CoInduction in Coq

Yves Bertot

CoInduction in Coq

Yves Bertot

March 29, 2006

When providing a collection of constructors to define an inductive type, we actually also define a dual operation: a destructor. This destructor is always defined using the same structure of pattern-matching, so that we have a tendency to forget that we do extend the "pattern-matching" capability with a new destructor at each definition.

Constructors and destructors play a dual role in the definition of inductive types. Constructors produce elements of the inductive type, destructors consume elements of the inductive type.

The inductive type itself is defined as the smallest collection of elements that is stable with respect to the constructors: it must contain all constants that are declared to be in the inductive type and all results of the constructors when the arguments of these constructors are already found to be in the inductive type. When considering structural recursion, recursive definitions are functions that consume elements of the inductive type. The discipline of structural recursion imposes that recursive calls *consume* data that is *obtained* through the destructor.

The inductive type uses the constructors and destructors in a specific way. Co-inductive types are the types one obtains when using them in a dual fashion. A co-inductive type will appear as the largest collection of elements that is stable with respect to the destructor. It contains every object that can be destructed by pattern-matching.

The duality goes on when considering the definition of recursive functions. Co-recursive functions are function that produce elements of the co-inductive type. The discipline of guarded co-recursion imposes that co-recursive calls *produce* data that is *consumed* by a constructor. The main practical consequence is that co-inductive types contain objects that look like *infinite objects*.

This rough sketch is more of a philosophical nature. When looking at the details, there are some aspects of co-inductive types that are not so simple to derive from a mere reflection of what happens with inductive types.

The possibility to have co-inductive types in theorem proving tools was studied by Coquand [7], Paulson [19], Leclerc and Paulin-Mohring [16], and Gimenez [13]. Most of these authors were inspired by Aczel [1]. The paper [2] provides a short presentation of terms and (possibly infinite) trees, mainly in set-theoretic terms; it also explains recursion and co-recursion.

In this document, we only consider the use of co-inductive types as it is provided in Coq.

1 Defining a co-inductive types

Since defining a set of constructors automatically defines a destructor, the definition of co-inductive types also relies on the definition of constructors. The same rules of positivity as for inductive types apply. Here are three simple examples of co-inductive types:

```
CoInductive Llist (A:Set) : Set :=
  Lcons : A -> Llist A -> Llist A | Lnil : Llist A.

CoInductive stream (A:Set) : Set :=
  Cons : A -> stream A -> stream A.

CoInductive Ltree (A:Set) : Set :=
  Lnode : A -> Ltree A -> Ltree A -> Ltree A  | Lleaf : Ltree A.
```

As for inductive types, this defines the type and the constructors, it also defines the destructor, so that every element of the co-inductive can be analysed by pattern-matching. However, the definition does not provide an induction principle. The reason for the absence of an induction principle can be explained in two ways, philosophical or technical. Philosophically, the induction principle of an inductive type expresses that this inductive type is minimal (it is a least fixed point), but the co-inductive type is rather viewed as greatest fixed point. Technically, the induction principle actually is a *consumption* tool, that consumes elements of an inductive type to produce proofs in some other type, programed by recursion. However, a co-recursive function can only be used to produce elements of the co-recursive type, so that the only way to deduce anything from an element of a co-inductive type is by pattern-matching.

The type `Llist` given above comes with constructors `Lcons` and `Lnil`. These constructors will make it possible to produce lists, exactly like the lists that we could produce in the inductive type `list`. Here are a few examples.

```
Implicit Arguments Lcons.
Implicit Arguments Cons.

Definition ll123 := Lcons 1 (Lcons 2 (Lcons 3 (Lnil nat))).

Require Import List.

Fixpoint list_to_llist (A:Set) (l:list A)
    {struct l} : Llist A :=
 match l with
   nil => Lnil A
 | a::tl => Lcons a (list_to_llist A tl)
 end.

Definition ll123' := list_to_llist nat (1::2::3::nil).
```

The function `list_to_llist` is recursive and produces elements in a co-inductive type, but we did not make it rely on co-recursion. Rather, it relies on structural recursion as it is provided with the inductive type `list`.

Similar examples which do not rely on co-recursion cannot be provided for the type `stream`, because elements of this type cannot be given with a finite number of uses of the constructor. There is always a need for another element of the co-inductive type. Co-recursion is the solution.

In the coq system, co-recursion is only allowed under a form that can ensure the strong normalization properties that are already satisfied by the inductive part of the calculus. The decision was taken to impose a syntactic criterion: co-recursive values can only appear as argument to constructors and inside branches of pattern-matching constructs. Here is a simple example:

```
CoFixpoint ones : stream nat := Cons 1 ones.
```

This definition underlines a funny aspect of co-recursion: a co-recursive value is not necessarily a function, because it is only constrained to *produce* an element of the co-inductive type. The definition contains a usage of the constructor `Cons` and a reference to the co-recursive value itself, but this co-recursive value is used as an argument to the constructor.

A similar value can be defined in the type `Llist`.

```
CoFixpoint lones : Llist nat := Lcons 1 lones.
```

Clearly, the list that we obtain is not a list that we could have obtained using the function `list_to_llist`. The type `Llist` is “larger” than the type `list`.

Some co-recursive functions can be defined to perform exactly like similar recursive functions on inductive types. Here is an instance:

```
Fixpoint map (A B:Set)(f:A -> B)(l:list A) {struct l} : list B :=
  match l with
    nil => nil
  | a::tl => f a::map A B f tl
  end.

CoFixpoint lmap (A B:Set)(f:A -> B)(l:Llist A) : Llist B :=
  match l with
    Lnil => Lnil B
  | Lcons a tl => Lcons (f a) (lmap A B f tl)
  end.
```

The two functions look similar, but we should bear in mind that the second one can also process infinite lists like `lones`.

2 Computing with co-recursive values

When we manipulate elements of inductive types, we implicitely expect to look at these values in constructor form: constructors applied to other terms in constructor form. However, attempting to put a value like `lones` in constructor form would require an infinity of unfoldings of its value, this would make the computation non-normalizing. For this reason, a co-recursive value is by default considered to be a normal form. This can be verified by requesting a computation on such a value.

```
Eval simpl in lones.
 = lones : Llist nat
```

However destructing a co-recursive value (with the help of a pattern-matching construct) corresponds to a regular redex and we can check that the first element of `lones` indeed is 1.

```
Eval simpl in
 match lones with Lnil => 0 | Lcons a _ => a end.
 = 1 : nat
```

3 Proving properties of co-inductive values

It is possible to prove that two co-inductive values are equal. The usual approach, identical to what happens in the inductive setting is to show that the two values have the same head constructor applied to the same arguments. However, making the head constructor appear is tricky, because the computation of co-recursive values is usually not performed. One way to provoke this computation is to rely on the following function and theorem.

```
Definition Llist_decompose (A:Set)(l:Llist A) : Llist A :=
  match l with Lnil => Lnil A | Lcons a tl => Lcons a tl end.
Implicit Arguments Llist_decompose.

Theorem Llist_dec_thm :
   forall (A:Set)(l:Llist A), l = Llist_decompose l.
Proof.
 intros A l; case l; simpl; trivial.
Qed.
```

Now, here is an example using this theorem:

```
Theorem lones_cons : lones = Lcons 1 lones.
Proof.
pattern lones at 1; rewrite Llist_dec_thm; simpl.
...
  ============================
   Lcons 1 lones = Lcons 1 lones

trivial.
Qed.
```

This is not a proof by co-recursion, just a proof by pattern-matching.

There are proofs of equality that seem obvious but cannot be performed in the calculus of inductive constructions as it is defined now. This happens when the proofs seems to require some sort of inductive argument. Here is an instance of an impossible proof:

```
Theorem lmap_id : forall (A:Set)(l:Llist A),
   lmap A A (fun x:A => x) l = l.
```

One would like to have an argument of the following form: if the list is nil, then the proof is trivial, if the list is not `nil`, then the head on both sides of the equality are naturally the same, and the equality for the tails should hold by some "inductive" argument. However, there is no induction hypothesis, because there is no inductive list in this statement and the proof of equality can only be proved by using the constructor of equality (because equality is itself an inductive type).

The solution for this kind of problem is to use a co-inductive proposition that expresses the equality of two lists by stating that they have the same elements. Here is the co-inductive definition for this proposition:

```
CoInductive bisimilar (A:Set) : Llist A -> Llist A -> Prop :=
  bisim0 : bisimilar A (Lnil A) (Lnil A)
| bisim1 : forall a l l',
       bisimilar A l l' -> bisimilar A (Lcons a l)(Lcons a l').
```

Proofs that two lists have the same elements can now also be obtained by using co-recursive values, as long as we use the `bisimilar` relation instead of equality. Here is an example of a proof, displayed as term of the calculus of inductive constructions to make the general structure visible. Please note that rewrites using the theorems `eq_ind_r` and `Llist_dec_thm` are performed to introduce the function `Llist_decompose` and force the expansion of the co-recursive function.

```
CoFixpoint lmap_bi (A:Set)(l:Llist A) :
  bisimilar A (lmap A A (fun x:A => x) l) l :=
 @eq_ind_r (Llist A) (Llist_decompose (lmap A A (fun x=> x) l))
   (fun x => bisimilar A x l)
   match l return bisimilar A
                    (Llist_decompose (lmap A A (fun x=>x) l))
                    l with
     Lnil => bisim0 A
   | Lcons a k =>
             bisim1 A a (lmap A A (fun x=> x) k) k (lmap_bi A k)
   end
   (lmap A A (fun x => x) l)
   (Llist_dec_thm A (lmap A A (fun x=>x) l))
.
```

The manual construction of co-inductive proofs is difficult. The alternative approach is to use tactics. The following script performs the same proof, but relying on tactics.

```
Theorem lmap_bi' : forall (A:Set)(l:Llist A),
  bisimilar A (lmap A A (fun x => x) l) l.
cofix.
intros A l; rewrite (Llist_dec_thm _ (lmap A A (fun x=>x) l)).
case l.
intros a k; simpl.
apply bisim1; apply lmap_bi'.
```

```
simpl; apply bisim0.
Qed.
```

The tactic `cofix` is the tactic that declares that the current proof will be a co-recursive value. It introduces a new assumption in the context so that the co-recursive value can be used inside its own definition. However, the same constraints as before exist: the co-recursive value can only be used as input to a constructor. In the case of `lmap_bi`, the use of `lmap_bi'` at the end of the proof is justified by the previous use of the constructor `bisim1`: `lmap_bi'` is thus used to provide an argument to `bisim1`.

In general, the constraint that co-recursive calls are used in correct conditions is only checked at the end of the proof. This sometimes has the unpleasant effect that one believes to have completed a proof and is only rebuked when the `Qed` or `Defined` commands announce that the constructed term is not well-formed. This problem is compounded by the fact that it is hard to control the hypotheses that are used by automatic tactics. Even though we believe the proof of a subgoal should not rely on the co-recursive assumption, it may happen that some tactic like `intuition` uses this assumption in a bad way. One solution to this problem is to use the `clear` tactic to remove the co-recursive assumption before using strong automatic tactics. A second important tool to avoid this problem is a command called `Guarded`, this command can be used at any time during the interactive proofs and it checks whether illegal uses of the co-recursive tactic have already been performed.

4 Applications

Co-inductive types can be used to reason about hardware descriptions [9] concurrent programming [14], finite state automata and infinite traces of execution, and temporal logic [5, 8]. The guarded by constructors structure of co-recursive functions is adapted to representing finite state automata. A few concrete examples are also given in [4].

Co-inductive types are especially well suited to model and reason about lazy functional programs that compute on infinite lists. However, the constraints of having co-recursive calls guarded by constructors imposes that one scrutinizes the structure of recursive functions to understand whether they really can be encoded in the language. One approach, used in [3] is to show that co-inductive objects also satisfy some inductive properties, which make it possible to define functions that have a recursive part, with usual structural recursive calls with respect to these inductive properties, and guarded co-recursive parts.

5 An example: introduction to exact real arithmetics

The work presented in this section is my own, but it is greatly inspired by reading the lecture notes [10] and the thesis [17] and derived papers [18, 15], and by [6]. These papers should be consulted for further references about exact real arithmetics, lazy computation, and co-inductive types.

We are going to represent real numbers between 0 and 1 (included) as infinite sequences of intervals I_n, where $I_0 = [0, 1]$, $I_{n+1} \subset I_n$ and the size of I_{n+1} is half the size of I_n. Moreover, I_{n+1} will be obtained from I_n in only one of three possible ways:

1. I_{n+1} is the left half of I_n,
2. I_{n+1} is the right half of I_{n+1},
3. I_{n+1} is the center of I_{n+1}: if a and b are the bounds of I_n, then $a + (b - a)/4$ and $a + 3(b - a)/4$ are the bounds of $I_n + 1$.

We can represent any of the intervals I_n using lists of `idigit`, where the type `idigit` is the three element enumerated type containing `L`, `R`, `C`. For instance, the interval [0,1] is given by the empty list, the interval [1/4,3/8] can be represented by the lists `L::C::R::nil`, `L::R::L::nil`, or `C::L::L::nil`. It is fairly easy to write a function of type `list idigit->R` that maps every list to the lower and upper bound of the interval it represents. We are going to represent real numbers by infinite sequences of intervals using the type `stream idigit`.

There is also an easy correspondence from floating-point numbers in binary representation to this representation. Let us first recall what the binary floating-point representation is. Any "binary" floating point is a list of boolean values. Interpreting `true` as the 1 bit and `false` as the 0 bit, a boolean list is interpreted as a real number in the following way:

```
Fixpoint bit_list_to_R (l:list boolean) : Rdefinitions.R :=
  match l with
    nil => 0
  | b::tl => let x := bit_list_to_R tl in
             if b then (1+x)/2 else x/2
  end.
```

We can inject the boolean values into the type `idigit` mapping `true` to `L` and `false` to `R`. It is fairly easy to show that this correspondance can be lifted to lists of booleans and idigits, so that the real number represented by a list is element of the interval represented by the corresponding list.

We represent real numbers by streams of `idigit` elements. The construction relies on associating a sequence of real numbers to each stream (actually the lower bounds of the intervals) and to show that this sequence converges to a limit. To ease our reasoning, we will also describe the relation between a stream and a real value using a co-inductive property:

```
CoInductive represents : stream idigit -> Rdefinitions.R -> Prop :=
  reprL : forall s r, represents s r -> (0 <= r <= 1)%R ->
          represents (Cons L s) (r/2)
| reprR : forall s r, represents s r -> (0 <= r <= 1)%R ->
          represents (Cons R s) ((r+1)/2)
| reprC : forall s r, represents s r -> (0 <= r <= 1)%R ->
          represents (Cons C s) ((2*r+1)/4).
```

We could also use infinite lists of booleans to represent real numbers. This is the usual representation of numbers. This representation also corresponds to sequences of intervals, but it has bad programming properties. In this representation, if we know that a number is very close to 1/2 but we don't know whether it is larger or smaller, we cannot produce the first bit. For instance, the number 1/3 is represented by the infinite sequence .0101... and the number 1/6 is represented by the infinite sequence .0010101... Adding the two numbers should yield the number 1/2. However, every finite prefix of .010101... represents an interval that contains numbers that are larger than 1/3 and numbers that are smaller than 1/3. Similarly, every finite prefix of .0010101... contains a numbers that are larger than 1/6 and numbers that are smaller. By only looking at a finite prefix of both numbers, we cannot decide whether the first bit of the result should be a 0 or a 1, because no number larger than 1/2 can be represented by a sequence starting with a 0 and no number smaller than 1/2 can be represented by a sequence starting with a 1.

With the extra digit, `C`, we can perform the computation as follows:

1. having observed that the first number has the form $x = LRx'$, we know that this number is between 1/4 and 1/2,
2. having observed that the second number has the form $y = LLy'$, we know that this number is between 0 and 1/4,
3. we know that the sum is between 1/4 and 3/4. therefore, we know that the sum is an element of the interval represented by `C::nil`, and we can output this digit.

We can also go on to output the following digits. In usual binary representation, if v is the number represented by the sequence s, then the number represented by the sequence $0s$ is $v/2$ and the number represented by the sequence $1s$ is $(v+1)/2$. This interpretation carries over to the digits `L` and `R`, respectively. For the digit `C`, we know that the sequence `C`s represents $(2v+1)/4$. Thus, if we come back to the computation of $1/3 + 1/6$, we know that x' is $4 * x - 1$, y' is $4 * y$, and the result should have the form `C::`z, where z is the representation of $(x' + y' + 1)/4$ (since $x' + y' + 1)/4$ is 1/2, we see that the result of the sum is going to be an infinite sequence of `C` digits.

We are now going to provide a few functions on streams. As a first example, the function `rat_to_stream` maps any two integers a b to a stream. When a/b is between 0 and 1, the result stream is the representation of this rational number.

```
CoFixpoint rat_to_stream (a b:Z) : stream idigit :=
  if Z_le_gt_dec (2*a) b then
    Cons L (rat_to_stream (2*a) b)
  else
    Cons R (rat_to_stream (2*a-b) b)
```

For the second example, we compute an affine combination of two numbers with rational coefficients. We will define the function that constructs the representation of the following formula.

$$\frac{a}{a'}v_1 + \frac{b}{b'}v_2 + \frac{c}{c'}$$

The numbers a, a', ... are positive integers and a', b', and c' are non-zero (this sign restriction only serves to make the example shorter).

We choose to define a one-argument function, where the argument is a record holding all the values a, a', ..., v_1, v_2. We define a type for this record and a predicate to express the sign conditions.

```
Record affine_data : Set :=
 {m_a : Z; m_a' : Z; m_b : Z; m_b' : Z; m_c : Z; m_c' : Z;
  m_v1 : stream idigit; m_v2 : stream idigit}.

Definition positive_coefficients (x:affine_data) :=
  0 <= m_a x /\ 0 < m_a' x /\ 0 <= m_b x /\ 0 < m_b' x
  /\ 0 <= m_c x /\ 0 < m_c' x.
```

We define a function `axbyc` of type

```
 forall x, positive_coefficients x -> stream idigit.
```

The algorithm contains two categories of computing steps. In computing steps of the first category, a digit of type `idigit` is produced, because analysing the values of a, a', …makes it possible to infer that the result will be in a precise part of the interval. The result then takes the form

$$\texttt{Cons}\ d\ \texttt{(axbyc}\ \langle a_1, a_1', b_1, b_1', c_1, c_1', v_1, v_2\rangle\texttt{)}$$

Where d is a digit and the values of a_1, a_1', …depend on the digit.

1. if $c/c' \geq 1/2$, then the result is sure to be in the right part of the interval, the digit d is `R` and the new parameters are chosen so that $a_1/a_1' = 2a/a'$, $b_1/b_1' = 2b/b'$, $c_1/c_1' = (2c - c')/c'$, because of the following equality:

$$\frac{1}{2}(\frac{2a}{a'}v_1 + \frac{2b}{b'}v_2 + \frac{2c-c'}{c'}) + \frac{1}{2} = \frac{a}{a'}v_1 + \frac{b}{b'}v_2 + \frac{c}{c'}$$

2. if $2(ab'c' + ba'c' + a'b'c) \leq a'b'c'$, then the result is sure to be in the left part of the interval, the digit d is `L` and the new parameters are chosen so that $a_1/a_1' = 2a/a'$, $b_1/b_1' = 2b/b'$, $c_1/c_1' = 2c/c'$ (we do not detail the justification),

3. if $(4(ab'c' + ba'c' + a'b'c) \leq 3a'b'c'$ and $4 * c \geq c'$, then the result is sure to belong to the center sub-interval, the digit d is `C` and the new parameters are chosen so that $a_1/a_1' = 2a/a'$, $b_1/b_1' = 2b/b'$, $c_1/c_1' = (4c - c')/2c'$.

The various cases of these productive steps are described using the following functions:

```
Definition prod_R x :=
  Build_affine_data (2*m_a x) (m_a' x) (2*m_b x) (m_b' x)
  (2*m_c x - m_c' x) (m_c' x) (m_v1 x) (m_v2 x).

Definition prod_L x :=
  Build_affine_data (2*m_a x) (m_a' x) (2*m_b x) (m_b' x)
  (2*m_c x) (m_c' x) (m_v1 x) (m_v2 x).

Definition prod_C x :=
  Build_affine_data (2*m_a x) (m_a' x) (2*m_b x) (m_b' x)
  (4*m_c x - m_c' x) (2*m_c' x) (m_v1 x) (m_v2 x).
```

In the second category of computing steps the values v_1 and v_2 are scrutinized, so that the interval for the potential values of the result is reduced as one learns more information about the inputs. If the values v_1 and v_2 have the form `Cons` d_1 v_1' and `Cons` d_2 v_2' respectively, The result then takes the form

$$\texttt{axbyc}\ \langle a, 2a', b, 2b', c_1, c_1', v_1', v_2'\rangle$$

Only the parameters c_1 and c_1' take a different form depending on the values of d_1 and d_2. The correspondance is given in the following table.

d_1	d_2	c_1	c_1'
L	L	c	c'
L	R	$bc' + 2cb'$	$2b'c'$
R	L	$ac' + 2ca'$	$2a'c'$
L	C	$bc' + 4cb'$	$4b'c'$
C	L	$ac' + 4ca'$	$4a'c'$
R	C	$2ba'c' + ab'c' + 4cb'a'$	$4a'b'c'$
C	R	$2ba'c' + ab'c' + 4cb'a'$	$4b'a'c'$
R	R	$ab'c' + ba'c' + 2ca'b'$	$2a'b'c'$
C	C	$ba'c' + ab'c' + 4cb'a'$	$4b'a'c'$

For justification, let us look only at the case where $v_1 = \mathtt{R}v_1'$ and $v_2 = \mathtt{C}v_2'$. In this case we have the following equations:

$$\begin{aligned}\frac{a}{a'}v_1 + \frac{b}{b'}v_2 + \frac{c}{c'} &= \frac{a}{a'}(\frac{1}{2}v_1' + \frac{1}{2}) + \frac{b}{b'}(\frac{1}{2}v_2' + \frac{1}{4}) + \frac{c}{c'}\\ &= \frac{a}{2a'}v_1' + \frac{b}{2b'}v_2' + \frac{2ba'c' + ab'c' + 4cb'a'}{4a'b'c'}\end{aligned}$$

This category of computation is taken care of by a function with the following form:

```
Definition axbyc_consume (x:affine_data) :=
 let (a,a',b,b',c,c',v1,v2) := x in
 let (d1,v1') := v1 in let (d2,v2') := v2 in
 let (c1,c1') :=
  match d1,d2 with
  | L,L => (c, c')
  | L,R => (b*c'+2*c*b', 2*b'*c')
  | R,L => (a*c'+2*c*a', 2*a'*c')
  | L,C => (b*c'+4*c*b', 4*b'*c')
  | C,L => (a*c'+4*c*a', 4*a'*c')
  | R,C => (2*a*b'*c'+b*a'*c'+4*c*a'*b', 4*a'*b'*c')
  | C,R => (2*b*a'*c'+a*b'*c'+4*c*b'*a', 4*b'*a'*c')
  | R,R => (a*b'*c'+b*a'*c'+2*c*a'*b', 2*a'*b'*c')
  | C,C => (b*a'*c'+a*b'*c'+4*c*b'*a', 4*b'*a'*c')
  end in
 Build_affine_data a (2*a') b (2*b') c1 c1' v1' v2'.
```

From the point of view of co-recursive programming, the first category of computing steps gives regular guarded-by-constructor corecursive calls. The second category of computing steps does not give any guarded corecursion. We need to separate the second category in an auxiliary function. We choose to define this auxiliary function by well-founded induction. The recursive function performs the various tests with the help of an auxiliary test function:

```
  Parameter axbyc_test :
  forall x,
   positive_coefficients x ->
   m_c’ x <= 2*m_c x+
   2*(m_a x*m_b’ x*m_c’ x +
        m_b x*m_a’ x*m_c’ x + m_a’ x*m_b’ x*m_c x)<=
     m_a’ x*m_b’ x*m_c’ x+
   4*(m_a x*m_b’ x*m_c’ x +
        m_b x*m_a’ x*m_c’ x + m_a’ x*m_b’ x*m_c x)<=
     3*m_a’ x*m_b’ x*m_c’ x /\ m_c’ x <= 4*m_c x+
   m_a’ x < 8*m_a x  m_b’ x < 8*m_b x.
```

In the first three cases, the recursive function just returns the value that it received, together with the proofs of the properties. To carry these agregates of values and proofs, we defined a specific type to combine these values and proofs.

```
Inductive decision_data : Set :=
  caseR : forall x:affine_data, positive_coefficients x ->
          m_c’ x <= 2*m_c x -> decision_data
| caseL : forall x:affine_data, positive_coefficients x ->
          2*(m_a x*m_b’ x*m_c’ x +
             m_b x*m_a’ x*m_c’ x + m_a’ x*m_b’ x*m_c x)<=
           m_a’ x*m_b’ x*m_c’ x -> decision_data
| caseC : forall x:affine_data, positive_coefficients x ->
          4*(m_a x*m_b’ x*m_c’ x +
             m_b x*m_a’ x*m_c’ x + m_a’ x*m_b’ x*m_c x)<=
          3*m_a’ x*m_b’ x*m_c’ x -> m_c’ x <= 4*m_c x ->
          decision_data.
```

The recursive function will thus have the type

```
 forall x, positive_coefficient x -> decision_data.
```

The definition has the following form:

```
Definition axbyc_rec_aux (x:affine_data)
   : (forall y, order y x ->
         positive_coefficients y -> decision_data)->
     positive_coefficients x -> decision_data :=
  fun f Hp =>
  match A.axbyc_test x Hp with
    inleft (inleft (left H)) => caseR x Hp H
  | inleft (inleft (right H)) => caseL x Hp H
  | inleft (inright (conj H1 H2)) => caseC x Hp H1 H2
  | inright H =>
    f (axbyc_consume x)
      (A.axbyc_consume_decrease x Hp H)
      (A.axbyc_consume_pos x Hp)
  end.

Definition axbyc_rec :=
  well_founded_induction A.order_wf
  (fun x => positive_coefficients x -> decision_data)
  axbyc_rec_aux.
```

The definition of axbyc_rec of course relies on proofs to ensure that axbyc_consume preserves the sign conditions and make the measure decrease, we do not include these proofs in these notes.

The main co-recursive function relies on the auxiliary recursive function to perform all the recursive calls that are not productive, the value returned by the auxiliary function is suited to produce data and co-recursive calls are then allowed.

```
CoFixpoint axbyc (x:affine_data)
   (h:positive_coefficients x):stream idigit :=
  match axbyc_rec x h with
    caseR y Hpos H => Cons R (axbyc (prod_R y) (A.prod_R_pos y Hpos H))
  | caseL y Hpos H => Cons L (axbyc (prod_L y) (A.prod_L_pos y Hpos))
  | caseC y Hpos H1 H2 =>
         Cons C (axbyc (prod_C y) (A.prod_C_pos y Hpos H2))
  end.
```

This function relies on auxiliary functions to perform the relevant updates of the various coefficients. For instance, here is the function prod_C:

```
Definition prod_C x :=
  Build_affine_data (2*m_a x) (m_a' x) (2*m_b x) (m_b' x)
  (4*m_c x-m_c' x) (m_c' x) (m_v1 x) (m_v2 x).
```

For each of these functions, it is also necessary to prove that they preserve the sign conditions, these proofs are fairly trivial.

It requires more work to prove that the function is correct, in the sense that it does produce the representation of the right real number, but this proof is too long to fit in these short tutorial notes. More work is also required to make the function more efficient, for instance by dividing a (resp. b, c) and a' (resp. b', c') by they greatest common divisor at each step.

The representation for real numbers proposed in [10] is very close to the representation used in these notes, except that the initial interval is [-1,1], and the three digits are interpreted as the sub-intervals [-1,0], [0,-1], [-1/2,1/2]. The whole set of real numbers is then encoded by multiplying a number in [-1,1] by an exponent of 2 (as in usual scientific, floating point notation). The work presented in [17] shows that both the representation in these notes and the representation in [10] are a particular case of a general framework based on overlapping intervals and proposes a few other solutions. In these notes, we have decided to restrict ourselves to affine binary operations, which makes it possible to obtain addition and multiplication by a rational number, but the most general setting relies on homographic and quadratic functions, which make it possible to obtain addition, multiplication, and division, all in one shot.

The method of separating a recursive part from a co-recursive part in a function definition was already present in [3]. However, the example of [3] is more complex because the functions are *partial*: there are streams for which eventual productivity is not ensured and a stronger description technique is required. This stronger technique is described in [4] as *ad-hoc* recursion. The papers [11, 12] propose an alternative foundation to functions that mix recursive and co-recursive parts.

6 Exercises

increasing streams Define a co-inductive predicate that is satisfied by any stream such that, if n and m are consecutive elements, then $n \leq m$.

Fibonnacci streams Define a co-inductive predicate, called `local_fib`, that is satisfied by any stream such that, if n, m, p are consecutive elements, then $p = n+m$. Define a co-recursive function that constructs a fibonacci stream whose first two elements are 1. Prove that the stream that is created satisfies the two predicates (`increasing` and `local_fib`).

7 Solutions

```
Require Export Omega.

CoInductive increasing : stream nat -> Prop :=
  ci : forall a b tl, a <= b -> increasing (Cons b tl) ->
              increasing (Cons a (Cons b tl)).

CoInductive local_fib : stream nat -> Prop :=
  clf : forall a b tl, local_fib (Cons b (Cons (a+b) tl)) ->
       local_fib (Cons a (Cons b (Cons (a+b) tl))).

CoFixpoint fibo_str (a b:nat) : stream nat := Cons a (fibo_str b (a + b)).

Definition str_decompose (A:Set)(s:stream A) : stream A :=
  match s with Cons a tl => Cons a tl end.

Implicit Arguments str_decompose.

Theorem str_dec_thm : forall (A:Set)(s:stream A), str_decompose s = s.
Proof.
intros A [a tl];reflexivity.
Qed.

Implicit Arguments str_dec_thm.

Theorem increasing_fibo_str :
  forall a b, a <= b -> increasing (fibo_str a b).
Proof.
Cofix.
intros a b Hle.
rewrite <- (str_dec_thm (fibo_str a b));simpl
assert (Heq:(fibo_str b (a+b))=(Cons b (fibo_str (a+b) (b+(a+b))))).
rewrite <- (str_dec_thm (fibo_str b (a+b)));simpl;auto.
rewrite Heq.
constructor.
assumption.
rewrite <- Heq.
apply increasing_fibo_str.
omega.
Qed.

Theorem increasing_fib : increasing (fibo_str 1 1).
Proof.
 apply increasing_fibo_str;omega.
Qed.
```

```
Theorem local_fib_str :
  forall a b, local_fib (fibo_str a b).
Proof.
cofix.
intros a b.
assert (Heq :
          (fibo_str b (a+b)) =
          (Cons b (Cons (a+b)(fibo_str (b+(a+b))((a+b)+(b+(a+b))))))).
rewrite <- (str_dec_thm (fibo_str b (a+b))); simpl.
rewrite <- (str_dec_thm (fibo_str (a+b) (b+(a+b)))); simpl;auto.
rewrite <- (str_dec_thm (fibo_str a b)); simpl.
rewrite Heq.
constructor.
rewrite <- Heq.
apply local_fib_str.
Qed.

Theorem local_fib_fib : local_fib (fibo_str 1 1).
Proof.
 apply local_fib_str.
Qed.
```

References

[1] Peter Aczel. *Non-Well-Founded Sets.* CSLI Lecture Notes, volume 14, 1988.

[2] Yves Bertot. Algebras and Coalgebras. *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, volume 2297 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.

[3] Yves Bertot. Filters on CoInductive Streams, an Application to Eratosthenes' Sieve. *Typed Lamdba-Calculi and Applications'05*, volume 3461 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[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] Alberto Ciaffaglione and Pietro Di Gianantonio. A Co-Inductive Approach to Real Numbers. *Types for Proofs and Programs*, volume 1956 of *Lecture Notes in Computer Science*, Springer-Verlag, 2000.

[7] Thierry Coquand. Infinite objects in Type Theory. *Types for Proofs and Programs*, volume 806 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.

[8] 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.

[9] Solange Coupet-Grimal and Line Jakubiec. Hardware verification using co-induction in coq. In *TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[10] Abbas Edalat and Reinhold Heckmann. Computing with Real Numbers. *Applied semantics*, volume 2395 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.

[11] 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.

[12] Pietro di Gianantonio and Marino Miculan. Unifying recursive and co-recursive definitions in sheaf categories. In Igor Walukiewicz, editor, *Foundations of Software Science and Computation Structures (FOSSACS'04)*, volume 2987 of *LNCS*. Springer Verlag, 2004.

[13] Eduardo Giménez. Codifying guarded definitions with recursive schemes. *Types for Proofs and Programs*, volume 996 of *Lecture Notes in Computer Science*, Springer-Verlag, 1994.

[14] Eduardo Giménez. An Application of Co-Inductive Types in Coq: Verification of the Alternating Bit Protocol. *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, Springer-Verlag, 1995.

[15] Jesse Hughes and Milad Niqui. Admissible digit sets. To appear in *Theoretical Computer Science*, special issue on real numbers and computers, 2005.

[16] François Leclerc and Christine Paulin-Mohring. Programming with streams in coq. A case study: the sieve of Eratosthenes. *Types for Proofs and Progams*, volume 806 of *Lecture Notes in Computer Science*, Springer-Verlag, 1993.

[17] Milad Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Raadboud University, Nijmegen, 2004.

[18] Milad Niqui. Formalising Exact Arithmetic in Type Theory. First conference on computability in Europe, *CiE2005*, volume 3526 of *Lecture Notes in Computer Science*, Springer-Verlag, 2005.

[19] Lawrence C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. *Conference on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag 1994.