Coq in Mathematics and Computer Science: Inductive and Coinductive Capacities.

Ekaterina Komendantskaya

INRIA Sophia Antipolis

Marseille08, 06 May 2008
Outline

1. Coq in Mathematics and Computer Science
Outline

1. Coq in Mathematics and Computer Science
2. Inductive/Coinductive Types
Outline

1. Coq in Mathematics and Computer Science
2. Inductive/Coinductive Types
3. Terminative and Productive Functions
Outline

1. Coq in Mathematics and Computer Science
2. Inductive/Coinductive Types
3. Terminative and Productive Functions
4. Syntactic Approach to Termination: Structural Recursion and Guardedness
Outline

1. Coq in Mathematics and Computer Science
2. Inductive/Coinductive Types
3. Terminative and Productive Functions
4. Syntactic Approach to Termination: Structural Recursion and Guardedness
5. Formalisation of Productive Non-Guarded Functions in Coq
Outline

1. Coq in Mathematics and Computer Science
2. Inductive/Coinductive Types
3. Terminative and Productive Functions
4. Syntactic Approach to Termination: Structural Recursion and Guardedness
5. Formalisation of Productive Non-Guarded Functions in Coq
6. Conclusions
Coq in Mathematics and Computer Science

Coq is a proof assistant using dependent type system.
Coq is a proof assistant using dependent type system.

- Choice of Type Theory: Type theory presents a powerful formal system that captures both the notion of computation (via the inclusion of functional programs written in typed $\lambda$-calculus), and proof (via the “formulas as types embedding”, where types are viewed as propositions and terms as proofs).
Coq in Mathematics and Computer Science

Coq is a proof assistant using dependent type system.

- **Choice of Type Theory**: Type theory presents a powerful formal system that captures both the notion of *computation* (via the inclusion of functional programs written in typed $\lambda$-calculus), and *proof* (via the “formulas as types embedding”, where types are viewed as propositions and terms as proofs).

- **Dependent products** → additional expressivity makes possible to consider propositions about programs/proofs; or to construct certified programs that satisfy a given property. (E.g. `prime_divisor`, `binary_world`, ...)

Coq in Mathematics and Computer Science

Coq is a proof assistant using dependent type system.

- **Choice of Type Theory**: Type theory presents a powerful formal system that captures both the notion of computation (via the inclusion of functional programs written in typed $\lambda$-calculus), and proof (via the “formulas as types embedding”, where types are viewed as propositions and terms as proofs).

- **Dependent products $\Rightarrow$** additional expressivity makes possible to consider propositions about programs/proofs; or to construct certified programs that satisfy a given property. (E.g. prime_divisor, binary_world,...)

- **Proof assistant = proof checker + proof-development system.** (*not theorem prover*)
The goal: to increase reliability of mathematical results.

Mathematical results may be difficult to verify, because of:
The goal: to increase reliability of mathematical results.

Mathematical results may be difficult to verify, because of:

- **Complexity**: the problem is very big, the number of cases very large, etc.
  
  $\implies$ computer assistance is needed.
The goal: to increase reliability of mathematical results.

Mathematical results may be difficult to verify, because of:

- **Complexity**: the problem is very big, the number of cases very large, etc.
  \[\Rightarrow\] computer assistance is needed.

- **Depth**: the problem is very deep, complicated, complex methods from different disciplines are needed (eg, Fermat theorem).
  \[\Rightarrow\] machine assistance for doing mathematical research.
The goal: to increase reliability of mathematical results.

Mathematical results may be difficult to verify, because of:

- **Complexity:** the problem is very big, the number of cases very large, etc.
  ➞ computer assistance is needed.
- **Depth:** the problem is very deep, complicated, complex methods from different disciplines are needed (eg, Fermat theorem).
  ➞ machine assistance for doing mathematical research.

The former is the reality, the latter is a challenge.
Type-theoretic approach to proof-checking

Decidability of type checking = core of the type-theoretic theorem proving

In situation $\Gamma$ we have $A$.
Proof. $p$. 

$\Gamma \vdash_T p : A$ 

$\text{Type}_\Gamma(p) = A$
Type-theoretic approach to proof-checking

Decidability of type checking = core of the type-theoretic theorem proving

\[ \Gamma \vdash T \ p : A \]

Type\_\(\Gamma(\_\) is a function that finds for \(p\) a type in the given context \(\Gamma\). The decidability of type-checking follows from:

- \(\text{Type}_\Gamma(p)\) generates a type of \(p\) in context \(\Gamma\) or returns “false”.
- The equality = is decidable.
Three main decidability problems:

TCP: \( \Gamma \vdash TM \): "Does proof M indeed proves A?"

TSP: \( \Gamma \vdash TM \): "Is proof M a proof?"

TIP: \( \Gamma \vdash M \): "Is A provable?"

Both TCP and TSP are decidable.

provability of formula A = "inhabitation" of type A

proof checking = type checking

interactive theorem proving = i. construction of a term/given a type
Three main decidability problems:

TCP $\Gamma \vdash_T M : A$?

“Does proof $M$ indeed proves $A$?”
Three main decidability problems:

TCP $\Gamma \vdash_T M : A$?
“Does proof $M$ indeed proves $A$?”

TSP $\Gamma \vdash_T M : ?$
“Is proof $M$ a proof?”

Both TCP and TSP are decidable.

provability of formula $A$ = "inhabitation" of type $A$
proof checking = type checking
interactive theorem proving = i. construction of a term/given a type
Three main decidability problems:

TCP $\Gamma \vdash_T M : A$

“Does proof $M$ indeed proves $A$?”

TSP $\Gamma \vdash_T M : ?$

“Is proof $M$ a proof?”

TIP $\Gamma \vdash_T ? : A$

“Is $A$ provable?”
Three main decidability problems:

TCP  \( \Gamma \vdash_T M : A \)?
    “Does proof M indeed proves A?”

TSP  \( \Gamma \vdash_T M : ? \)
    “Is proof M a proof?”

TIP  \( \Gamma \vdash_T ? : A \)?
    “Is A provable?”

Both TCP and TSP are decidable.

provability of formula A  \( \equiv \) “inhabitation” of type A
proof checking  \( \equiv \) type checking
interactive theorem proving  \( \equiv \) i. construction of a term/given a type
Coq is the leading proof-assistant

Coq wins comparison with Agda, Lego, Nurpl, HOL, Isabelle, Mizar, ACL2, PVS in [Barendregt,Geuvers01].

**Parameters:**

**Presence of Proof Objects:** the script generates and stores a term that is isomorphic to a proof that can be checked on independent/simple proof checker. $\Rightarrow$ high reliability. (!)

**Reliability. (!)**

**Poincaré Principle** There is a distinction between computations and proofs; computations do not require a proof. (E.g. $1+0 = 1$ does not require a proof.) The principle is useful to deal with Conversion Rule. (!)

**Logic - Intuitionistic**

**Dependent Types (!)**

**Inductive Types (!)**
Marelle Team, INRIA, April 2008:

Majour:

- = limitis of pure functional programming: no computational effects (side effects, interactive input/output, exceptions,..);
- Proof checker and not prover (2 researchers);
- Syntactic restrictions: difficult to have different views/representations of one object;
- Constructive logic (??);
- Too much of expressiveness: Coq Art.
- Structural recursion, Guardedness…;
Limits of Coq?

Marelle Team, INRIA, April 2008:

Majour:

- Limitis of pure functional programming: no computational effects (side effects, interactive input/output, exceptions,..);
- Proof checker and not prover (2 researchers);
- Syntactic restrictions: difficult to have different views/representations of one object;
- Constructive logic (?);
- Too much of expressiveness: Coq Art.
- Structural recursion, Guardedness…;

Minor, technical hurdles:

- higher-order unification;
- deciding guardedness;
- need for a better organised documentation.
What is the one best thing about Coq?

Marelle Team, INRIA, April 2008:

- Mathematics and programming together; compute and prove simultaneously; \(\Rightarrow\) Research in Coq (3 researchers);
- Dependent types;
- Type theory \(\Rightarrow\) formal rigour;
- Implicit arguments, type inference.
- Extraction;
- Replication of proofs;
- Simple, uniform notation.
Successfull applications of Coq (http://coq.inria.fr/)

Mathematics

- Geometry,
- Set Theory,
- Algebra,
- Number theory,
- Category Theory,
- Domain theory,
- Real analysis and Topology,
- Probabilities.
**Successfull applications of Coq (http://coq.inria.fr/)**

---

<table>
<thead>
<tr>
<th>Mathematics</th>
<th>CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometry,</td>
<td>Infinite Structures,</td>
</tr>
<tr>
<td>Set Theory,</td>
<td>Pr. Lang.: Data Types and</td>
</tr>
<tr>
<td>Algebra,</td>
<td>Data Structures;</td>
</tr>
<tr>
<td>Number theory,</td>
<td>Pr. Lang.: Semantics and</td>
</tr>
<tr>
<td>Category Theory,</td>
<td>Compilation;</td>
</tr>
<tr>
<td>Domain theory,</td>
<td>Formal Languages Theory</td>
</tr>
<tr>
<td>Real analysis and Topology,</td>
<td>and Automata;</td>
</tr>
<tr>
<td>Probabilities.</td>
<td>Decision Procedures and</td>
</tr>
<tr>
<td></td>
<td>Certified Algorithms;</td>
</tr>
<tr>
<td></td>
<td>Concurrent Systems and</td>
</tr>
<tr>
<td></td>
<td>Protocols;</td>
</tr>
<tr>
<td></td>
<td>Operating Systems;</td>
</tr>
<tr>
<td></td>
<td>Biology and Bio-CS.</td>
</tr>
</tbody>
</table>
1 Coq in Mathematics and Computer Science

2 Inductive/Coinductive Types

3 Terminative and Productive Functions

4 Syntactic Approach to Termination: Structural Recursion and Guardedness

5 Formalisation of Productive Non-Guarded Functions in Coq

6 Conclusions
Inductive Types and Recursive Functions

Coq = COC [Coquand,Huet’88] + CIC [Coquand,Paulin’93]

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

Coq = COC ['88] + CIC ['93] + CCC [Gimenez'96]

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

CoFixpoint repeat (a: A): str A :=
  SCons a (repeat a).
Mixing Inductive and Coinductive types

See demo file...
1. Coq in Mathematics and Computer Science

2. Inductive/Coinductive Types

3. **Terminative and Productive Functions**

4. Syntactic Approach to Termination: Structural Recursion and Guardedness

5. Formalisation of Productive Non-Guarded Functions in Coq

6. Conclusions
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.
Termination

We require all computations to terminate, because of:

- Curry-Howard Isomorphism (propositions → types; proofs → 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 → types; proofs → 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:

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

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.
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:

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

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 \textit{productive at the input value} \( i \), if it outputs a productive value at \( i \).

Example
Filter is productive only on certain inputs.
Filter Function, [Bertot05]

**Definition**

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

$$
\text{filter} \ (\text{SCons} \ x \ \text{tl}) = \begin{cases} 
\text{SCons} \ x \ (\text{filter} \ \text{tl}) & \text{if } P(x) \\
\text{filter} \ \text{tl} & \text{otherwise}
\end{cases}
$$
A more general example

**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 $\text{dyn}$:

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

**Example**

Suppose $B$ is the set of natural numbers, $h = \text{id}$, $g = +1$; $g' = \ast 2$; $P = \"even\"$. 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 $\text{dyn} = \text{filter}$. 
A more general example

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 $\text{dyn}$:

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

Example

Suppose $B$ is the set of natural numbers, $h = \text{id}$, $g = +1$; $g' = \times 2$; $P =$ “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 $\text{dyn} = \text{filter}$. 
A more general example

**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 \( \text{dyn} \):

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

**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-, Partially-, 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 ⇝ filter even (1::repeat 1) ⇝ filter even (repeat 1)
  ... 
  ```
Totally-, Partially-, 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:

  \[
  \text{filter even (repeat 1) repeat 1} \rightsquigarrow \text{filter even (1::repeat 1)} \rightsquigarrow \text{filter even (repeat 1)} ...
  \]

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.
1. Coq in Mathematics and Computer Science

2. Inductive/Coinductive Types

3. Terminative and Productive Functions

4. Syntactic Approach to Termination: Structural Recursion and Guardedness

5. Formalisation of Productive Non-Guarded Functions in Coq

6. Conclusions
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
| O => 0
| S O => 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**

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

\[
\begin{aligned}
\log(S\ 0) &= 0 \\
\log(S(S\ n)) &= S(\log\ S(\text{div2}\ n)).
\end{aligned}
\]
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).
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.
Guardedness [Gimenez96: Calculus of Coinductive Constructions]

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:
\[ \text{nats} = (\text{SCons}\ 1\ (\text{map}\ (+\ 1)\ \text{nats})). \]
where the function \text{map} above is defined as follows:

\[ \text{map}\ f\ (\text{s: str}) : \text{str} := \text{Cons}\ (f\ (\text{hd}\ s))\ (\text{map}\ f\ (\text{tl}\ s)). \]
Non-guarded functions:

[*] is not satisfied:
Filters, dyn;

[**] is not satisfied:
Consider the following function computing lists of ordered natural numbers:
\[
\text{nats = (SCons 1 (map (+ 1) nats))},
\]
where the function \text{map} above is defined as follows:
\[
\text{map f (s: str): str := Cons (f (hd s)) (map f (tl s))}.
\]
Non-guarded functions:

[*] is not satisfied:
Filters, dyn;

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

\[ \text{nats} = (\text{SCons} \ 1 \ (\text{map} \ (+ \ 1) \ \text{nats})) \]

where the function \( \text{map} \) above is defined as follows:

\[ \text{map} \ f \ (s: \text{str}): \text{str} := \text{Cons} \ (f \ (\text{hd} \ s)) \ (\text{map} \ f \ (\text{tl} \ s)) \].
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)).
1. Coq in Mathematics and Computer Science

2. Inductive/Coinductive Types

3. Terminative and Productive Functions

4. Syntactic Approach to Termination: Structural Recursion and Guardedness

5. Formalisation of Productive Non-Guarded Functions in Coq

6. Conclusions
Our Method: Inductive and Coinductive Components

CoRecursive Function (Non-Guarded):

Inductive Component

CoInductive Component
Our Method: Inductive and Coinductive Components

CoRecursive Function (Non-Guarded):

Inductive Component

Predicate eventually

CoInductive Component

Predicate infinite
Our Method: Inductive and Coinductive Components

CoRecursive Function (Non-Guarded):

Inductive Component

Predicate eventually

CoInductive Component

Predicate infinite

CoRecursive Function (Guarded):
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.

\[
\text{Inductive eventually}_s : \text{str} \ A \to \text{Prop} := \\
\quad \text{ev}_b : \forall x s, P x \to \text{eventually}_s (\text{SCons} A x s) \\
\quad \text{ev}_r : \forall x s, \neg P x \\
\quad \to \text{eventually}_s s \to \text{eventually}_s (\text{SCons} A x s).
\]
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.

\begin{align*}
\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, \ \text{not } P x \\
\rightarrow & \ \text{eventually}_s s \rightarrow \text{eventually}_s (\text{SCons} \ A \ x \ s).
\end{align*}
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.

\[
\text{Inductive eventually} \_s : \text{str} \ A \ \rightarrow \ \text{Prop} :=
\]
\[
\mid \text{ev}_b : \forall x \ s, \ P \ x \ \rightarrow \ \text{eventually} \_s \ (\text{SCons} \ A \ x \ s) \\
\mid \text{ev}_r : \forall x \ s, \ \text{not} \ P \ x \\
\rightarrow \ \text{eventually} \_s \ s \ \rightarrow \ \text{eventually} \_s \ (\text{SCons} \ A \ x \ s).
\]
Eventually for \( \text{dyn} \)

\[
\text{eventually} \, \text{dyn} \, (x : B) : \text{Prop} :=
\begin{align*}
\text{ev} \, \text{dyn1} & : P \, x = \text{true} \to \text{eventually} \, \text{dyn} \, x \\
\text{ev} \, \text{dyn2} & : P \, x = \text{false} \to \text{eventually} \, \text{dyn} \, (g' \, x) \to \\
\text{eventually} \, \text{dyn} \, x.
\end{align*}
\]

Compare \text{eventually} with \( \diamond \) in [Pnueli81, Jacobs02].
Inversion Lemmas

**Lemma eventually\_s\_inv:**

\[
\text{forall } (s : \text{str } A), \\
\text{eventually}_s s \rightarrow \text{forall } x s', s = \text{SCons } A x s' \rightarrow \\
\text{not } P x \rightarrow \text{eventually}_s s'.
\]
Inversion Lemmas

**Lemma eventually_s_inv:**

\[
\text{forall } (s : \text{str } A), \\
\text{eventually}_s s \rightarrow \text{forall } x s', s = \text{SCons } A x s' \rightarrow \\
\text{not } P x \rightarrow \text{eventually}_s s'.
\]

**Lemma eventually_dyn_inv :**

\[
\text{forall } x, \text{eventually}_\text{dyn} x \rightarrow P x = \text{false} \rightarrow \\
\text{eventually}_\text{dyn} (g' x).
\]
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).
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 \( \text{dyn} \)

Fixpoint \( \text{pre_dyn}(x:B)(\text{d:eventually_dyn } x)\{\text{struct } d\}: A*B:= \)

\[
\begin{align*}
&\text{match } P\ x\ \text{as } b\ \text{return } P\ x = b \rightarrow A*B\ \text{with} \\
&\mid \text{true } \rightarrow \text{fun } t \Rightarrow (h\ x, g\ x) \\
&\mid \text{false } \rightarrow \text{fun } t \Rightarrow \\
&\text{pre_dyn}\ (g'\ x)\ (\text{eventually_dyn_inv } x\ d\ t) \\
&\text{end}\ (\text{refl_equal } (P\ x)). \\
\end{align*}
\]
Inductive Component of \textit{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 \text{dyn}

\begin{Verbatim}
\text{Fixpoint pre\_dyn(x:B)(d:eventually\_dyn x){struct d}}:\text{A*B:=}
\text{match P x as b return P x = b -> A*B with}
| \text{true} => \text{fun t => (h x, g x)}
| \text{false} => \text{fun t =>}
\text{pre\_dyn (g' x) (eventually\_dyn\_inv x d t)}
\text{end (refl\_equal (P x)).}
\end{Verbatim}
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.

```coq
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.
```

Katya (INRIA Sophia Antipolis)

Coq in Mathematics and Computer Science: Inductive and Coinductive Capacities.

Marseille08 42 / 56
The same predicate for $\text{dyn}$

\[
\text{CoInductive } \infty_{\text{dyn}} (x : B): \text{Prop} :=
\]
\[
\begin{align*}
\text{di} &: \forall (d : \text{eventually}_\text{dyn} x), \\
\infty_{\text{dyn}} (\text{snd} (\text{pre}_\text{dyn} x d)) &\rightarrow \infty_{\text{dyn}} x.
\end{align*}
\]

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_dyn :
forall x, infinite_dyn x -> eventually_dyn x.

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 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))).
Guarded Representation of dyn

\[
\text{CoFixpoint } \text{dyn} \ (x:B)(h:\text{infinite\_dyn} \ x) : \ \text{str} := \\
\text{SCons} \ (\text{fst} \ (\text{pre\_dyn} \ x \ (\text{infinite\_eventually\_dyn} \ \text{ev} \ x \ h)))
\]
\[
(\text{dyn} \ _ \ (\text{infinite\_always\_dyn} \ x \ h \\
(\text{infinite\_eventually\_dyn} \ x \ h))).
\]
Recursive Equation Lemma for \(\text{dyn}\)

\[
\text{Theorem dyn_equation :}
\]

\[
\forall x \; \text{infinite_dyn } x, \; \text{bisimilar_s } (\text{dyn } x \; i)
\]

\[
(\text{match } \text{P} x \; \text{as } b \; \text{return } \text{P} x = b \rightarrow \text{infinite_dyn } x \rightarrow \text{str})
\]

\[
\text{with}
\]

\[
| \text{true} \Rightarrow \text{fun } t \; i \Rightarrow 
\]

\[
\text{SCons}(h \; x)(\text{dyn}(g \; x)(\text{dyn_step1 } x \; t \; i))
\]

\[
| \text{false} \Rightarrow \text{fun } t \; i \Rightarrow \text{dyn } (g' \; x)(\text{dyn_step2 } x \; t \; i)
\]

\[
\text{end } \; (\text{refl_equal } (\text{P } x)) \; i).
\]
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.
Other Productive Non-Guarded Functions

What other productive non-guarded functions do we know?

Every terminative function gives rise to a non-guarded totally productive function.

Example
Terminative function \( x \rightarrow 1 \) gives rise to the totally productive function \( f: \text{stream} \text{ nat} \rightarrow \text{stream} \text{ nat}: \)

\[
f(x::y::\text{tl}) = 
\begin{cases} 
  x::f(y::\text{tl}) & \text{if } x \leq y \\
  f((x-1)::y::\text{tl}) & \text{otherwise.}
\end{cases}
\]
What other productive non-guarded functions do we know?

Every terminative function gives rise to a non-guarded totally productive function.

Example

Terminative function $x - 1$ gives rise to the totally productive function $f: \text{stream nat} \rightarrow \text{stream nat}$:

$$
f (x::y::tl) = \begin{cases} 
  x::f(y::tl) & \text{if } x \leq y \\
  f((x-1)::y::tl) & \text{otherwise.}
\end{cases}
$$
1 Coq in Mathematics and Computer Science

2 Inductive/Coinductive Types

3 Terminative and Productive Functions

4 Syntactic Approach to Termination: Structural Recursion and Guardedness

5 Formalisation of Productive Non-Guarded Functions in Coq

6 Conclusions
Conclusions

1. Coq is very useful but has limits.
2. We work on eliminating one of such limits - syntactic restrictions.
3. We generalised the method of [Bertot’05] to:
   - various output data types including streams, expression and binary trees;
   - included dynamically changing functions;
and thereby gave a general analysis of the method.
4. We work with partial productivity, and not just total productivity.
5. We establish the uniform Recursive Equation Lemmas for the functions we formalise, this was not achieved in [Bertot05].
Conclusions

1. Coq is very useful but has limits.
2. We work on eliminating one of such limits - syntactic restrictions.
3. We generalised the method of [Bertot’05] to:
   - various output data types including streams, expression and binary trees;
   - included dynamically changing functions;
and thereby gave a general analysis of the method.
4. We work with partial productivity, and not just total productivity.
5. We establish the uniform Recursive Equation Lemmas for the functions we formalise, this was not achieved in [Bertot05].

Future work → further automatisation.
Thank you!