# Filters on CoInductive streams an application to Eratosthenes' sieve

Yves Bertot

October 27, 2004

Our objective is to describe a formal proof of correctness for the following Haskell [13] program in a type theory-based proof verification system, such as the Coq system [10, 1].

```
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 [5, 11, 12] 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 prime numbers smaller than 4, and so on. 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 [14]. The exact formulation of the program given here was found in A. J. T. Davie's book on functional programming [7], who describes it as a re-phrasing of an initial program implemented in SASL by D. Turner [19]. A proof of Eratosthenes' sieve in type theory was already studied by F. Leclerc and C. Paulin-Mohring in 1993 [15], but their program has a different structure and does not exhibit the filter problem that is central here. Another program computing the stream of prime numbers is given as example in [8]. 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 [9].

Before performing the verification proof of such a program in type theory, we need to be able to formulate it. This is difficult, because type theory based frameworks only provided restricted capabilities for the definition of recursive functions, which basically ensure that all functions are total. 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. Because the result is an infinite stream, it means that an infinity of values should be found in the input, but this is not always possible and

actually depends on the input: filter functions are partial. 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 contraint 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 only if some information has been produced in the result, in the form of a constructor . The terminology is that calls must be "guarded-by-constructors" [11]. In their usual form, filters do not respect this syntactic criterion.

We propose to combine insights coming from reasoning techniques on linear temporal logic [4, 6] and on general recursion, essentially the techique advocated by A. Bove [2] 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 co-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.

# 1 Co-induction and co-recursion

Co-inductive types are defined by giving together a type and a collection of constructors. There is a tool, based on a pattern-matching construct, which expresses that all elements of a co-inductive type are obtained through one of the constructors. However, there is no obligation that the process of constructing a term in a co-inductive type should be a finite process as is usually the case for inductive types [17, 11].

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 intantiate `A` with the type `nat` of natural numbers.

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

This definition expresses that a stream of type `str` is like a list of elements of A: it has a first element and a tail, grouped together using the constructor `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 explicitely requested by a pattern-matching operation on these values.

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. We say that such a recursive function is *guarded by constructors*.

The "guarded-by-constructors" criterion can be theoretically justified by the fact that a co-inductive type actually is a final co-algebra in the category of co-algebras associated to the collection of constructors given in the co-inductive definition and a guarded-by-constructors function actually defines another co-algebra in this category. The existence of value in the final co-algebra is a natural consequence of the finality property.

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 implicitely anymore and we need to use explictely 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 co-recursive function whose type concludes on aco-inductive predicate, a 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.

# 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 Bove in [2]: 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 [4] and Coupet-Grimal [6] 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[1], 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 :

```
∀s, F_infinite s -> 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:

```
∀s, eventually s ->
  ∀x s', s = SCons x s'-> not(P x) -> eventually s'
```

For this definition to be accepted, the expression `eventually_inv s h x s' heq hn` must be recognized as a sub-term of `h`. This is achieved because this proof is actually obtained through a pattern-matching construct on the proof `h`. In this pattern-matching construct, we must ensure that the sub-expression that is returned in each possible case is a sub-term of `h`. There are two cases.

1. Either the proof `h` was obtained with the constructor `ev_r` applied to three arguments arguments `x1`, `s1`, and `h1`. In this case, `s1` $=$ `s'` and `h1` is a sub-term of `h` that is also a proof of `eventually s1`. We can return `h1`.
2. Either the proof `h` was obtained with the constructor `ev_b` applied to `x1`, `s1`, and `hp`. In this case, `x1` $=$ `x` and `hp` is a proof that `P x1` holds. The fact `hp` is inconsistent with the fact `hn` which must be a proof of `not(P x)`. Because of this inconsistency, we are relieved from the need to produce a sub-term proof.

In other words, we only need to produce a sub-term proof for the consistent cases. When the constructor that is used does not contain a sub-term proof for the recursive call, the fact that this constructor may have been used is inconsistent.

The function `pre_filter_i` is not satisfactory, because we also need the recursive function to produce the stream, on which filtering carries on, together with a proof that this stream contains an infinity of satisfactory elements. 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 handle 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 call 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 implicitely applied to `P` anymore. 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'.
```

This proof is tricky and we have to use a maximal induction principle as described in [1] (sect. 14.1.5).

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 [1], 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, the second is that the elements in the result are in the same order, with no repetition. These characteristics seem more complex to express because they are not local properties of each stream element taken separately, but they are global properties of the streams. We propose 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 stream's 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 `R1` to a connected stream for a relation `R2`, provided the relations `R1` and `R2` satisfy 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 R1 and R2 must satisfy express that if $x_1, \ldots x_k$ is a sequence of values such that P $x_1$ and P $x_k$ hold, not(P $x_i$) holds for all the other indices $i$ from 1 and $k$, and R1 $x_i$ $x_{i+1}$ holds, then R2 $x_1$ $x_k$ holds.

The theorem filter_connected is actually stronger than the two previous theorems. The theorem filter_always is a obtained with filter_connected for R1 the relation that is always satisfied and R2 the relation of x and y that holds if and only if P y holds. The theorem filter_keep Q is a corollary for R1 and R2 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 R1 and R2 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:

```
∀m n, {not_mult m n}+{not(not_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 := step_all pre_prime.
```

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 :=
    step_all (partial_prime 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 can construct the stream of all prime numbers (`lt_n_Sn 1` is a proof of $1 < 2$ and `le_n 2` is a proof of $2 \leq 3$):

```
Definition primes :=
   (sieve (nums 2) (conj (lt_n_Sn 1)(pprs2 2 (le_n 2)))).
```

We finally obtain the following theorem.

```
Theorem pre_primes: connected step_prime 1 primes.
```

The complete proof has been verified using the Coq system. The proof files are available at the following internet address:
`ftp://ftp-sop.inria.fr/lemme/Yves.Bertot/filters.tar.gz`

### 3.3 Code extraction

Once the `sieve` function is defined in the calculus of constructions, we can map it back to Haskell code using the extraction facility described in [16, 18]. The code we obtain for the `sieve` function is close to the one we initially intended to certify, except that it uses a re-defined type of streams, instead of using the built-in type of lists from Haskell.

```
sieve s =
  case s of
    SCons p rest ->
    SCons p (sieve (filter (\x -> mult_dec p x) rest))
```

The code we obtain for the `filter` function is less easy to recognize. A simple difference with the original code is that the extracted code uses its own datatype for boolean values, where `Left` is used to represent `True` and `Right` is used to represent `False`. The main difference is that the function is decomposed into two recursive functions. However, we maintain that this code is equivalent, up to the *unfolding/folding* technique of [3] to the initial `filter`.

```
filter_i p_dec s =
  case s of
    SCons x s' ->
      (case p_dec x of
         Left -> Pair x s'
         Right -> filter_i p_dec s')

filter p_dec s =
  case filter_i p_dec s of
    Pair a b -> SCons a (filter p_dec b)
```

# 4 Conclusion

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 is probably closer to the initial description of the sieve by Eratosthenes. 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 [14]. 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] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions*. Springer-Verlag, 2004.

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

[3] Rod M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[4] 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.

[5] 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.

[6] 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.

[7] Antony J. T. Davie. *An introduction to functional programming systems using Haskell*. Cambridge Computer Science texts. Cambridge University Press, 1992.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress 77*, pages 993–998. North-Holland, 1977.

[15] 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 Progams*, volume 806 of *LNCS*, pages 191–212. Springer Verlag, 1993.

[16] 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.

[17] 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.

[18] Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[19] David A. Turner. *SASL Language Manual*. St. Andrews University Department of Computer Science, 1976.