



# Filters on Co-Inductive streams: an application to Eratosthenes' sieve

Yves Bertot

## ► To cite this version:

Yves Bertot. Filters on Co-Inductive streams: an application to Eratosthenes' sieve. RR-5343, INRIA. 2004, pp.21. inria-00070658

**HAL Id: inria-00070658**

**<https://inria.hal.science/inria-00070658>**

Submitted on 19 May 2006

**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.

# *Filters on Co-Inductive streams: an application to Eratosthenes' sieve*

Yves Bertot

**N° 5343**

Octobre 2004

Thème SYM



*rapport  
de recherche*



## Filters on Co-Inductive streams: an application to Eratosthenes' sieve

Yves Bertot

Thème SYM — Systèmes symboliques  
Projet Lemme

Rapport de recherche n° 5343 — Octobre 2004 — 21 pages

**Abstract:** We show how to model filter functions on the co-inductive types of infinite streams in type theory. These functions are partial but the theory imposes total functions. Our solution relies on describing a predicate characterizing the definition domain of filter functions, with a combination of inductive and co-inductive aspects.

**Key-words:** Formal methods, Calculus of Inductive Constructions, Co-induction, Infinite data-structures, Prime numbers

# Filtres sur les flux co-inductifs: une application au crible d'Eratostène

**Résumé :** Nous montrons comment modéliser des fonctions de filtres sur le type co-inductif des flux de données infinis en théorie des types. Le problème particulier est que ces fonctions sont partielles et que le cadre théorique impose des fonctions totales. Notre solution repose sur la description d'un prédicat qui caractérise le domaine de définition d'une fonction de filtre et la combinaison d'aspects inductifs et co-inductifs dans ce prédicat.

**Mots-clés :** Méthodes formelles, Calcul des constructions inductives, Co-induction, structures de données infinies, Nombres premiers

Our objective is to describe a formal proof of correctness for the following Haskell [14] program in a type theory-based proof verification system, such as the Coq system [11, 2].

```
sieve (p:rest) = p:sieve [r | r <- rest, r `rem` p /= 0]
primes = sieve [2..]
```

This program is a functional implementation of Eratosthenes' sieve that consists in removing all multiples of previously found primes from the sequence of natural numbers. We want to prove that the expression `primes` is the stream containing all the prime numbers in increasing order.

This work relies on co-inductive types [6, 12, 13] because the program manipulates infinite lists, also known as streams. It first uses the infinite list of natural numbers larger than 2, then the infinite list of numbers larger than 3 and containing no multiples of 2, then the infinite list of numbers larger than 4 and containing no multiples of 2 and 3, and so on, to eventually produce the infinite list of prime numbers. This example was initially proposed as a challenge by G. Kahn and used as an illustration of a program and its proof of correctness in a language for co-routines in [15]. The exact formulation of the program given here was found in A. J. T. Davie's book on functional programming [8], who describes it as a re-phrasing of an initial program implemented in SASL by D. Turner [21]. A proof of Eratosthenes' sieve in type theory was already studied by F. Leclerc and C. Paulin-Mohring in 1993 [16], but their program has a different structure and does not exhibit the filter problem that is central here. In particular, their program does not return a list of all primes, but a list of all integers where the primes are tagged as such. Another program computing the stream of prime numbers is given as example in [9]. The technique advocated in that work relies on a more general notion of ultra-metric spaces to combine inductive and co-inductive aspects in recursive definitions. It is later extended to sheaves [10].

Before performing the proof of such a program in type theory, we need to be able to formulate it. In this case the problem is difficult, because the program does not fit directly in the restricted programming language that is provided in the Calculus of Inductive Constructions. The Haskell program uses a filter function. This function receives a boolean predicate and a stream as argument and it is supposed to take from the stream all the elements of the stream that satisfy the predicate and place them in the resulting stream. For arbitrary streams, such a filter function is not guaranteed to produce even one number, and it is in general undecidable whether an infinity of numbers will really be produced.

When formalizing co-inductive structures in theorem provers, the style of recursion that is allowed is restricted in such a way that computation always terminates. For instance, if we want to get the first integer from an infinite stream of integers, then this computation is guaranteed to terminate. Since the productivity of a filter function is not decidable, this means that filter functions cannot be programmed directly. Our principal contribution is actually to find a solution to the problem of programming filters, and more generally, a large class of partial co-recursive functions.

When computing on usual inductive structures, the termination of computation is usually ensured by a syntactic restriction on the way functions may be defined: they have to

be “guarded-by-destructors”. Intuitively, this constraint impose a bound on the number of possible recursive calls using the size of the algebraic term given as input. In spite of its apparent simplicity, this criterion is quite powerful, because inductive types are more general than simple algebraic types: infinitely branching nodes are allowed and it is only the absence of infinite branches that is used to restrict computation.

For functions that produce terms in co-inductive types, recursive functions are also allowed, but this time restrictions are not placed on the way the input is used, but on the way the output data is produced. A common syntactic criterion is to accept a recursive call to a co-recursive function is only if some information is produced in the result, in the form of a constructor . We say that co-recursive calls must be “guarded-by-constructors”. In their usual form, filters do not respect this syntactic criterion.

The computation on usual inductive types is carried throughout so that no application of a function to an argument remains in the result value. When computing values in a co-inductive type, the computation is by default not performed: it is only when a fragment of information is requested that the computation is triggered, and it stops as soon as enough information is provided. This corresponds to the usual notion of *call-by-need* or *lazy computation*. The interaction of recursive functions (with inductive arguments) and co-recursive functions is consistent and still ensures the termination of all computations: a recursive function is guaranteed to terminate in finite time, so it can only request finite information from a co-inductive expression, and this finite information is produced in only finite recursive calls of co-recursive functions.

To program filters on streams, we propose to combine insights coming from reasoning techniques on linear temporal logic [5, 7] and on general recursion, essentially the technique advocated by A. Bove [3] in the context of Martin-Löf type theory. We transpose this technique to the Calculus of Inductive Constructions, the underlying theory for the Coq system, with some added difficulties coming from the use of two sorts. Coping with these two sorts also has advantages; we obtain the possibility to extract our model back to conventional programming languages and to execute the programs that were proved correct.

Here is the plan of this paper. In a first part we give a rapid overview of recursive programming techniques in the Calculus of Inductive Constructions. In a second part we show that filters cannot be programmed directly using these techniques, mainly because not all streams are valid inputs for filters, and we describe a few notions of linear temporal logic that make it possible to characterize the valid inputs. We show that the linear temporal logic predicates can be used as a basis to program a filter function. In a third part, we describe how this adapts to the context of Eratosthene’s sieve. The fourth part brings concluding remarks and underlines the opportunities for future improvement.

The formal proofs are available on internet at the following address:  
<ftp://ftp-sop.inria.fr/lemme/Yves.Bertot/filters.tar.gz>

# 1 Recursion in the calculus of inductive constructions

Proof verification systems have strong requirements of internal consistency. One of the basic requirements is that all the values should be well-defined. General recursion may lead to infinite computation and this implies that the values of functions applied to arguments may not be well-defined. Therefore, many proof calculi, among which the calculus of inductive constructions, restrict recursive computation to guarantee the termination of all computations. In the calculus of inductive constructions, two different techniques are used, depending on whether we are working with inductive types or co-inductive types.

## 1.1 Inductive types and recursion

### 1.1.1 Basic recursion

In the calculus of inductive constructions, an inductive type is described by giving simultaneously the type and basic functions that can be used to construct values of this type, the *constructors*. Here is an example of an inductive type, describing binary trees where all internal nodes are labelled with natural numbers (of type `nat`).

```
Inductive tree : Set :=
  leaf : tree | node : nat -> tree -> tree -> tree.
```

This definition expresses that we can use the constant `leaf` and the function `node` to construct values of type `tree`. The consistency of the calculus ensures that every term of type `tree` can be reduced to an expression of finite size containing only natural numbers and occurrences of the functions `leaf` and `node`. The constructor `node` exhibits some form of recursion since two of its arguments may themselves be trees.

When programming a recursive function on trees, the framework only accepts recursive definitions that ensure that the computation will terminate, by imposing that recursive calls are always performed on strict sub-parts of the initial argument. For instance, here is the definition of a function that takes a tree and adds the numbers in this tree:

```
Fixpoint sum_tree (t:tree) : nat :=
  match t with
  | leaf => 0
  | node n t1 t2 => n + sum_tree t1 + sum_tree t2
  end.
```

This definition uses pattern-matching: the two cases where the input is a leaf or a node are described as patterns and the values for these patterns are described on the right-hand side of the arrow sign. When the tree is a node, the result combines the node's label `n` and the results of recursive calls on the sub-parts `t1` and `t2`.

Recursive calls are only allowed on sub-terms of the initial argument, a criterion that can be verified by analyzing the definition's structure. Sub-terms are simply described as the variables appearing in patterns when the value being matched is the initial argument.



In the function `sum_tree` the initial argument is `t`, there is one pattern-matching construct, and in the second rule the variables `t1` and `t2` are obtained in the pattern. Thus, these variables are sub-terms of `t` and the recursive calls are valid.

Actually, the guard system is more general than what is hinted in the previous paragraph. Whole expressions, not just variables, can be accepted as sub-terms, as long as these expressions reduce to pattern-matching constructs where all leaves are sub-terms. Seeing an example of this extension requires understanding other complications: dependent types and dependent pattern-matching.

### 1.1.2 Dependent types

Dependent types make it possible to construct functions that combine some result and some information about the result. For instance, a boolean function on trees to test whether its argument is a leaf or not could be described with the following type:

```
leaf_dec : ∀t:tree, {t=leaf}+{t<>leaf}
```

The value returned by `leaf_dec` on a given tree is either of the form `left h` where `h` is a proof that `t` is a leaf or of the form `right h'`, where `h'` is a proof of that `t` is not a leaf.

A proof of `t<>leaf` is a function that takes a proof of `t=leaf` as argument and returns a proof of `False`. The proposition `False` is given by an inductive definition:

```
Inductive False : Prop := .
```

This inductive definition looks odd: it has no constructors. Actually this is quite natural: the absence of constructors asserts that there are no proofs of `False` in the empty context (i.e., in the absence of any assumption). However, one may find proofs of `False` in some contexts, the contexts that have accumulated enough assumptions to describe inconsistent situations. These proofs of `False` are very useful. If we have the objective of producing of value of type `A` and we have a proof `h` of `False`, we can simply exhibit the following term:

```
match h return A with end
```

A pattern-matching construct is well formed if it contains a rule for every constructor of the type being matched. Here `h` has type `False`, and `False` has no constructors, so that no rule is needed. Pattern matching constructs can be annotated with an indication `return exp` to express what is the type of the whole expression. The term `exp` is modified in each branch of the construct to adapt to the specific pattern. Thus, each branch actually may actually have a different type, depending on the rule's pattern. This is *dependent pattern-matching*; it is not directly apparent when matching a value of type `False`, but we will use it shortly.

A pattern-matching construct on a proof of `False` is the central component of a function called `absurd_set`, provided in the Coq libraries with the following dependent type:

```
absurd_set : forall (A:Prop)(C:Set), A -> not A -> C
```

Given a proof of a property and a proof of its negation, this returns a value of any type.

### 1.1.3 Advanced guards

We will now describe a partial function, that is intended to receive as argument a tree that is not a leaf and returns the first sub-tree of this tree. To express that this function is partial, we give it a second argument, a proof that the first argument is not a leaf:

```
Definition protect_first (t:tree) : t<>leaf -> tree :=
  match t return t<>leaf -> tree with
  | leaf =>
    fun h : leaf<>leaf => absurd_set h (refl_equal leaf)
  | node n t1 t2 =>
    fun h : node n t1 t2 <> leaf => t1
  end.
```

The expression `refl_equal leaf` is a proof of `leaf=leaf`. The dependent pattern-matching construct on `t` expresses that when `t` is a term of the `leaf` form, we can use the assumption `leaf<>leaf`. With these two terms we can obtain `False` and produce an expression of type `tree`. We already said that `absurd_set` basically contains a pattern-matching construct on a false value. Since such a pattern-matching construct contains no branches, it is basically as a sub-term of `t` from the point-of-view of the guard conditions. In the other case, the assumption `t<>leaf` is dependently re-phrased in `node n t1 t2<>leaf` and the value that is returned is `t1`, obviously a sub-term because this is a variable in a pattern on `t`.

As a result, in any context where we have the assumption `h` of type `t<>leaf`, the expression `protect_first t h` will be accepted as a guarded sub-term of `t`. Using the same technique, we can write a function `protect_second` and a function `protect_label`. We can then re-phrase the sum function:

```
Fixpoint sum_tree' (t:tree) : nat :=
  match leaf_dec t with
  | left h => 0
  | right h' => protect_label t h' +
    sum_tree' (protect_first t h') +
    sum_tree' (protect_second t h')
  end.
```

### 1.1.4 The distinction between Prop and Set

The calculus of inductive constructions as implemented in the Coq system is a two-sorted system. This is useful to let the user annotate with the sort `Prop` the information that is only logical and should not be re-computed in efficient implementations of an algorithm. On the other hand, the information that has some computational content is usually annotated with `Set`; this information should be re-computed every time. This distinction between logical and computational information must be consistent. It is not possible to write algorithms where some information is logical but is used to determine the value of some information that

is computational. Still, functions may have both computational and logical arguments to produce computational arguments. The intuition is that the logical arguments are only used to produce logical annotations attached to computational values or to ensure the termination of recursive functions.

In the functions that we have described so far, the type of the function `leaf_dec` expresses that this function returns some computational data that has a boolean nature (the form of the result is either `left` or `right`) and some logical data (when the form is `left`, there is a proof `h` that the function's input is a leaf). Other functions, like `protect_first` use logical information to avoid producing data in some cases. As a last example, the function `sum_tree` receives some logical information from one function and transmits it to the others and the whole logical information flow is useful to keep the recursion guard system happy.

The distinction between computational and logical information is given when defining new types, by saying that they belong in the sort `Prop` or in the sort `Set`. With these declarations, the Coq system also provides an *extraction* mechanism, that makes it possible to produce code in Ocaml, Scheme, or Haskell that will behave as predicted by the proofs performed in Coq, but where all computations concerning objects of sort `Prop` have been removed [20, 17]. It is thus relevant to make efforts to place some arguments in the sort `Prop` as it helps optimizing the code that is eventually produced.

### 1.1.5 Simple general recursion

Logical information can be used more directly than in `sum_tree` to control that recursive functions will always terminate. For recursive functions with several arguments, we can choose the argument on which the recursive constraints will be satisfied. For instance, a function can receive as initial argument a value `x` and a proof `h` that some property `P x` holds. The function may be recursive in `x`, so that recursive calls are only allowed on sub-terms of `x`. But if `P` is itself an inductive predicate, then the function may be recursive in `h` and in this case, recursive calls can occur on any `y` if there is a proof of `P y` that is a sub-term of the initial proof `h`, even if `y` is not a sub-term of `x`,

This technique has been known for a while [1, 18, 19] and is the basis for *well-founded recursion*. In well-founded recursion, a single inductive predicate, called *accessibility* is used to control the recursion of a wide variety of recursive functions. Given a type `A` and a binary relation on `A`, an element `x` of `A` is called *accessible* if all its predecessors for the relation `R` are accessible. This predicate is obviously inductive. simple examples of accessible elements are the elements that have no predecessor, then one can find the elements that have only elements that have no predecessors as predecessors, and so on. When given a proof that some `x` is accessible and a proof that some `y` is a predecessor of `x`, it is easy to construct a proof that `y` is accessible that can be accepted as a sub-term of the initial accessibility proof. Using this, it is possible to write recursive functions that take as arguments an element of `A` and proof that this element is accessible, where it is only necessary to show that the recursive calls are done on predecessors of the initial argument for the relation `R`. A binary relation on some type such that all elements of the type are accessible is called *well-founded*. When a relation is well-founded, it is possible to write recursive functions with only one

argument  $x$ , where every recursive call must receive as extra argument a proof that the recursive argument is predecessor of  $x$  for the well-founded relation. This is the technique called *well-founded recursion*. For more details and examples, see [2].

The difficulty of proving that some relations were well-founded prompted A. Bove to come back to the initial idea of controlling the recursion of an algorithm by directly using an inductive predicate. It appears that one can associate an inductive predicate to every recursive function (there is difficulty for nested functions, see [4]). The recursive function is thus modelled by a function with an extra argument, intuitively understood as a proof that the argument belongs to the function's domain. We call this technique *recursion on an ad-hoc predicate* because the recursion is controlled by an inductive predicate that is defined especially for the function.

Recursion on an ad-hoc predicate is slightly more complex in the calculus of inductive constructions than in Martin-Löf's type theory because of the sort distinction: it is more difficult to produce sub-proofs in the calculus of inductive constructions, because the distinction between computational and logical information prevents us from performing case analysis on proofs in contexts where the expression is used as computational data.

A. Bove studied recursive functions over inductive types. We actually generalize the technique to functions producing values in co-inductive types.

## 1.2 Co-induction and co-recursion

Co-inductive types are defined like inductive types by giving together a type and a collection of constructors. However, there is no intuition that terms in a co-inductive type have only finite branches. Actually, infinite branches may be the rule rather than the exception.

For instance, we can work in a context where some type  $A$  is declared and use the type of streams of elements of  $A$ . In this paper, we later instantiate  $A$  with the type  $\text{nat}$  of natural numbers.

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

This definition expresses that a stream of type  $\text{str}$  is like a list of elements of  $A$ : it has a first element and a tail, grouped together using the constructor  $\text{SCons}$ . Inductive definitions of list usually contain a constructor for the empty list, but here there is no such constructor: all our streams are infinite.

Writing programs that use streams means that we have to be careful to avoid traversing the whole data structure, because this operation will never terminate. In the calculus of inductive constructions, there are a few safeguard that prevent this. The first safeguard is that computation of values in co-inductive types is not performed unless explicitly requested. The second safeguard is that the definitions of recursive functions returning co-inductive types must respect a few guarding constraints, like for recursive function over inductive types, except that the guards are not expressed in terms of using the input but in terms of producing the output. The intuitive motto is “every recursive call must produce some information”. In practice, every recursive call must be embedded in a constructor of the co-

inductive type, the whole expression being allowed to appear only inside a pattern-matching construct, an abstraction, or another constructor of this co-inductive type.

Here is an example of a well-formed function, that will be used in our work (in this example, the type `A` is not used implicitly anymore and we need to use explicitly the type of streams of natural numbers, written `str nat`).

```
CoFixpoint nums (n:nat) : str nat := SCons n (nums (n+1))
```

This function does not use the fact that its input is in an inductive type. Every recursive call produces a new element of the stream. The value “`nums 2`” is exactly the model for the Haskell value `[2 ..]`.

Proofs by co-induction are proofs whose statement concludes on a co-inductive predicate, that is a co-inductive type with logical content (a co-inductive type in sort `Prop`). When performing a proof by co-induction, we have the same constraints as when defining a co-recursive function: the co-inductive hypothesis expresses the same logical content as the whole theorem, but it can only be used to prove a statement appearing as a premise of one of the constructors in the co-inductive predicate.

Recursive functions can have several arguments, where some arguments in sort `Prop` have been added as logical information to control the recursion. This makes it possible to implement more general forms of recursion. For co-recursive function, there is only one result and the co-recursive guard cannot be pushed aside on a secondary result. To obtain more general forms of recursion, we shall again use secondary arguments to the function.

## 2 The filter problem

A filter function is a function that takes a predicate and a stream as arguments, and returns the stream that contains all the elements of the argument that satisfy the predicate. It can be programmed in Haskell using the following text:

```
filter f (x:tl) | f x = x:(filter f tl)
filter f (x:tl)      = (filter f tl)
```

The notation `[x | x <- rest, x ‘rem’ p /= 0]` actually stands for the following more traditional functional expression:

```
filter (\x -> x ‘rem’ p /= 0) rest
```

When translated into Coq, this gives the following (invalid) code:

```
CoFixpoint filter (f:A->Prop)(s:str) : str :=
  match s with
  | SCons x tl =>
    match f x with
    | true => SCons x (filter f tl)
    | false => filter f tl
  | end.
```

There are more palatable equivalent notations, but we used this formulation to emphasize the fact that the second recursive call appearing in this program is not valid: it is a recursive call not embedded inside a constructor. We need to be able to perform several recursive calls before returning the next data, and we don't know how far in the stream we may need to search before finding it, but this is rejected. This is consistent with the constraint that there should be no infinite computation: if we take a predicate and a stream where no element satisfies the predicate, the program will loop forever without producing any result. Understanding this counter-example will give us a key to a technique to model filter functions.

## 2.1 Characterizing valid filter inputs

If we want to use a filter function, we need to give it arguments that won't make it loop. We use the same technique as for recursion on an ad-hoc predicate: an extra argument expresses that the input satisfies the right conditions to ensure data production.

For a given predicate  $P$ , a stream is correct if we can find an element of the stream that satisfies the predicate and if the sub-stream starting after that element is also correct. We can simplify this analysis by saying that a stream is correct if we can find an element satisfying the predicate and if its tail is also correct.

That there is one element satisfying the predicate actually is an inductive property, not a co-inductive one, so we will characterize the correct streams for a given predicate using both an inductive predicate on streams and a co-inductive predicate on streams.

This is reminiscent of linear temporal logic, viewing the different elements of the stream as a succession of states in time. The property that the predicate is eventually satisfied means that the property `eventually P` is satisfied. The property that must be repeated for all streams is an `always (eventually P)`. Castéran and Rouillard [5] and Coupet-Grimal [7] have already studied how these linear logic predicates can be encoded as inductive predicates. In our case, we assume that we are working in a context where the predicate  $P$  is given, and we encode directly the combination of `always` and `eventually` as a predicate on streams, which we call `F_infinite` (the predicates `always`, `eventually` and `F_infinite` are similar to the ones with the same name in [2], except that our predicates are parameterized by a property on stream elements instead of a property on streams).

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

```
CoInductive always : str -> Prop :=
  as_cons:
    forall x s, P x -> always s -> always (SCons x s).
```

```
CoInductive F_infinite : str -> Prop :=
```

```

al_cons:
  forall x s,
    eventually (SCons x s) -> F_infinite s ->
      F_infinite (SCons x s).

```

Now, a filter function should have the following type :

$$\forall s, F\_infinite\ s \rightarrow s$$

We have shown that characterizing the correct inputs for the filter function relies on both a co-inductive and an inductive part; this suggests that the filter function should have both a recursive part and a co-recursive part. The recursive part is responsible for finding the first element, making as many recursive calls as necessary without producing any data, but being guarded by a `eventually` property on the input, when the first element is found, we can have a co-recursive call, using the fact that we have now produced the required stream constructor to make the recursive call valid.

### 2.1.1 Programming the recursive part

The recursive part of the filter function is defined by recursion on the ad-hoc predicate `eventually`. It also uses a function `P_dec` that is supposed to compute whether the property `P` is satisfied or not.

Here is a first attempt where we only produce the first value that satisfies the predicate.

```

Fixpoint
pre_filter_i (s:str) (h:eventually s){struct h}:A :=
match s as b return s = b -> A with
  SCons x s' =>
    fun heq =>
      match P_dec x with
      left _ => x
      | right hn => pre_filter_i s'
                    (eventually_inv s h x s' heq hn)
      end
end (refl_equal s).

```

The theorem `eventually_inv` has the following statement:

$$\forall s, \text{eventually } s \rightarrow \forall x\ s',\ s = \text{SCons } x\ s' \rightarrow \text{not}(P\ x) \rightarrow \text{eventually } s'$$

Like the example `protect_first` in section 1.1.3, it is designed so that the expression `eventually_inv s h x s' heq hn` is a valid sub-term of `h`.

This first attempt is not satisfactory, because we also need the recursive function to produce the stream, on which filtering continues and together with a proof that this stream is also valid. Thus we want to program a function `filter_i` with the following type:

```
forall s,
  eventually s -> F_infinite s ->
    ({x:A, P x},{s':str, F_infinite s'})
```

This function takes one extra argument that is a proof that all the sub-streams eventually satisfy the predicate, it returns two pieces of data annotated with logical information. The first piece of data is a number  $x$  and the annotation is a proof that  $x$  satisfies the predicate  $P$ , the second piece of data is a stream  $s'$  and the annotation is a proof that an infinity of elements of  $s'$  satisfy  $P$ . We do not describe the code of `filter_i` here, it has the same structure as the function `pre_filter_i`, but it contains more code to gather the logical information.

### 2.1.2 Programming the co-recursive part

Assuming the `filter_i` function and a theorem `always_eventually`, which indicates that any stream that satisfies `F_infinite` also satisfies `eventually`, we can produce the `filter` function, which contains a single co-recursive calls using the data returned by `filter_i`.

```
CoFixpoint filter (s : str) (hs : F_infinite s) : str :=
  let (a, b) := filter_i s (always_eventually s hs) hs in
  let (n, hn) := a in let (s', hs') := b in
  SCons n (filter s' hs').
```

### 2.1.3 Proving properties of the result stream

Because the `filter` function has a recursive and a co-recursive part, all proofs about the resulting stream will have an inductive and a co-inductive part. For instance, we can prove that every property that is satisfied by all the elements of the initial stream is also satisfied by all the elements of the resulting stream. To state this theorem we have to change our implicit notations: the predicates `always` and `F_infinite` and the function `filter` are not implicitly applied to  $P$  anymore and we have to be more precise in our statement. This results in added arguments to the various predicates and functions.

```
Theorem filter_keep:
  ∀(P Q:A -> Prop) (P_dec : ∀x,{P x}+{not(P x)})
    (s:str) (h:F_infinite P s),
  always Q s -> always Q (filter P P_dec s h).
```

To establish this theorem, we first have to prove that the element and the stream returned by `filter_i` satisfy the properties  $Q$  and `always Q`, respectively. This proof uses an induction over a proof of `eventually P s`. The theorem has the following statement:

```
Theorem filter_i_keep:
  ∀(P Q:A -> Prop) (P_dec:∀x,{P x}+{not(P x)})
    (s:str)(h:eventually P s)(ha : F_infinite P s),
```



```

always Q s ->
∀x hx s' hs',
filter_i P P_dec s h ha =
(exist (fun n => P n) x hx,
  exist (fun s => F_infinite P s) s' hs') ->
Q x /\ always Q s'.

```

For this proof, we have to be careful and use a maximal induction principle instead of the one that is customarily used for proofs by induction over an inductive predicate, because some data actually depends on this inductive predicate. This maximal induction principle is obtained with the Coq command `Scheme`.

We can also prove that all elements of the resulting stream satisfy the predicate `P`.

```

Theorem filter_always:
  forall (s:str)(h:F_infinite s), always (filter s h).
Proof.
  cofix.
  intros s h; rewrite (st_dec_eq (filter s h)); simpl.
  case (filter_i s (always_eventually s h) h).
  intros [n hn][s' hs']; apply as_cons.
  assumption.
  apply filter_always.
Qed.

```

We give the script to perform the proof using the tactic language provided in Coq. The `cofix` tactic provides an assumption that expresses exactly the same statement as the theorem we want to prove, but this assumption can only be used after a use of `as_cons` (the constructor of `always`). Here we need to prove that the first element satisfies `P`, but this is already given in the result of `filter_i`, so that we do not need an extra inductive proof.

This proof contains a rewriting step with a theorem `st_dec_eq`. This theorem is used to force the evaluation of the co-inductive value `(filter s h)` because otherwise, co-inductive values remain unevaluated. This method to force evaluation for at least one step is described in [2], along with other techniques for proofs about co-inductive data.

#### 2.1.4 Non-local properties

If we only have the theorems `filter_always` and `filter_keep`, there are two important characteristics that are still missing. The first characteristic is that no value present in the input and satisfying `P` is forgotten. These characteristics seems more complex to express because they are not local properties of each stream element taken separately, but they are global properties of the streams. In fact, we have a solution to express them as properties between consecutive elements, using a new co-inductive predicate named `connected`. Intuitively, a stream is *connected* by some binary relation `R` with respect to some value `x` if any two consecutive elements of the stream are connected by `R` and the streams first element is connected with `x`. Here is the co-inductive definition:

```

CoInductive connected (R : A -> A -> Prop) : A -> str -> Prop :=
  connected_cons :
    forall k x s, R k x -> connected R x s ->
      connected R k (SCons x s).

```

For instance, to express that some stream contains all the natural numbers above a given  $k$  that satisfy the property  $P$  in increasing order, we can use the following binary relation:

```

Definition step_all_P x y :=
  x < y /\ forall z, x < z < y -> not(P z) /\ P y

```

and we say that the stream satisfies the property `connected step_all_P k`. This example already shows that the `connected` predicate will make it possible to express that some order is preserved and that no values are missing.

Actually our main theorem will simply express that the filter function maps any connected stream for a relation  $R_1$  to a connected stream for a relation  $R_2$ , provided the relations  $R_1$  and  $R_2$  satisfy the proper conditions with respect to  $P$ .

```

Theorem filter_connected :
  forall (R1 R2 : A -> A -> Prop),
    (forall x y z, R1 x y -> ~ P y -> R2 y z -> R2 x z) ->
    (forall x y, P y -> R1 x y -> R2 x y) ->
    forall s (h : F_infinite s) x,
      connected R1 x s -> connected R2 x (filter s h).

```

The two conditions that  $R_1$  and  $R_2$  must satisfy express that if  $x_1, \dots, x_k$  is a sequence of values such that  $P x_1$  and  $P x_k$  hold,  $\text{not}(P x_i)$  holds for all the other indices  $i$  from 1 and  $k$ , and  $R_1 x_i x_{i+1}$  holds, then  $R_2 x_1 x_k$  holds.

The theorem `filter_connected` is actually stronger than the two previous theorems. The theorem `filter_always` is a corollary for  $R_1$  the relation that is always satisfied and  $R_2$  the relation of  $x y$  that holds if and only if  $P y$  holds. The theorem `filter_keep Q` is a corollary for  $R_1$  and  $R_2$  that are both the relation of  $x$  and  $y$  that holds if and only if  $Q y$  holds.

We thus have a generic implementation of a filter function, together with a powerful generic theorem to prove its properties. This package can be re-used for any development using filters on arbitrary streams, as long as users provide the predicate, the decision function, and proofs that the streams taken as arguments satisfy the predicate infinitely many times. To perform proofs about the filtered streams, users simply need to exhibit the relations  $R_1$  and  $R_2$  and proofs of the properties they have to satisfy. Even though we used a clever technique to implement the filter function, it can be used and reasoned about with only the simple techniques of co-induction.

### 3 Application to Eratosthenes' sieve

We can now come back to our initial objective and use our filter function to model Eratosthenes' sieve.

#### 3.1 The sieve's specification

##### 3.1.1 Defining primality

Our model does not follow strictly the initial program in the sense that we change our filtering predicate for a predicate `not_mult m`, which accepts all numbers that are not multiples of `m` and we use a function `mult_dec` with the following type:

$$\forall m \ n, \{\text{mult } m \ n\} + \{\text{not}(\text{mult } m \ n)\}$$

We use an auxiliary notion of *partial primes*. We say that a number `n` is partially prime up to another number `m` if it is not a multiple of any number larger than 1 and smaller than `m` (the bounds are excluded). This notion is useful to characterize the streams that are given as arguments to the `filter` function, as we see later. We then define the notion of *pre-prime* numbers, which are partially prime up to themselves. The pre-prime numbers actually are 0, 1, and the prime numbers. We prove a few theorems around these notions:

**partial\_prime\_le** If a number is partially prime up to `m`, it is partial prime up to any `n` less than or equal to `m`.

**partial\_prime\_step** If a number is partially prime up to `m` and not a multiple of `m`, then it is partially prime up to `1 + m`.

**infinite\_primes** For every number, there exists a larger pre-prime number.

**pre\_prime\_decompose** For every number `n` that is not pre-prime, there exists a pre-prime divisor of `n` between 1 and `n` (bounds excluded).

**partial\_prime\_next** If a number is partially prime up to `m` and there are no pre-prime numbers between `m` and `n` (`m` included, `n` excluded), then it is partially prime up to `n`. The proof of this theorem uses the previous one.

##### 3.1.2 Specifications for input and output streams

Obviously the `connected` predicate is well-suited to express that some stream contains all the prime numbers above a given bound. We simply need to use the following binary relation:

**Definition** `step_prime x y :=`  
`x < y /\ (forall k, x < k < y -> not(pre_prime k))`  
 `/\ pre_prime y.`

The input must be a stream containing numbers that are partially prime, so we simply use the following relation to describe the specification of the input:

```
Definition step_partial_prime m x y :=
  x < y /\
  (forall k, x < k < y -> not(partial_prime k m))
  /\ partial_prime y m.
```

This gives us a different binary relation for different values of  $m$ .

### 3.1.3 Main theorems

We have two main theorems concerning the filter function. The first theorem expresses that the filter function can be used. We need to express that the right `F_infinite` property holds to use the filter function. This is expressed with the following theorem.

```
Theorem partial_primes_to_F_infinite:
  forall m, 1 < m ->
  forall k s, connected (step_partial_prime m) k s ->
  F_infinite (not_mult m) s.
```

The proof of this theorem relies on the basic theorem that there are infinitely many primes. It contains both a co-inductive step to prove that the stream tail also satisfies the `F_infinite` property and an inductive step to prove that we can find a number that is not a multiple of  $m$  in the stream. The inductive part of the proof is done by general induction over the distance to an arbitrary prime number above the first element of the stream, this distance is bound to decrease and stay positive as we traverse the stream while finding only multiples of  $m$ , because this arbitrary prime number is necessary in the stream and not a multiple of  $m$ .

The second theorem about the filter function shows that the function `filter (not_mult m) (mult_dec m)` maps any stream satisfying

```
connected (step_partial_prime m) k
to a stream satisfying
connected (step_partial_prime (S m)) k.
```

This theorem is not proved using any form of induction, we only need to check that the `step_partial_prime` relations satisfy the right conditions for the theorem `filter_connected`.

The other theorems concentrate on streams that are connected for the relations `step_partial_prime`. First, the theorem `partial_prime_next`, which we described in section 3.1.1, can be lifted to connected streams. Second, the first element of connected streams is a prime number.

```
Theorem pre_prime_connect_partial_prime:
  forall m s,
  connected (step_partial_prime (S m)) m s ->
  pre_prime (hd s).
```

### 3.2 Obtaining the main function

The streams that are manipulated in the main function are the streams of all partial primes up to  $m$ , starting at  $m$ . For this reason, we have defined another property that characterizes the main streams.

```
Definition start_partial_primes s :=
  1 < hd s /\
  connected (step_partial_prime (hd s)) (hd s) (tl s).
```

After filtering out the multiples of the stream's first element, we obtain a new stream where the first element  $p$  is itself prime and the rest is another stream that satisfies the property `start_partial_primes`. This property is the invariant that is respected by arguments to `sieve` throughout the recursion, this invariant is expressed in a theorem named `start_partial_primes_invariant`. With this invariant we can now define a Coq model for the sieve function.

```
CoFixpoint sieve s: start_partial_primes s -> str nat :=
  match s return start_partial_primes s -> str nat with
  SCons p rest =>
    fun H =>
      let (Hm, Hpprs) := H in
      let Ha :=
        partial_primes_to_F_infinite p Hm p rest Hpprs in
      SCons p
      (sieve
       (filter (not_mult p) (mult_dec p) rest Ha)
       (start_partial_primes_invariant p rest Ha Hm Hpprs))
  end.
```

Although this definition is cluttered with logical information, the reader should be convinced that this function really follows the same structure as the initial Haskell function we used as a guideline: construct a stream with the first element of the input and then call the `sieve` function on the result of filtering out the multiples of this first element.

We can then show that the resulting stream actually contains all the prime numbers above its first element. This is a simple proof by co-induction.

The last step is to verify that the stream of natural numbers starting from  $k$  is also the streams of partial primes up to 2 starting from  $k$  (this theorem is called `pprs2`) and we finally obtain the following theorem (`le_n 2` is a proof of  $2 \leq 2$ ).

```
Theorem pre_primes:
  connected step_prime 1
  (sieve (nums 2) (conj (le_n 2) (pprs2 2 (le_n 2))))).
```

## 4 Last part

Our first experiments was actually carried out on a more complex but very similar program, stated as follows.

```
fm a n (x:l) | n < x = fm a (n+a) (x:l)
fm a n (x:l) | n = x = fm a (n+a) l
fm a n (x:l) | x < n = x:(fm a n l)

sieve (x:l) = x:(sieve (fm x (x+x) l))

primes = sieve [2 ..]
```

The function `fm` actually performs the filter step of removing all the multiples of a number in a stream, but it avoids the computation of remainders by keeping the next expected multiple in an auxiliary variable. This function has an internal state and it does not behave in the same manner as `filter`. Actually, there are streams for which `filter (not_mult m) ...` behaves properly and `fm` does not, since this function relies more crucially on the property that all the values found in the streams are in increasing order. Defining this function and reasoning about it still relies on the same technique of mixing inductive and co-inductive predicates and developing an auxiliary function that is recursive on the inductive predicate. This work was done together with Damien Galliot as part of a student project in 2003.

The first result of this paper is to show that we can model more general recursive programming than what seems imposed by the basic “guarded-by-constructors” constraint. We believe this work describes an improvement on the domain of co-inductive reasoning that is similar to the improvement brought by well-founded recursion when compared to plain structural recursion. The key point was to adapt Ana Bove’s work on simple general recursion to co-inductive structures and this was made possible thanks to a remark by C. Paulin-Mohring that sub-terms were not restricted to variables.

The second important contribution is to describe a filter function in a general form, together with a general theorem that makes it possible to prove properties of this function’s result. While the function relies on a lot of expertise in the description of inductive and co-inductive programs, the general theorem makes it possible to relieve users from the task of performing inductive or co-inductive proofs, by simply coming back to relations between successive elements of the input and the output. We have shown the usability of our general theorem on the example of Eratosthenes’ sieve. It is interesting to compare our proof to the one proposed in [15]. Our proof only uses local notions: the properties of two consecutive elements in a stream, while their proof uses more general notions concerning whole streams.

## Acknowledgements

The author wishes to thank Pierre Castéran for teaching him the techniques of co-induction in the Calculus of Inductive Constructions and for his remarks on early drafts of this paper,

Gilles Kahn for discussions on the sieve example, Venanzio Capretta for sharing his knowledge on the technique of recursion on an ad-hoc predicate, Christine Paulin-Mohring for describing the extensions to the guard systems, and Laurence Rideau and Laurent Théry for their comments on early drafts of the paper.

## References

- [1] Peter Aczel. An introduction to inductive definitions. In J. Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*. North Holland, 1977.
- [2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art: the Calculus of Inductive Constructions*. Springer-Verlag, 2004.
- [3] Ana Bove. Simple general recursion in type theory. *Nordic Journal of Computing*, 8(1):22–42, 2001.
- [4] Ana Bove and Venanzio Capretta. Nested general recursion and partiality in type theory. In Richard J. Boulton and Paul B. Jackson, editors, *Theorem Proving in Higher Order Logics: 14th International Conference, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, September 2001.
- [5] Pierre Castéran and Davy Rouillard. Reasoning about parametrized automata. In *Proceedings, 8-th International Conference on Real-Time System*, volume 8, pages 107–119, 2000.
- [6] Thierry Coquand. Infinite objects in Type Theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 62–78. Springer Verlag, 1993.
- [7] Solange Coupet-Grimal. An axiomatization of linear temporal logic in the calculus of inductive constructions. *Journal of Logic and Computation*, 13(6):801–813, 2003.
- [8] Antony J. T. Davie. *An introduction to functional programming systems using Haskell*. Cambridge Computer Science texts. Cambridge University Press, 1992.
- [9] Pietro di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs*, volume 2646 of *LNCS*, pages 148–161. Springer Verlag, 2003.
- [10] Pietro di Gianantonio and Marino Miculan. Unifying recursive and co-recursive definitions in sheaf categories. In Igora Walukiewicz, editor, *Foundations of Software Science and Computation Structures (FOSSACS'04)*, volume 2987 of *LNCS*. Springer Verlag, 2004.

- [11] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. *The Coq Proof Assistant User's Guide*. INRIA, May 1993. Version 5.8.
- [12] Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for proofs and Programs*, volume 996 of *LNCS*, pages 39–59. Springer Verlag, 1994.
- [13] Eduardo Giménez. An application of co-inductive types in Coq: Verification of the alternating bit protocol. In *Proceedings of the 1995 Workshop on Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 135–152. Springer-Verlag, 1995.
- [14] P. Hudak, S. Peyton Jones, P. Wadler, et al. *Report on the Programming Language Haskell*. Yale University, New Haven, Connecticut, USA, 1992. Version 1.2.
- [15] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress 77*, pages 993–998. North-Holland, 1977.
- [16] François Leclerc and Christine Paulin-Mohring. Programming with streams in coq. A case study: the sieve of Eratosthenes. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 191–212. Springer Verlag, 1993.
- [17] Pierre Letouzey. A new extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
- [18] Bengt Nordström. Terminating general recursion. *BIT*, 28, 1988.
- [19] Christine Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in *Lecture Notes in Computer Science*, 1993. LIP research report 92-49.
- [20] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
- [21] David A. Turner. *SASL Language Manual*. St. Andrews University Department of Computer Science, 1976.





---

Unité de recherche INRIA Sophia Antipolis  
2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes  
4, rue Jacques Monod - 91893 ORSAY Cedex (France)

Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique  
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)

Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Unité de recherche INRIA Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

---

Éditeur  
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)  
<http://www.inria.fr>  
ISSN 0249-6399