Inductive and Coinductive Components of Corecursive Functions in Coq

Ekaterina Komendantskaya, joint work with Yves Bertot

INRIA Sophia Antipolis

TYPES’08,
28 March 2008
Motivation

- Inductive and Coinductive Types in Coq
- Terminative and Productive Functions
- Structural Recursive and Guarded Functions
Outline

1 Motivation
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Structural Recursive and Guarded Functions

2 Formalisation of Productive Non-Guaranteed Functions in Coq
   - Inductive Component of Corecursive Functions
   - Coinductive Component of Corecursive Functions
   - Proving Properties about the Models
Outline

1 Motivation
   - Inductive and Coinductive Types in Coq
   - Terminative and Productive Functions
   - Structural Recursive and Guarded Functions

2 Formalisation of Productive Non-Guaraded Functions in Coq
   - Inductive Component of Corecursive Functions
   - Coinductive Component of Corecursive Functions
   - Proving Properties about the Models

3 Conclusions
Filter Function, [Bertot05]

Definition

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

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

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

Fixpoint div2 n : nat :=
  match n with
  | O => 0
  | S O => 0
  | S (S n') => S (div2 n')
end.
Coinductive Types and Corecursive Functions

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

CoFixpoint repeat (a: A): str A :=
SCons a (repeat a).
We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\rightarrow$ types; proofs $\rightarrow$ programs): non-terminating proofs can lead to inconsistency.
Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\rightarrow$ types; proofs $\rightarrow$ programs): non-terminating proofs can lead to inconsistency.
- To decide type-checking of dependent types, we need to reduce expressions to normal form.
Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions $\rightarrow$ types; proofs $\rightarrow$ programs): non-terminating proofs can lead to inconsistency.
- To decide type-checking of dependent types, we need to reduce expressions to normal form.

Example

The function `div2` is terminative.
Productive Values

Values in co-inductive types are **productive** when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.
Productive Values

Values in co-inductive types are **productive** when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

The element of the stream at position \( n \) can be found by:

\[
\begin{aligned}
\text{nth} \ 0 \ (\text{SCons} \ a \ \text{tl}) &= a \\
\text{nth} \ (S \ n) \ (\text{SCons} \ a \ \text{tl}) &= \text{nth} \ n \ \text{tl}
\end{aligned}
\]

A given stream \( s \) is productive if we can be sure that the computation of the list \( \text{nth} \ n \ s \) is guaranteed to terminate, whatever the value of \( n \) is.
Motivation

Terminative and Productive Functions

Productive Values

Values in co-inductive types are **productive** when all observations of fragments made using recursive functions are guaranteed to be computable in finite time.

The element of the stream at position $n$ can be found by:

**Definition**

\[
\begin{align*}
\text{nth } 0 \ (\text{SCons a tl}) &= a \\
\text{nth } (S \ n) \ (\text{SCons a tl}) &= \text{nth } n \ tl
\end{align*}
\]

A given stream $s$ is productive if we can be sure that the computation of the list $\text{nth } n \ s$ is guaranteed to terminate, whatever the value of $n$ is.

**Example**

For any $n$, the value $\text{repeat } n$ is productive.
Productive Functions

We call a function *productive at the input value* $i$, if it outputs a productive value at $i$.

**Definition**

Let $A$, $B$ be of type $\text{Set}$. For a predicate $P : B \rightarrow \text{bool}$ and functions $h : B \rightarrow A$, $g$, $g' : B \rightarrow B$, we define the function $\text{dyn}$ as follows:

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

Example

Suppose $B$ is the set of natural numbers, $h = \text{id}$, $g = +1$; $g' = \ast 2$; $P = \text{"even"}$. If we take $x = 1$, $\text{dyn}$ will compute the infinite list: $2, 6, 14, 30, 62, 126, ...$ If $B$ is a set of streams, we can have $\text{dyn} = \text{filter}$. 

Katya (INRIA Sophia Antipolis)
Productive Functions

We call a function *productive at the input value* \( i \), if it outputs a productive value at \( i \).

**Definition**

Let \( A, B \) be of type \( \text{Set} \). For a predicate \( P : B \to \text{bool} \) and functions \( h : B \to A, g, g' : B \to B \), we define the function \( \text{dyn} \) as follows:

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

**Example**

Suppose \( B \) is the set of natural numbers, \( h = \text{id}, g = +1; g' = \ast 2; P = \text{“even”}. \) If we take \( x = 1 \), \( \text{dyn} \) will compute the infinite list: 2, 6, 14, 30, 62, 126, ...
Productive Functions

We call a function *productive at the input value* $i$, if it outputs a productive value at $i$.

**Definition**

Let $A, B$ be of type $\text{Set}$. For a predicate $P : B \rightarrow \text{bool}$ and functions $h : B \rightarrow A$, $g$, $g' : B \rightarrow B$, we define the function $\text{dyn}$ as follows:

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

**Example**

Suppose $B$ is the set of natural numbers, $h = \text{id}$, $g = +1$; $g' = \ast 2$; $P = \text{“even”}$. If we take $x = 1$, $\text{dyn}$ will compute the infinite list: 2, 6, 14, 30, 62, 126, ...

If $B$ is a set of streams, we can have $\text{dyn} = \text{filter}$. 
Totally Productive, Partially Productive, Non-Productive Functions

- **Totally Productive**
  (Function repeat)

- **Partially Productive**
  (Filters on streams and trees; dyn).

- **Non-Productive**
  Computing \(\text{nth 0 (filter even (repeat 1))}\) provokes the following computation:

  \[
  \text{filter even (repeat 1)} \text{ repeat 1} \leadsto \text{filter even (1::repeat 1)} \leadsto \text{filter even (repeat 1)}...
  \]
Totally Productive, Partially Productive, Non-Productive Functions

- **Totally Productive**
  (Function repeat)

- **Partially Productive**
  (Filters on streams and trees; dyn).

- **Non-Productive**
  Computing `nth 0 (filter even (repeat 1))` provokes the following computation:
  
  ```
  filter even (repeat 1) repeat 1 \rightarrow filter even
  (1::repeat 1) \rightarrow filter even (repeat 1)...
  ```

Our method makes it possible to formalise totally and partially productive functions in Coq, using inductive and coinductive predicates to characterise the arguments on which these functions output productive values.
A structurally recursive definition is such that every recursive call is performed on a structurally smaller argument.

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

In this way we can be sure that the recursion terminates.

**Example**

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

Definitions where the recursive calls are not required to be on structurally smaller arguments, that is, where the recursive calls can be performed on any argument, are called *general recursive* arguments.

Example

\[
\begin{align*}
\log(\text{S} 0) &= 0 \\
\log(\text{S}(\text{S} n)) &= \text{S}(\log \text{S}(\text{div2} n)).
\end{align*}
\]
General Recursion

Definitions where the recursive calls are not required to be on structurally smaller arguments, that is, where the recursive calls can be performed on any argument, are called *general recursive* arguments.

**Example**

\[
\begin{align*}
\log(S\ 0) &= 0 \\
\log(S(S\ n)) &= S(\log S(div2\ n)).
\end{align*}
\]
Method of Ad-hoc Predicates [Aczel77], [Bove02].

Fixpoint log (x:nat)(h:log_domain x){struct h} : nat :=
match x as y return x = y -> nat with
| 0 => fun h' => False
| S 0 => fun h' => 0
| S (S p) =>
  fun h' => S (log (S (div2 p)) (log_domain_inv x p h h'))
end (refl_equal x).
Method of Ad-hoc Predicates [Aczel77], [Bove02].

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

The guardedness condition insures that

* each corecursive call is made under at least one constructor;
** if the recursive call is under a constructor, it does not appear as an argument of any function.

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

The guardedness condition insures that

* each corecursive call is made under at least one constructor;
** if the recursive call is under a constructor, it does not appear as an argument of any function.

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

Example

Non-guarded functions:

[*] is not satisfied:
Filters, dyn;
[**] is not satisfied:
Consider the following function computing lists of ordered natural numbers:
nats = (SCons 1 (map (+ 1) nats)).
where the function map above is defined as follows:

map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
Non-guarded functions:

[*] is not satisfied:
Filters, dyn;

[**] is not satisfied:
Consider the following function computing lists of ordered natural numbers:

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

where the function map above is defined as follows:

```
map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
```
Non-guarded functions:

[*] is not satisfied:
Filters, dyn;

[**] is not satisfied:
Consider the following function computing lists of ordered natural numbers:
nats = (SCons 1 (map (+ 1) nats)).
where the function map above is defined as follows:

map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
Non-guarded functions:

[*] is not satisfied:
Filters, dyn;

[**] is not satisfied:
Consider the following function computing lists of ordered natural numbers:
\[ \text{nats} = (S\text{Cons } 1 \ (\text{map } (+ \ 1) \ \text{nats})). \]
where the function map above is defined as follows:

\[ \text{map } f \ (s: \text{str}): \text{str} \ := \text{Cons} \ (f \ (\text{hd } s)) \ (\text{map } f \ (\text{tl } s)). \]
The Method of Separation of Inductive and Coinductive Components

CoRecursive Function (Non-Guarded):

Inductive Component

ColInductive Component
The Method of Separation of Inductive and Coinductive Components

CoRecursive Function (Non-Guarded):

- Inductive Component
- Predicate eventually

- CoInductive Component
- Predicate infinite
The Method of Separation of Inductive and Coinductive Components

CoRecursive Function (Non-Guarded):

- Inductive Component
  - Predicate eventually

CoRecursive Function (Guarded):

- CoInductive Component
  - Predicate infinite
Predicate eventually: First-Step Productivity

Using eventually, we can describe the inductive component of a corecursive function. This component is a recursive function that performs all the computations and tests that lead to the first guarded corecursive call.

\[
\text{Inductive eventually} s : \text{str A} \rightarrow \text{Prop} := \\
| \text{ev}_b: \text{forall x s, P x} \rightarrow \text{eventually}_s (\text{SCons A x s}) \\
| \text{ev}_r: \text{forall x s, not P x} \rightarrow \text{eventually}_s s \rightarrow \text{eventually}_s (\text{SCons A x s}).
\]
Using eventually, we can describe the inductive component of a corecursive function. This component is a recursive function that performs all the computations and tests that lead to the first guarded corecursive call.

\[
\text{Inductive eventually}_s:\ \text{str A} \rightarrow \text{Prop} := \\
| \text{ev}_b:\ \forall x\ s, \ P\ x \rightarrow \text{eventually}_s\ (\text{SCons A x s}) \\
| \text{ev}_r:\ \forall x\ s, \ \neg P\ x \rightarrow \text{eventually}_s\ s \rightarrow \text{eventually}_s\ (\text{SCons A x s}).
\]
Predicate eventually: First-Step Productivity

Using eventually, we can describe the inductive component of a corecursive function. This component is a recursive function that performs all the computations and tests that lead to the first guarded corecursive call.

```
Inductive eventually_s: str A -> Prop :=
| ev_b: forall x s, P x -> eventually_s (SCons A x s)
| ev_r: forall x s, not P x -> eventually_s s -> eventually_s (SCons A x s).
```
Shape of eventually

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

1. The predicate \texttt{eventually} is defined inductively, with one constructor for each branch appearing in the function definition.

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

   \begin{verbatim}
   ev_dyn1 : P x = true  ->  eventually_dyn x
   \end{verbatim}
Shape of eventually

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

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

   \[ ev\_dyn1 : P\ x = true \rightarrow eventually\_dyn\ x \]

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

   \[ ev\_dyn2 : P\ x = false \rightarrow eventually\_dyn\ (g'\ x) \rightarrow eventually\_dyn\ x \]
Inversion Lemmas

Lemma eventually_s_inv:

forall (s : str A),
eventually_s s ->forall x s', s = SCons A x s' ->
not P x -> eventually_s s'.
Inversion Lemmas

**Lemma eventually\_s\_inv:**

\[
\text{forall } (s : \text{str } A), \text{ eventually}\_s s \rightarrow \forall x s', s = \text{SCons } A x s' \rightarrow \neg P x \rightarrow \text{eventually}\_s s'.
\]

**Lemma eventually\_dyn\_inv:**

\[
\text{forall } x, \text{ eventually}\_\text{dyn } x \rightarrow P x = \text{false} \rightarrow \text{eventually}\_\text{dyn } (g' x).
\]
Formalisation of Productive Non-Guarded Functions in Coq

Inductive Component of Corecursive Functions

Inductive Component of Filter

Fixpoint pre_filter_s (s : str A) (h : eventually_s s) struct h : A * str A :=

match s as b return s = b -> A*str A with
   SCons x s’ =>
   fun heq =>
   match P_dec x with
     | left _ => (x, s’)
     | right hn =>
   pre_filter_s s’ (eventually_s_inv s h x s’ heq hn)
end
end (refl_equal s).
Inductive Component of Filter

Fixpoint pre_filter_s (s : str A) (h : eventually_s s) struct h : A * str A :=
match s as b return s = b -> A*str A with
SCons x s’ =>
fun heq =>
match P_dec x with
| left _ => (x, s’)
| right hn =>
pre_filter_s s’ (eventually_s_inv s h x s’ heq hn)
end
end (refl_equal s).
Fixpoint pre_filter_s (s : str A) (h : eventually_s s) struct h : A * str A :=

match s as b return s = b -> A*str A with
SCons x s’ =>
fun heq =>
match P_dec x with
| left _ => (x, s’)
| right hn =>
pre_filter_s s’ (eventually_s_inv s h x s’ heq hn)
end
end (refl_equal s).
Inductive Component of \texttt{dyn}

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

\begin{verbatim}
Fixpoint \texttt{pre\_dyn}(x:B)(d:\texttt{eventually\_dyn }x)\{\texttt{struct } d\}: A*B:=
\begin{array}{l}
\text{match } P \ x \ \text{as } b \ \text{return } P \ x = b \ \rightarrow \ A*B\ \text{with} \\
\mid \ \text{true} \Rightarrow \ \text{fun } t \Rightarrow (h \ x, g \ x) \\
\mid \ \text{false} \Rightarrow \ \text{fun } t \Rightarrow \\
\text{pre\_dyn} \ (g' \ x) \ (\text{eventually\_dyn\_inv } x \ d \ t) \\
\text{end } (\text{refl\_equal } (P \ x)).
\end{array}
\end{verbatim}
Inductive Component of `dyn`.

```
Fixpoint pre_dyn(x:B)(d:eventually_dyn x){struct d}: A*B:=
  match P x as b return P x = b -> A*B with
  | true => fun t => (h x, g x)
  | false => fun t =>
    pre_dyn (g' x) (eventually_dyn_inv x d t)
  end (refl_equal (P x)).
```
Corecursive computations are introduced by repeating computations performed by the inductive component. This can happen only if recursive calls satisfy the eventually predicate repeatedly. We need the predicate infinite to express this.

\[
\text{CoInductive infinite_s : str -> Prop :=}
\]

\[
\begin{align*}
\text{al_cons: forall (s:str A) (h: eventually s),} \\
\text{infinite_s (snd(pre_filter_s s h)) -> infinite_s s.}
\end{align*}
\]
Corecursive computations are introduced by repeating computations performed by the inductive component. This can happen only if recursive calls satisfy the eventually predicate repeatedly. We need the predicate `infinite` to express this.

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

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

The `infinite` predicate describes exactly those arguments to the function for which the function is guaranteed to be productive.
Relating eventually and infinite

**Lemma** infinite\_eventually\_s :
for all s, infinite\_s s → eventually\_s s.

**Lemma** infinite\_eventually\_dyn :
for all x, infinite\_dyn x → eventually\_dyn x.
Relating eventually and infinite

Lemma infinite_eventually_s :
forall s, infinite_s s -> eventually_s s.

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

Lemma infinite_always_s :
forall (s:str A) (h:infinite_s s),
infinite_s (snd(pre_filter_s s(infinite_eventually_s s h))).

Lemma infinite_always_dyn :
forall x, infinite_dyn x -> forall e: eventually_dyn x,
infinite_dyn (snd (pre_dyn x e)).
Guarded Representation of a filter

CoFixpoint filter (s : str A) : forall (h: infinite_s s), str A :=

match s return infinite_s s -> str A with
  | SCons x s’ =>
    fun h’ : infinite_s (SCons A x s’) =>
    SCons A (fst (pre_filter_s _ infinite_eventually_s (SCons A x s’) h’)))
    (filter _ (infinite_always_s (SCons A x s’) h’))
end.
Guarded Representation of a filter

CoFixpoint filter (s : str A) : forall (h: infinite_s s), str A :=
match s return infinite_s s -> str A with
| SCons x s’ =>
fun h’ : infinite_s (SCons A x s’) =>
SCons A (fst
(pre_filter_s _ infinite_eventually_s (SCons A x s’) h’)))
(filter _ (infinite_always_s (SCons A x s’) h’))
end.
Guarded Representation of \texttt{dyn}

\begin{verbatim}
CoFixpoint dyn (x:B)(h:infinite_dyn x) : str :=
SCons (fst (pre_dyn x (infinite_eventually_dyn ev x h)))
  (dyn _ (infinite_always_dyn x h
  (infinite_eventually_dyn x h))).
\end{verbatim}
Guar ded Representation of dyn

CoFixpoint dyn (x:B)(h:infinite_dyn x) : str :=
SCons (fst (pre_dyn x (infinite_eventually_dyn ev x h)))
  (dyn _ (infinite_always_dyn x h
  (infinite_eventually_dyn x h))).
Recursive Equation Lemma for \textit{dyn}

\textbf{Theorem dyn\_equation :}

\begin{verbatim}
forall x \ i, \ bisimilar_s (dyn x \ i)
(match Px as b return Px = b -> infinite_dyn x -> str with
|true => fun t i => SCons(h x)(dyn(g x)(dyn_step1 x t i))
|false => fun t i => dyn (g' x) (dyn_step2 x t i)
end (refl_equal (P x)) i).
\end{verbatim}

\textbf{Lemma dyn\_step1 :}

\begin{verbatim}
forall x, P x = true -> infinite_dyn x ->
infinite_dyn (g x).
\end{verbatim}

\textbf{Lemma dyn\_step2 :}

\begin{verbatim}
forall x, P x = false -> infinite_dyn x ->
infinite_dyn (g' x).
\end{verbatim}
More Complicated Applications of Our Method:

Expression trees and dynamic filtering on expression trees.

The dynamic filter function on expression trees was used to establish a normalisation algorithm for an admissible representation of a closed interval of real numbers in [Geuvers1993], [Niqui2004].
More Complicated Applications of Our Method:

Expression trees and dynamic filtering on expression trees.

The dynamic filter function on expression trees was used to establish a normalisation algorithm for an admissible representation of a closed interval of real numbers in [Geuvers1993], [Niqui2004]. The function was not guarded.

We applied our method to give a Coq formalisation of the function.
Conclusions

1. We generalised the method of [Bertot05] to a wider class of functions:
   - various output data types including streams, expression and binary trees;
   - included dynamically changing functions.

   and thereby gave a general analysis of the method.

2. We work with partial productivity, and not just total productivity.

3. We establish the uniform **Recursive Equation Lemmas** for the functions we formalise, this was not achieved in [Bertot05].
Conclusions

1. We generalised the method of [Bertot05] to a wider class of functions:
   - various output data types including streams, expression and binary trees;
   - included dynamically changing functions.

   and thereby gave a general analysis of the method.

2. We work with partial productivity, and not just total productivity.

3. We establish the uniform Recursive Equation Lemmas for the functions we formalise, this was not achieved in [Bertot05].

Future work → further automatisation.
Thank you!