [24,43], in terms of domain theory. For a very careful domain theoretic characterisation of productivity of streams and trees, see [37]. However, we will omit the domain theoretic definitions here, and describe productivity from a computational point of view. Namely, we use recursive functions in order to define classes of productive functions in Coq. We hope that this section will give the reader the opportunity to capture the spirit of a functional approach to productivity. For more on productivity of infinite data structures, see, e.g. [19,22,26].

Values in co-inductive types usually cannot be observed as a whole, because of their infiniteness. Instead, they are often described as some finite tree-like structures where some sub-terms still remain to be computed and are described using unevaluated functions applied to arguments. Values in co-inductive types are said to be *productive* when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

When the co-inductive type being considered is the type of streams, we can ask to see the element of the stream at position n using the following function:

Definition 3.2

$$\begin{cases} \texttt{nth 0 (SCons a tl)} = \texttt{a} \\ \texttt{nth (S n) (SCons a tl)} = \texttt{nth n tl} \end{cases}$$

It is a regular structural recursive function with structural argument n.

A given stream s is productive if we can be sure that the computation of the list `nth n s` is guaranteed to terminate, whatever the value of `n` is (satisfying the condition for the function `nth` will be enough to ensure that it is satisfied for any other recursive function, but this is hard to prove).

Example 3.3 For any `n`, the value `repeat n` is productive, (see also Definition 2.5).

We can do the similar recursive observations on coinductive expression trees. We describe the observation of values at different positions in an expression tree using a type of direction `direct` and a function `fetch` that takes sequences of directions to access a given position in a tree.

Definition 3.4 Assuming the type `direct` is given by the following:
`Inductive direct : Type := L | R`, the function `fetch` of the type
`forall A B:Set, list direct -> ETrees A B -> A+B` is defined as follows:

$$\begin{cases} \texttt{fetch nil (A\_node a t)} = \texttt{inl a} \\ \texttt{fetch (\_::tl) (A\_node a t)} = \texttt{fetch tl t} \\ \texttt{fetch nil (B\_node b t1 t2)} = \texttt{inr b} \\ \texttt{fetch (L::tl) (B\_node b t1 t2)} = \texttt{fetch tl t1} \\ \texttt{fetch (R::tl) (B\_node b t1 t2)} = \texttt{fetch tl t2} \end{cases}$$

An expression tree `t` is said to be productive if the computation of `fetch l t` is guaranteed to terminate whatever the value of `l` is.

We call a function *productive at the input value* i, if it outputs a productive value at i. This understanding of productivity of functions differs slightly from the approach of [43,37,38], where a function is said to be productive if it takes productive values as an input and outputs productive values. Let us explain this on the following examples.

The function *dyn* defined below takes an arbitrary value as an input. It returns productive values only for some inputs.

Definition 3.5 Let A, B be of type `Set`. For a predicate $P: B \to \texttt{bool}$ and functions $h: B \to A$, g, $g': B \to B$, we define the function *dyn* as follows:

$$\text{dyn (x)} = \begin{cases} \text{SCons } h(x) \text{ (dyn } (g(x))) & \text{if } P(x) \\ \text{dyn } (g'(x)) & \text{otherwise.} \end{cases}$$

Thus, unlike [37,38], we do not require productive functions to take coinductive values as an input. The only requirement we impose is that the produced data must be coinductive. This approach is consistent with the understanding of coinductive functions in Coq: arguments of corecursive functions can be of any type, and only the produced values are required to be of coinductive type.

There is a tradition of studying *productive* functions, probably meaning that these functions are *totally productive* once given productive values as arguments. However, in this work we want to study functions that are only *partially productive*, that is, functions that will return productive values only for a subset of their input type, a subset which we characterise precisely using inductive and coinductive predicates. We illustrate this further as follows.

Consider the filter function on streams that was formalised in [10] and was used to filter prime numbers.

Definition 3.6 (Filter for streams).
For a given predicate P,

$$\text{filter (SCons x tl)} = \begin{cases} \text{SCons x (filter tl)} & \text{if } P(x) \\ \text{filter tl} & \text{otherwise.} \end{cases}$$

The filter function examines the first element `x` of a given list (`SCons x tl`) for a property P, and, in case the property is satisfied, it uses `x` to form a new list. Then it recursively examines the tail of the stream.

In general, filter functions on streams make it possible to define non-productive values. The filter function above can be non-productive on certain values and for certain predicates.

Example 3.7 For instance, computing `nth 0 (filter even (repeat 1))` provokes the following computation:

`filter even (repeat 1) repeat 1` $\rightsquigarrow$ `filter even (1::repeat 1)` $\rightsquigarrow$ `filter even (repeat 1)`...

The first arrow comes from computing a little portion of (repeat 1). The second arrow comes from observing that 1 is not even and reducing the definition of filter. This leads to the same position as initially without producing the first element of the stream required by (`nth 0`). The same computation should be triggered again and indefinitely.

The method of formalising corecursive functions that we propose in this paper, makes it possible to formalise such functions in Coq, using inductive and coinductive predicates to characterise the arguments on which these functions output productive values. By abuse of terminology that causes no confusion, we will call these functions "productive".

We conclude this section with another important example of a partially productive corecursive function. The following functions generalise the filter from Definition 3.6 to the case of expression trees with dynamic filtering:

Definition 3.8 (Dynamic Filter for Expression Trees).
Let P and P_1 be predicates, and let $h_1 : A_1 \to A_2$, $h_2 : B_1 \to A_2$, $h'_1 : A_1 \to A_1$, $h'_2 : B_1 \to B_1$, $h3$: `ETrees` $A_1 B_1 \to$ `ETrees` $A_1 B_1$. The latter functions will "dynamically" change the filtered values in the process of filtering:

$$\text{e\_filter (A\_node } a\ t_1) = \begin{cases} \text{A\_node } h_1(a) \text{ (e\_filter (A\_node } h'_1(a)\ t_1)) & \text{if } P(a) \\ \text{e\_filter } h_3 \text{ (A\_node } a\ t_1) & \text{otherwise.} \end{cases}$$

$$\text{e\_filter (B\_node } b\ t_1\ t_2) = \begin{cases} \text{A\_node } h_2(b) \text{ (e\_filter (B\_node } h_2'(b)\ t_1\ t_2)) & \text{if } P_1(b) \\ \text{e\_filter } h_3 \text{ (B\_node } b\ t_1\ t_2) & \text{otherwise.} \end{cases}$$

The function was used to establish a normalisation algorithm for an admissible representation of a closed interval of real numbers in [27,37].

We have discussed the reasons why termination and productivity are important for our theory, and also we showed how they relate to each other. We introduced important examples of partially productive functions. In the next section, we will discuss the way how terminative and productive functions are syntactically defined in Coq.

4 Structural Recursion and Guardedness; Method of Ad-hoc Predicates

There are two syntactic tests that ensure termination and productivity of functions in Coq, they are called *structurally smaller calls condition* [21,40] and *guarded-by-constructors condition* [31].

A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument. In this way we can be sure that the recursion terminates.

Guardedness is a sufficient condition for productivity. It is described in two steps. The first step defines *pre-guarded positions*. A position is pre-guarded if it occurs as the root of the function body, or if it is a direct sub-term of a pattern-matching construct or a conditional statement, which is itself in a pre-guarded position.

The second step defines *guarded positions*. A position is guarded if it occurs as a direct sub-term of a constructor for the co-inductive type that is being defined and if this constructor occurs in a pre-guarded position or a guarded position. A corecursive function is *guarded* if all its corecursive calls occur in guarded positions.

Guardedness ensures that at least one constructor of the co-inductive type is produced for each corecursive call, and thus at least a fragment of corecursive data is produced each time a corecursive call occurs.

Example 4.1 The function `div2` from Definition 2.2 is structurally recursive.

The function `repeat` from Definition 2.5 is guarded.

Definitions where the recursive calls are not required to be on structurally smaller arguments, that is, where the recursive calls can be performed on any argument, are called *general recursive*. Many important algorithms, such as the algorithm of computing logarithm discussed in Section 2, are not structurally recursive but general recursive.

The standard way of handling general recursion in constructive type theory uses a well-founded recursion principle derived from the accessibility predicate `Acc` [2,39]. The idea behind the accessibility predicate is that an element a is *accessible* by a relation $<$ if there is no infinite decreasing sequence starting from a. A set A is said to be *well-founded* with respect to $<$ if all its elements are accessible by $<$. Hence, to guarantee that a general recursive algorithm that performs the recursive calls on elements of type A terminates, we have to prove that A is well-founded and that the arguments supplied to the recursive calls are smaller than the input.

In Coq, the method was implemented in [33,4], see also [12]. The method of using accessibility predicates was improved by Bove in her thesis [15] and series of papers [14,17,16,18]. The core of the improvement proposed by Bove was to separate computational and logical parts of the definitions of general recursive algorithms. That is, the method amounts to defining an inductive special-purpose accessibility (ad-hoc) predicate that characterises the inputs on which the algorithm terminates. Proving that a certain function is total amounts to proving that the corresponding accessibility predicate is satisfied on every input.

Example 4.2 We continue Definition 2.3, and formalise the function log defined in Section 2. We need an inversion lemma about the predicate `log_domain` expressing that when `x` has the form "`S (S p)`", if `x` is in the domain, then "`S (div2 p)`" also is:

```
Lemma log_domain_inv :
forall x p : nat, log_domain x -> x = S(S p) ->
                  log_domain (S (div2 p)).
```

At each recursive call of the function, we need to use the above *inversion lemma* stating that the proof argument for recursive call can be deduced from the initial proof argument.

We also need to express that 0 is not in the domain of the function:
`Lemma log_domain_non_0: forall x :nat, log_domain x -> x` $\neq$ `0.`

Now we can use the ad-hoc predicate `log_domain_inv`, together with the function `div2` from Definition 2.2 and define the function `log` as a usual structurally recursive function, but where the structural argument is the *proof argument*; we box the proof arguments in the example below.

```
Fixpoint log (x:nat) (h: log_domain x) struct h : nat :=
match x as y return x = y -> nat with
| 0 => fun h' => False_rec nat (log_domain_non_0 x h h')
| S 0 => fun h' => 0
| S (S p) =>
   fun h' => S (log (S (div2 p)) (log_domain_inv x p h h'))
end (refl_equal x).
```

It is important that the Coq checker can recognise `log_domain_inv x p h h'` as a structurally smaller proof with respect to `h`. See also [12], Section 15.4.

However, the source of our interest in this method of ad-hoc predicates lies not in general recursive functions, but in productive non-guarded corecursive functions. In the same way as the syntactic structurally recursive condition removes many useful terminative functions from the picture, the syntactic guardedness condition of Coq rejects many productive functions.

Informally speaking, the guardedness condition insures that

* each corecursive call is made under at least one constructor;

** if the recursive call is under a constructor, it does not appear as an argument of any function.

Violation of any of these two conditions makes a function rejected by the guardedness test in Coq.

Example 4.3 The corecursive functions from Definitions 3.6, 3.8 and 3.5 are not directly formalisable in Coq, because they do not satisfy the guardedness condition *. However, we know about many useful examples of filters from Definitions 3.6 and 3.8 that are productive, [10,38,25].

All the examples we have given so far exhibit *partially* productive non-guarded functions. And this may give the reader an impression that it is "partiality" of corecursive functions that makes them non-guarded. However, totally productive functions can be non-guarded, too. Moreover, we claim that every terminative function gives rise to a non-guarded *totally* productive function, or, in other words, one can always craft a totally productive non-guarded function using a terminative function.

Example 4.4 For example, terminative function $x - 1$ gives rise to the totally productive function `f: str nat -> str nat`:

$$\texttt{f (x::y::tl)} = \begin{cases} \texttt{x::f(y::tl)} & \text{if } x \leq y \\ \texttt{f((x-1)::y::tl)} & \text{otherwise.} \end{cases}$$

This second corecursive call does not satisfy the guardedness condition *.

In general, the fact that one uses a terminative function to define a corecursive function guarantees productiveness, but in the same time one can always craft the resulting corecursive function in such a way that a corecursive call will not be under a constructor, thus violating guardedness condition *.

The next example illustrates the class of functions that fail to satisfy the guardedness condition **.

Example 4.5 Consider the following function computing lists of ordered natural numbers:

```
nats = (SCons 1  (map (+ 1) nats)).
```

where the function `map` above is defined as follows:

```
map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
```

That is, we start with the list with 1, then add `(+ 1)` to the head of the list to get the second element 2; and continue the same computation on the tail.

The recursive call here is made under the constructor `SConc`, but it also appears on the argument place of the function `map`. The latter fact violates the guardedness condition **, and hence the function will be rejected by Coq. Despite of this, the value `nats` is known to be productive.

The similar problem arises with the Example 1 in [26]: the function is productive but non-guarded.

In [10] was found a Coq formalisation of a coinductive *filter* function from Definition 3.6. The work was aimed at showing that values produced through filter functions could still be described as guarded corecursive functions. Notably, the solution involved the method of building an ad-hoc predicate similar to the ad-hoc predicate of Bove. The fact that the method introduced for inductive algorithms is expandable to coinductive ones is significant in its own right. But the example of [10] exhibited even more: the method of formalising corecursive filter functions requires separation of not only "logical" and "computational" parts of algorithms, but also of inductive and coinductive parts.

In the remaining sections, we develop the general method of formalising productive functions that fail to satisfy the guardedness condition *. Our method does not cover the class of productive functions that do not satisfy the guardedness condition **. Such functions were studied (e.g.) in [1,26].

5 Inductive Components of Corecursive Functions

For each productive function, we will describe two modalities, `eventually` and `infinite` that we are going to use when characterising the inductive and coinductive components of productive functions. These are variations of $\Diamond$ and $\Box$ specified coalgebraically in [34], and they originate from the temporal modalities introduced in [42].

The conventional definitions of `eventually` and `infinite` as formalised, (e.g.) in [12], express whether a given stream `s` satisfies some given property `P` at least once or infinitely many times. We modify these predicates in order to characterise the conditions for a corecursive function to perform a guarded corecursive step. In this section, we consider `eventually` and its role in defining a recursive function which characterises the *inductive component* of a given corecursive function.

We propose to take as a starting point the defining equation of the corecursive function we wish to formalise; as e.g., Definitions 3.6, 3.8, 3.5. Then we define predicates on the input types of these functions. The predicate `eventually` captures the conditions for the corecursive function to perform the next guarded corecursive step. For any given function, and on any given data type, we can define `eventually` systematically as follows:

1. *The predicate* `eventually` *is defined inductively, with one constructor for each branch appearing in the function definition.*

For example, we will notice that in Definitions 5.2 and 5.3 the number of constructors we use to define `eventually` will vary from two to four, according to the number of branches appearing in Definitions 3.6, 3.8.

2. *When a branch contains only guarded recursive calls, the constructor expresses that the input data satisfies the* `eventually` *predicate as soon as it satisfies all the conditions needed to reach this branch.*

For instance, the `dyn` function from Definition 3.5 performs a boolean test on the predicate `P`, and returns the value `SCons (h x) (dyn (g x))` if `(P x)` is `true`, or the value `(dyn (g' x))` if `(P x)` is `false`. Each of these values contains a recursive call, but the first call on `(g x)` is guarded while the other call on `(g' x)` is non-guarded.

For the first recursive call, we can directly provide a constructor to this inductive predicate that states that `x` satisfies the predicate if `(P x)` is `true`. (We choose to name the `eventually` predicate `eventually_dyn`.)

```
ev_dyn1 : P x = true  -> eventually_dyn x
```

3. *When a branch contains non-guarded recursive calls, the constructor expresses that the input data satisfies the predicate as soon as it satisfies all the conditions leading to this branch, and the inputs to all non-guarded recursive calls satisfy the predicate.*

Let us return to the `dyn` function. For the second recursive call, we provide a constructor stating that `x` satisfies the predicate when `(P x)` is `false` only if the recursive call would then reach a value that already satisfies the predicate:

```
ev_dyn2 : P x = false  -> eventually_dyn (g' x) -> eventually_dyn x
```

We have covered all possible recursive branches in the behaviour of `dyn`, so we can collect the constructors in an inductive definition:

Definition 5.1

```
Inductive eventually_dyn (x:B) : Prop :=
| ev_dyn1 : P x = true  -> eventually_dyn x
| ev_dyn2 : P x = false  -> eventually_dyn (g' x) -> eventually_dyn x.
```

The same systematic approach gives the predicates `eventually_s` for the filter on streams and `eventually_t` for the filter on expression trees:

Definition 5.2

```
Inductive eventually_s: str A ->  Prop :=
| ev_b: forall x s, P x ->  eventually_s (SCons A x s)
| ev_r: forall x s,
  ~ P x -> eventually_s s -> eventually_s (SCons A x s).
```

Note that `eventually_t` has four constructors, depending on whether the input data is an A-node or a B-node and depending on whether the observable data carried by this node satisfies a predicate P or not, precisely as in Definition 3.8:

Definition 5.3 `Inductive eventually_t: ETrees A1 B1  -> Prop :=`

```
|ev_rB: forall x t t1 ,
  ~P1 x -> eventually_t (h3 (B_node A1 B1 x t t1))  ->
  eventually_t (B_node A1 B1 x t t1)
|ev_bB: forall x t t1, P1 x  -> eventually_t (B_node A1 B1 x t t1)
|ev_rA: forall x t,
  ~P x -> eventually_t (h3 (A_node A1 B1 x t)) ->
  eventually_t (A_node A1 B1 x t)
|ev_bA: forall x t, P x -> eventually_t (A_node A1 B1 x t).
```

Thus, `eventually_dyn`, `eventually_t` and `eventually_s` are constructed using one systematic method.

The `eventually` predicates are satisfied if the conditions for the recursive function to perform a guarded corecursive call are satisfied at least once. In the future, we will say that `eventually` ensures only the *first-step-productivity*, as opposed to the conventional *productivity* of Section 3.

Using the inductive definition of `eventually` predicates, we can describe the first component of the function we want to compute. This component is a recursive function that performs all the computations and tests that lead to the first guarded corecursive call. For lack of a better name, we will call this function the *inductive component* of the corecursive function being defined. Separating this inductive component from the rest of the function behaviour is interesting: it will hide all the non-guarded corecursive calls and make it possible to go directly from one guarded call to the next one.

This inductive component is modelled by following the technique of ad-hoc predicates where the chosen predicate is the `eventually` predicate.

The method of ad-hoc predicates relies on inversion lemmas. Each inversion lemma expresses that the argument to the recursive call satisfies the `eventually` predicate if the initial argument does. Moreover, the proof of each inversion lemma is carefully crafted to make sure that the lemma's conclusion is understood as a *structurally smaller proof* with respect to one of the lemma's premises (this is the essence of recursion on an ad-hoc predicate, see [12]).

There is one inversion lemma needed for the `dyn` function and the filter on streams. Two inversion lemmas are needed to formalise the filter on trees. In general, the number of the required inversion lemmas equals to the number of non-guarded calls in the initial corecursive function.

Lemma 5.4 (Inversion Lemmas)
```
Lemma eventually_dyn_inv :
 forall x, eventually_dyn x -> P x = false -> eventually_dyn (g' x).

Lemma eventually_s_inv: forall (s : str A),
 eventually_s s -> forall x s', s = SCons A x s' ->
                                           ~ P x ->  eventually_s s'.

Lemma eventually_t_inv:
 forall t : ETrees A1 B1, eventually_t t ->
 forall (x: A1) (t': ETrees A1 B1), t =  (A_node A1 B1 x t') ->
                    ~ P x -> eventually_t (h3 (A_node A1 B1 x t')).

Lemma eventually_t_inv':
 forall t : ETrees A1 B1,  eventually_t t ->
 forall(x:B1) (t' t1: ETrees A1 B1), t = (B_node A1 B1 x t' t1) ->
                 ~P1 x -> eventually_t (h3 (B_node A1 B1 x t' t1)).
```

We give the full proofs of these inversion lemmas in [13].

We use these inversion lemmas to define the inductive component of corecursive functions: this inductive component does all computations until it performs the first guarded corecursive call. For instance, for the function on streams, the inductive component performs all computation steps until one reaches the first element in the stream that satisfies the property `P` used for filtering.

The data produced by the inductive component is organised in two parts: the first part contains the observable data that is included in the head constructor of the output. Such a head constructor is necessarily present, because the `eventually` predicate holds and this means that there is a guarded corecursive call. The second part contains the argument for the next recursive call of the initial corecursive function.

The inductive component can perform recursive calls to itself, but they go from values satisfying the `eventually` predicate to values satisfying the `eventually` predicate, thanks to the inversion lemmas.

Definition 5.5 The inductive component of the recursive function for `dyn` is called `pre_dyn`; the additional (proof) arguments are boxed below:

```
Fixpoint pre_dyn (x:B) (d:eventually_dyn x) {struct d} : A*B :=
  match P x as b return P x = b -> A*B with
    true => fun t => (h x, g x)
  | false => fun t => pre_dyn (g' x) (eventually_dyn_inv x d t)
  end (refl_equal (P x)).
```

This code exhibits two aspects of programming with ad-hoc predicates, which make them complicated to read for neophytes: first, the function has an extra proof argument and recursive calls must also have this extra proof argument. Second, the body of the function contains a pattern-matching construction that must be applied to a proof of `P x = P x`. Inside the proof matching construct, this equality is transformed differently in the two branches: in one branch it takes the form of `P x = true` while in the other it takes of the form of `P x = false`.

For the function `e_filter`, the inductive component is named `pre_filter_t`.

Definition 5.6
```
Fixpoint pre_filter_t (t: ETrees A1 B1)
(h: eventually_t t){struct h} : A2 * ETrees A1 B1  :=
match t as b return t = b ->  A2*ETrees A1 B1  with
|A_node x t' => fun heq : t = (A_node A1 B1 x t') =>
  match P_dec x with
  |left _ => (h1 (inl B1 x), A_node (fst (h2 (inl B1 x))) t')
  |right hn => pre_filter_t (h3 (A_node A1 B1 x t'))
                        (eventually_t_inv t h x t' heq hn)
  end
|B_node x t' t1 => fun heq': t = (@B_node A1 B1 x t' t1) =>
  match P1_dec x with
  |left _ => (h1 (inr A1 x), B_node (snd (h2 (inr A1 x))) t' t1)
  |right hn' => pre_filter_t(h3 (B_node A1 B1 x t' t1))
                      (eventually_t_inv' t h x t' t1 heq' hn')
  end
end (refl_equal t).
```

We formulate the omitted inductive component of the filter for streams in [13].

In this section, we have shown the algorithm to formalise the inductive components of several recursive functions. It remains to describe how computation can carry on beyond the first guarded corecursive call.

6 Coinductive Components of Corecursive Functions

In this section we characterise the coinductive part of corecursive functions. Corecursive computations are introduced by repeating computations performed by the inductive component defined in the previous section. Now, because the inductive component finds guarded corecursive call, guarded co-recursion can take place. However, this can only happen if recursive calls satisfy the `eventually` predicate repeatedly. We need an extra predicate to express this.

We define a coinductive predicate `infinite`, a variation of which was denoted by $\Box$ in the examples of coalgebraic specifications by Jacobs [34]. It is notable that M. Niqui, who suggested a similar formalisation of filter functions for streams, used a predicate `is_infinite?` in order to extract the desired guarded filter, see [37].

We suggest to define `infinite` predicates co-inductively by expressing that a value satisfies this predicate if it satisfies the `eventually` predicate and the second part computed by the inductive component satisfies the `infinite` predicate again. Remember that this second part is the argument for the next recursive call of the main corecursive function.

Definition 6.1 For the `dyn` function, we choose to call the `infinite` predicate `infinite_dyn`. It is described as follows:

```
CoInductive infinite_dyn (x : B): Prop :=
  di : forall d: eventually_dyn x,
   infinite_dyn (snd (pre_dyn x d)) -> infinite_dyn x.
```

For the `filter_s` function, we choose to call the predicate `infinite_s`:

```
CoInductive infinite_s : str ->  Prop :=
  al_cons: forall (s:str A) (h: eventually s),
   infinite_s (snd(pre_filter_s s h)) ->  infinite_s s.
```

For the `e_filter` function we choose to call the predicate `infinite_t`:

```
CoInductive infinite_t : ETrees A1 B1 -> Prop :=
  cf: forall  (t:ETrees A1 B1) (h:eventually_t t),
    infinite_t (snd (pre_filter_t t h)) -> infinite_t t.
```

Notice that the shape of `infinite` predicates for all these functions is almost identical.

The `infinite` predicate describes exactly those arguments to the function for which the function is guaranteed to be productive. Thus, we will reproduce the scheme already followed in recursion with respect to an ad-hoc predicate: the recursive function will be modelled in Coq by a function that receives an extra argument, a proof that the initial argument satisfies the `infinite` predicate. Computation will be performed by repeatedly calling the inductive component of the function. Since the inductive component function can only compute on arguments that satisfy the `eventually` predicate, we need a lemma stating that the `infinite` predicate implies the `eventually` predicate. This lemma is always obtained using a very simple proof by pattern matching.

For the `dyn` and `filter_t` functions the lemma has the following statements:

Lemma 6.2

```
Lemma infinite_eventually_dyn :
  forall x, infinite_dyn x -> eventually_dyn x.
```

```
Lemma infinite_eventually_t :
  forall t, infinite_t t -> eventually_t t.
```

After the computation of the first component, computation must go on, with recursive calls obtained from the second part of the data computed in the inductive component. However, this data must also be associated with a proof that it satisfies the `infinite` predicate. This is expressed by means of the following lemmas, given for `dyn` and `e_filter`:

Lemma 6.3

```
Lemma infinite_always_dyn :
  forall x, infinite_dyn x -> forall e: eventually_dyn x,
  infinite_dyn (snd (pre_dyn x e)).
```

```
Lemma infinite_always_t : forall(t : ETrees A1 B1)(h:infinite_t t),
     infinite_t (snd (pre_filter_t t (infinite_eventually_t t h))).
```

We give the proofs of these lemmas in [13].

Finally, we use these two categories of lemmas to formalise the main functions from Definitions 3.5, 3.8 as guarded corecursive functions.

Definition 6.4 For the `dyn` function the definition has the following shape (proof arguments are boxed):

```
CoFixpoint dyn (x:B) (h:infinite_dyn x) : str :=
SCons (fst (pre_dyn x (infinite_eventually_dyn ev x h) ))
 (dyn _ (infinite_always_dyn x h (infinite_eventually_dyn x h)) ).
```

For the `e_filter` function, the definition has the following shape:

```
CoFixpoint e_filter (t:ETrees A1 B1)(h: infinite_t t)
  : ETrees A2 B2 :=
match t return infinite_t t -> ETrees A2 B2 with
| A_node x t' => fun h' : infinite_t (A_node x t') =>
    A_node (fst (pre_filter_t t (infinite_eventually_t h')))
           (e_filter _ (infinite_always h'))
| B_node x t' t2 => fun h' : infinite_t (B_node x t' t2) =>
    A_node (fst (pre_filter_t t (infinite_eventually_t h')))
           (e_filter _ (infinite_always h'))
```

```
end.
```

The guardedness of the functions is simply achieved by the presence of constructors `SCons` or `A_node`. The recursive calls receive the extra proof arguments. Notice that we don't need to give explicitly the value on which the recursive call is performed, because this value can be inferred from the second proof argument. This explains why we used the anonymous value place holder `_` in several places.

We formalise the filter from Definition 3.6 in [13].

In this Section, we have given a systematic characterisation of the coinductive components of productive functions. We employed the predicate `infinite` to obtain the guarded formalisation of productive functions in Coq. It remains to provide tools to reason on the functions we modelled.

7 Proving Properties About the Models

In this section, we give the proofs of *recursive equation lemmas*, statements that express that the functions we obtain in Definition 6.4 really are good models of the non-guarded functions from Definitions 3.5, 3.6, 3.8. We discuss how such proofs can be obtained in a systematic way, outlining a few useful lemmas about these functions.

The inductive component and coinductive component for each corecursive function are modelled by Coq functions which take both a regular argument (named `x` in the function `dyn` and `t` in the function `e_filter`) and a proof about this argument (named `h` in both `dyn` and `e_filter`). Of course, we expect the result to depend on the regular argument, but we do not expect the computation to depend on the value of the proof. We only expect the computation to depend on the existence of this proof. In other words, when receiving a given argument `x` and two different proofs `e1` and `e2` that `x` satisfies the `infinite_dyn` predicate, we expect the resulting values `dyn x e1` and `dyn x e2` to be the same. This needs to be expressed in lemmas which we call *proof-irrelevance* lemmas.

Lemma 7.1 (Proof-irrelevance Lemmas for `dyn`) .
For the `dyn` *function the proof irrelevance lemmas have the following shape:*

```
Lemma pre_dyn_prf_irrelevant :
 forall x e1 e2, pre_dyn x e1 = pre_dyn x e2.

Lemma dyn_prf_irrelevant :
  forall x x' (i: infinite_dyn x) (i':infinite_dyn x'), x = x' ->
  bisimilar_s (dyn x i) (dyn x' i').
```

The proofs of these lemmas are given in [13]. Also, in [13] we formulate and prove the similar proof irrelevance lemmas for filters on streams and expression trees. A similar proof irrelevance lemma was used in [38]; a very good study of applications of the Principle of Proof Irrelevance in type theory can be found in [8].

Note that the above proof irrelevance lemma, as well as analogous lemmas for filters on streams and trees [13], uses the "bisimilar" relation from Definition 2.6 instead of the equality relation.

In coalgebra, one has the *coinductive proof principle* which states that bisimilar objects are equivalent, [35]. The constructive theory distinguishes equality and bisimilarity; and in many cases we can only give constructive proofs of bisimilarity of infinite objects; but not of their equality.

Nevertheless, for all practical purposes, bisimilar objects can be used like equal objects. For example, one can prove in Coq that equality implies bisimilarity; or that for finite objects bisimilarity implies equality. Lemmas and theorems relating bisimilarity and equality can be found in [12].

It remains to show that the functions `dyn`, `filter` and `e_filter` modelled in Coq actually perform the computation that was initially intended. The main idea is to perform as in [4,5] and to prove a recursive equation. However, we have to cope with the last difficulty. The models we obtain have an extra argument, which is used to restrict the function to the inputs on which they really are productive.

We circumvent the difficulty by integrating extra proof arguments in the formulation of the recursive equation. As a consequence, we need a few more lemmas expressing that all recursive calls that happen in the recursive equation are made on values that satisfy the `infinite` predicate, as soon as the initial argument already satisfies this predicate. These lemmas will be called *step-lemmas*.

Example 7.2 For instance, the specification of `dyn` imposes that there should be a recursive call `dyn (g' x)` as soon as `P x = false`. Thus, we need to have a lemma that states that `(g' x)` satisfies `infinite_dyn` as soon as `x` does and `P x = false`.

Lemma 7.3 (Step-lemmas for `dyn`) .
For the function `dyn`*, there are two step lemmas:*

```
Lemma dyn_step1 :
  forall x, P x = true -> infinite_dyn x -> infinite_dyn (g x).

Lemma dyn_step2 :
  forall x, P x = false -> infinite_dyn x -> infinite_dyn (g' x).
```

We formalise the proofs for these lemmas in [13]. The first lemma expresses that the recursive call on `g x` is legitimate when `P x` is true, while the second lemma expresses that the recursive call on `g' x` is legitimate when `P x` is false.

We prove the similar step-lemmas for filters on expression trees and streams in [13]. They are all formulated uniformly with the step-lemmas above. For expression trees, for example, we need four step lemmas, to give an account both for the cases when `P x` is true or false, and for the two constructors used to define expression trees. Conceptually, all these lemmas can be formulated in a uniform way.

We can then formulate the recursive equation for the functions we modelled. This recursive equation is not as easy to read as the initial description, because proof arguments have been added in many places, but if we ignore these proof arguments, the initial intent appears clearly.

For the `dyn` function, the equation lemma has the following shape, where bisimilarity is again used instead of equality. To help the reader to distinguish proof content from algorithmic content, we boxed the proof arguments.

Theorem 7.4 (Recursive Equation Lemma for `dyn`) .

```
Theorem dyn_equation :
  forall x [i], bisimilar_s (dyn x [i])
    (match P x [as b return P x = b -> infinite_dyn x -> str] with
    | true => [fun t i] => SCons (h x) (dyn (g x) [(dyn_step1 x t i)])
    | false => [fun t i =>] dyn (g' x) [(dyn_step2 x t i)]
    end [(refl_equal (P x)) i]).
```

We prove this lemma in [13], and give similar proofs for `e_filter` and `filter`.

In [10], recursive equations were not given or proved: the properties of interest for the filter output were directly stated and proved, culminating with the proof that the infinite stream of prime integers could be obtained by repetitively filtering from the infinite stream of natural numbers. In this paper, we take a more systematic approach, with the hope that all steps can be automated.

8 Conclusions and Further Work

In this paper, we revisited the technique developed in [10] to model filter functions on streams in a type-theory with coinductive types and guarded corecursion. We showed that the same technique could be applied to a wide class of functions: firstly, we showed that it could be applied even if the output data-type was not a stream type, and secondly, we showed that the input data does not have to be of coinductive type. In the process, we delineated the various steps of the description and we showed that the technique can lead to the main theorem stating that the model exhibits the expected behaviour.

In practice, this work extends the expressive power of guarded co-recursion by showing that some class of totally and partially productive corecursive functions can be modelled, even though their initial specification would be expressed in a non-guarded corecursive equation. The method presented in this paper does not cover the productive functions that fail to satisfy the guardedness condition ** of Section 4. Such functions were studied (e.g.) in [1,26]. Combining these methods with the method we have described here can be one of the objectives for the future work.

Many of the steps in the technique we describe here can easily be automated, others are very tricky to formulate. In future work, we wish to give a precise description of the class of functions for which the technique works, and a precise description of each step that is taken in producing the `eventually` predicate, the first recursive component using recursion on the ad-hoc predicate `eventually`, etc. In the end, all the steps could be implemented as a program that takes as input the syntax tree describing the function one wants to model and produces both the Coq model and the main lemmas about this model.

Another extension of this work is to study what can be done for functions that are *provably* totally productive (i.e., productive on every input) but yet not guarded. In this case, the final recursive equation should be expressible without proof components. This extension can probably take inspiration from the work done on well-founded induction.

References

[1] A. Abel. *Type-Based Termination. A Polymorphic Lambda-Calculus with Sized Higher-Order Types.* PhD thesis, Fakültat für Mathematik, Informatik und Statistik der Ludwig-Maximilians-Universität München, 2006.

[2] P. Aczel. An introduction to inductive definitions. In *Handbook of Mathematical Logic*, pages 739 – 782. North-Holland Publishing Company, 1977.

[3] P. Aczel. Algebras and coalgebras. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, pages 79–88, 2000.

[4] A. Balaa and Y. Bertot. Fixpoint equations for well-founded recursion in type theory. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher-Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *LNCS*, pages 1–16. Springer-Verlag, 2000.

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

[6] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 117–309. Oxford University Press, 1992.

[7] H. Barendregt and H. Geuvers. Proof-checking using dependent type systems. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume II, chapter 18, pages 1149–1240. Elsevier Science, 2001.

[8] G. Barthe. The relevance of proof irrelevance. In *Authomata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 755–768. Springer Verlag, 1998.

[9] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14:97–141, Feb. 2004.

[10] Y. Bertot. Filters and co-inductive streams, an application to eratosthenes' sieve. In *TLCA*, volume 3461 of *LNCS*, pages 102 – 115. Springer Verlag, 2005.

[11] Y. Bertot. Affine functions and series with co-inductive real numbers. *Mathematical Structures in Computer Science*, 17(1):37–63, 2007.

[12] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Constructions.* Springer-Verlag, 2004.

[13] Y. Bertot and E. Komendantskaya. Experiments towards general co-recursion: Coq code, 2008. http://hal.inria.fr/inria-00277075/en/.

[14] A. Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, 2001.

[15] A. Bove. *General Recursion in Type Theory.* PhD thesis, Department of Computing Science, Chalmers University of Technology, 2002.

[16] A. Bove. General recursion in type theory. In *TYPES*, pages 39–58, 2002.

[17] A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In *TPHOLs*, pages 121–135, 2001.

[18] A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15(4):671–708, 2005.

[19] W. Buchholz. A term calculus for (co-)recursive definitions on streamlike data structures. *Ann. Pure Appl. Logic*, 136(1-2):75–90, 2005.

[20] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2), 2005.

[21] T. Coquand. Pattern-matching in type theory. In *Informal Proceedings of the 1992 workshop on Types for Proofs and Programs*, pages 71–84. B. Nordström and K. Petersson and G. Plotkin, 1992.

[22] T. Coquand. Infinite objects in type theory. In *Types for Proofs and Programs, Int. Workshop TYPES'93*, volume 806 of *LNCS*, pages 62–78. Springer-Verlag, 1994.

[23] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76, 1988.

[24] E. Dijkstra. On the productivity of recursive definitions, 1980. EWD749.

[25] A. Edalat and J. Potts. A new representation for exact real numbers. In *MFPS XIII*, volume 6 of *ENTCS*. Elsevier, 1997.

[26] J. Endrullis, C. Grabmayer, D. Hendriks, A. Isihara, and J. W. Klop. Productivity of stream definitions. In *FCT*, pages 274–287, 2007.

[27] H. Geuvers. *Logics and Type Systems*. PhD thesis, University of Nijmegen, 1993.

[28] P. D. Gianantonio. *A Functional Approach to Computability on Real Numbers*. PhD thesis, Universita di Pisa-Genova-Udine, 1993.

[29] P. D. Gianantonio and M. Miculan. A unifying approach to recursive and co-recursive definitions. In *TYPES*, pages 148–161, 2002.

[30] P. D. Gianantonio and M. Miculan. Unifying recursive and co-recursive definitions in sheaf categories. In *FoSSaCS*, pages 136–150, 2004.

[31] E. Gimenez. *Un Calcul de Constructions Infinies et son Application a la Verification des Systemes Communcants*. PhD thesis, Laboratoire de l'Informatique du Parallelism, Ecole Normale Superiere de Lyon, 1996.

[32] W. Howard. The formulae-as-types notion of construction. In J. Selding and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[33] G. Huet. Induction principles formalised in calculus of constructions. In J. Seldin and J. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lmabda Calculus, and formalism*, pages 479–490. Academic Press, 1988.

[34] B. Jacobs. Exercises in coalgebraic specification. In R. Backhouse, R. Crole, and J. Gibbons, editors, *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, number 2297 in LNCS, pages 237–280. Springer, 2002.

[35] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222 – 259, 1997.

[36] J. Matthews. Recursive function definition over coinductive types. In *TPHOLs*, pages 73–90, 1999.

[37] M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. PhD thesis, Radbout Universiteit Nijmegen, 2004.

[38] M. Niqui. Coinductive field of exact real numbers and general corecursion. In *Proc. of CMCS'06*, volume 164 of *ENTCS*, pages 121–139. ELSEVIER, 2006.

[39] B. Nordstöm. Terminating general recursion. *BIT*, 28(3):605 – 619, 1988.

[40] C. Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In *TLCA*, pages 328–345, 1993.

[41] L. Paulson. Mechanizing coinduction and corecursion in higher-order logic. *Logic and Computation*, 2(7):175 – 204, 1997.

[42] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, (31):45–60, 1981.

[43] B. Sijtsma. On the productivity of recursive list definitions. *ACM Transactions on Programing Languages and Systems*, 11(4):633 – 649, 1989.

[44] The Coq Development Team. The coq proof assistant reference manual. http://coq.inria.fr.