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

Ekaterina Komendantskaya

INRIA Sophia Antipolis

Marseille08,
06 May 2008

Outline

Outline

Outline

Outline

Outline

Outline

Coq in Mathematics and Computer Science

Coq is a proof assistant using dependent type system.

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 λ-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 λ-calculus), and *proof* (via the "formulas as types embedding", where types are viewed as propositions and terms as proofs).
- Dependent products $\Longrightarrow$ 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 λ-calculus), and *proof* (via the "formulas as types embedding", where types are viewed as propositions and terms as proofs).
- Dependent products $\Longrightarrow$ 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.
  $\Longrightarrow$ 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.
  $\Longrightarrow$ computer assistance is needed.
- Depth: the problem is very deep, complicated, complex methods from different disciplines are needed (eg, Fermat theorem).
  $\Longrightarrow$ machine assistance for doing mathematical research.

The goal: to increase reliability of mathematical results.

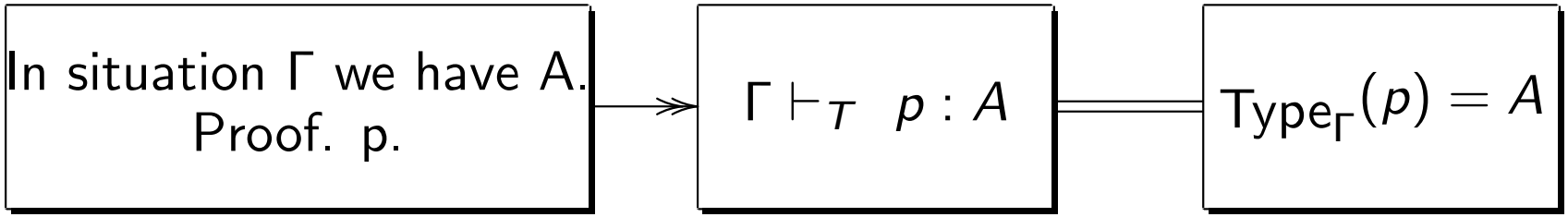
Mathematical results may be difficult to verify, because of:

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

The former is the reality, the latter is a challenge.

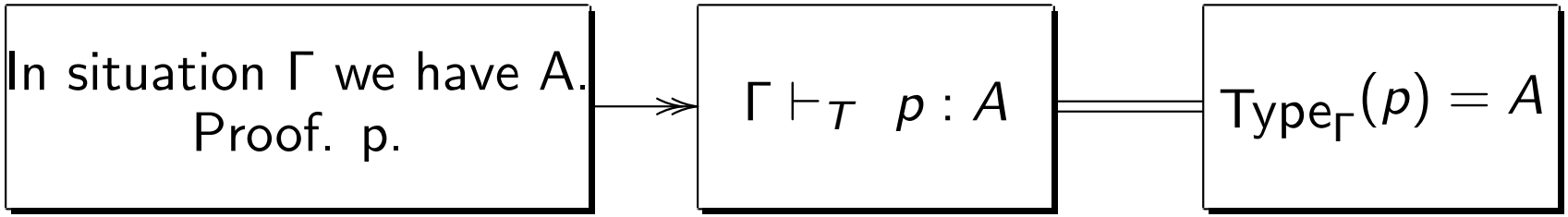
Type-theoretic approach to proof-checking

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



Type-theoretic approach to proof-checking

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



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

- $\mathrm{Type}_\Gamma(p)$ generates a type of p in context Γ or returns "false".
- The equality $=$ is decidable.

Three main decidability problems:

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?"

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	=	"inhabitation" of type A
proof checking	=	type checking
interactive theorem proving	=	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. $\Longrightarrow$ 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 (!)

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, Guardedeness...;

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, Guardedeness...;

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 simultaneiously; $\Longrightarrow$ Research in Coq (3 researchers);
- Dependent types;
- Type theory $\Longrightarrow$ 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/)

Mathematics

- Geometry,
- Set Theory,
- Algebra,
- Number theory,
- Category Theory,
- Domain theory,
- Real analysis and Topology,
- Probabilities.

CS

- Infinite Structures,
- Pr. Lang.: Data Types and Data Structures;
- Pr. Lang.: Semantics and Compilation;
- Formal Languages Theory and Automata;
- Decision Procedures and Certified Algorithms;
- Concurrent Systems and Protocols;
- Operating Systems;
- Biology and Bio-CS.

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...

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 $\rightarrow$ 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:

$$\begin{cases} \texttt{nth 0 (SCons a tl)} = \texttt{a} \\ \texttt{nth (S n) (SCons a tl)} = \texttt{nth n tl} \end{cases}$$

A given stream `s` is productive if we can be sure that the computation of the list `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:

$$\begin{cases} \texttt{nth 0 (SCons a tl)} = \texttt{a} \\ \texttt{nth (S n) (SCons a tl)} = \texttt{nth n tl} \end{cases}$$

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

Example

For any `n`, the value `repeat n` is productive.

Productive Functions

We call a function *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,

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

A more general example

Definition

Let A, B be of type `Set`. For a predicate $P : B \to \texttt{bool}$ and functions $h : B \to A$, g, $g' : B \to B$, we define the function *dyn*:

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

A more general example

Definition

Let A, B be of type `Set`. For a predicate $P : B \to \texttt{bool}$ and functions $h : B \to A$, g, $g' : B \to B$, we define the function *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, $h = id$, $g = +1$; $g' = *2$; $P =$ "even". If we take $x = 1$, `dyn` will compute the infinite list:
2, 6, 14, 30, 62, 126, ...

A more general example

Definition

Let A, B be of type `Set`. For a predicate $P : B \to \texttt{bool}$ and functions $h : B \to A$, g, $g' : B \to B$, we define the function *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, $h = id$, $g = +1$; $g' = *2$; $P =$ "even". If we take $x = 1$, `dyn` will compute the infinite list:
2, 6, 14, 30, 62, 126, ...
If B is a set of streams, we can have `dyn` = 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` $\rightsquigarrow$ `filter even (1::repeat 1)` $\rightsquigarrow$ `filter even (repeat 1)`...

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` $\rightsquigarrow$ `filter even (1::repeat 1)` $\rightsquigarrow$ `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.

Structural Recursion

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.

Structural Recursion

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

```
Fixpoint div2 n :  nat :=
match n with
| O => O
| S O => O
| 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

$$\begin{cases} \log(\mathrm{S}\ 0) = 0 \\ \log(\mathrm{S}(\mathrm{S}\ n)) = \mathrm{S}(\log\ \mathrm{S}(\mathrm{div2}\ n)). \end{cases}$$

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

$$\begin{cases} \log(\mathrm{S}\ 0) = 0 \\ \log(\mathrm{S}(\mathrm{S}\ n)) = \mathrm{S}(\log\ \mathrm{S}(\mathrm{div2}\ n)). \end{cases}$$

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).
```

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 [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.

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

```
CoFixpoint repeat (a:A): str A := SCons a (repeat a).
```

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:

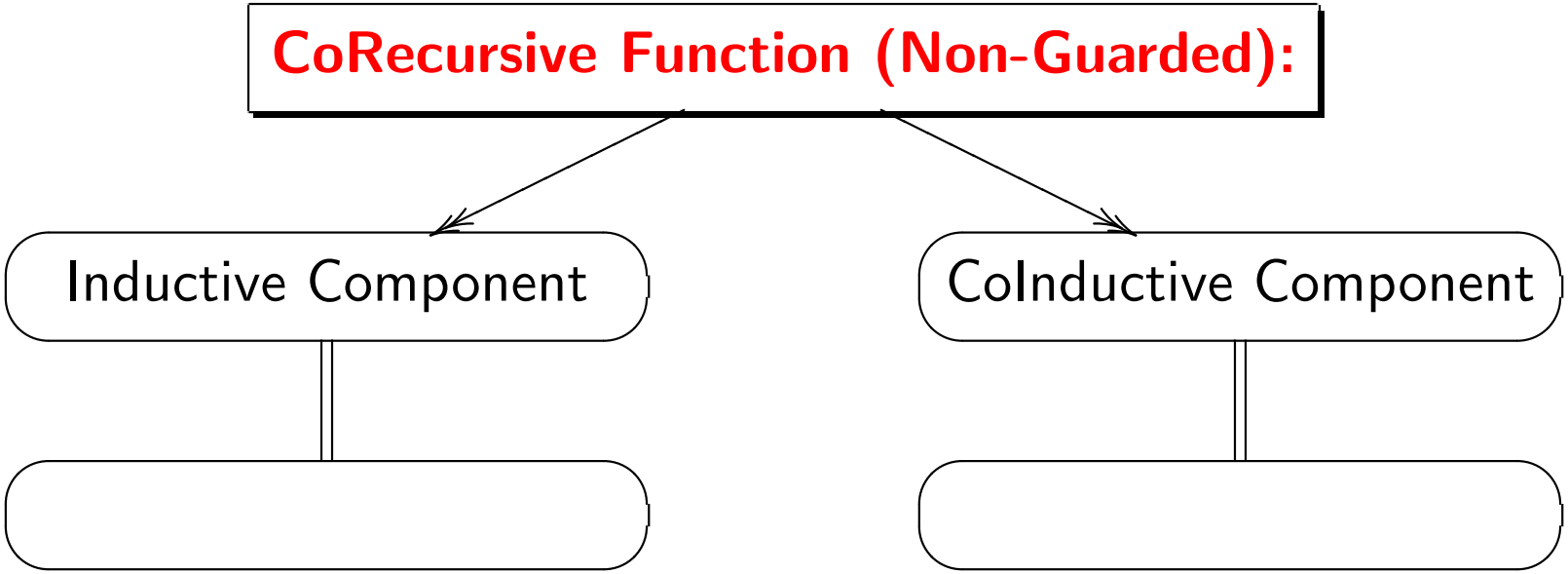
[$*$] is not satisfied:
Filters, `dyn`;
[$**$] is not satisfied:
Consider the following function computing lists of ordered natural numbers:

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

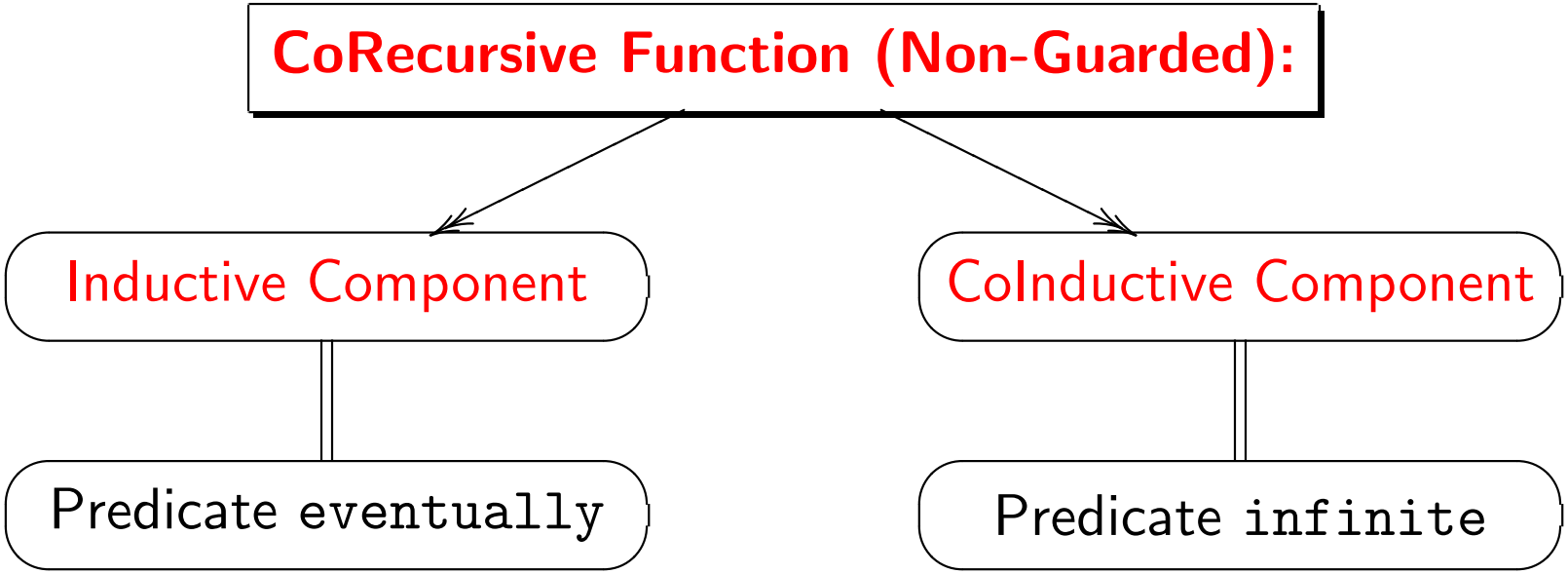
where the function `map` above is defined as follows:

```
map f (s: str): str := Cons (f (hd s)) (map f (tl s)).
```

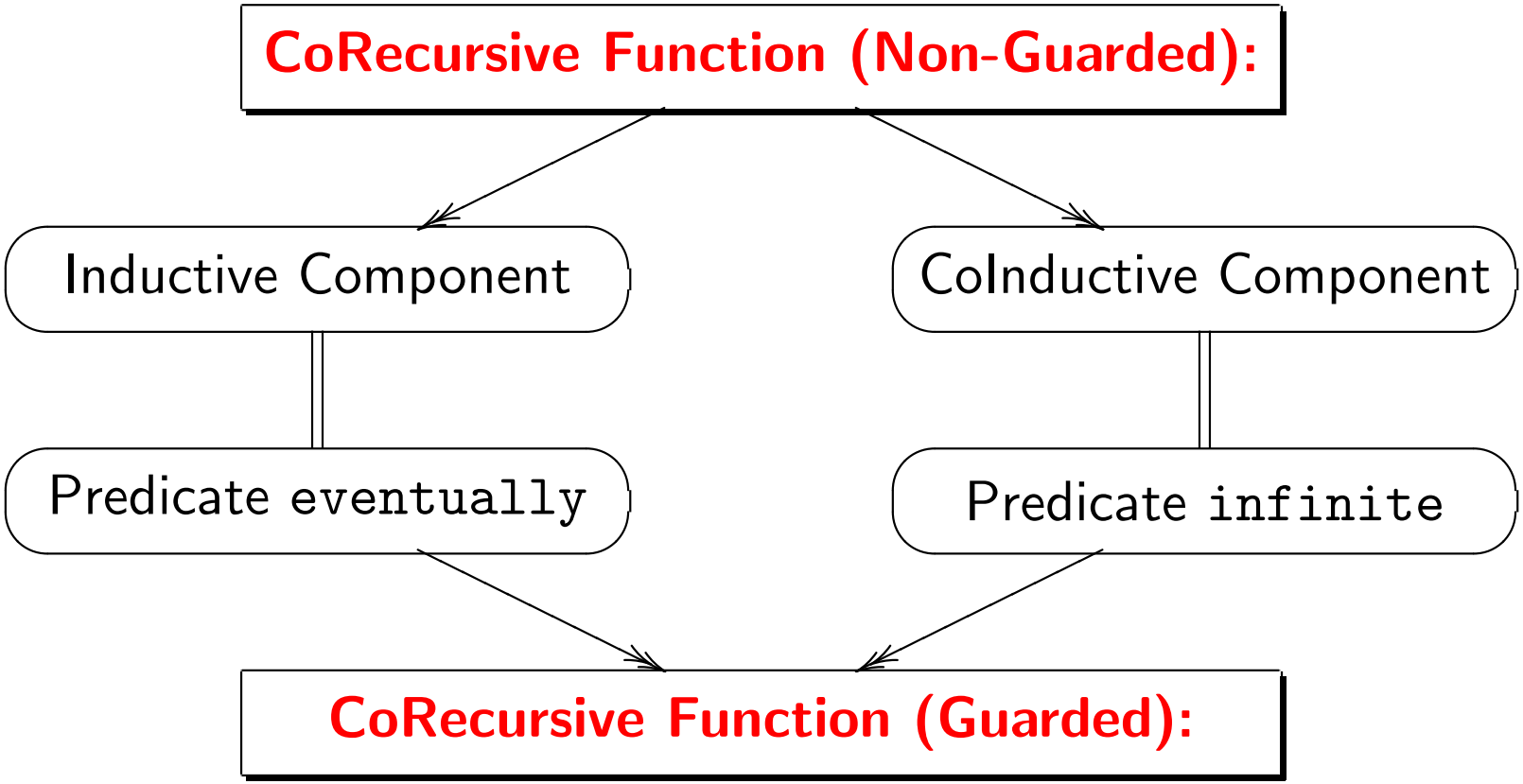
Our Method: Inductive and Coinductive Components



Our Method: Inductive and Coinductive Components



Our Method: Inductive and Coinductive Components



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).
```

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).
```

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).
```

eventually for dyn

```
Inductive eventually_dyn (x:  B) : Prop :=
| ev_dyn1 :  P x = true -> eventually_dyn x
| ev_dyn2 :  P x = false -> eventually_dyn (g' x) ->
eventually_dyn x.
```

Compare `eventually` with $\Diamond$ in [Pnueli81, Jacobs02].

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

```
Lemma eventually_dyn_inv :
forall x, eventually_dyn x -> P x = false ->
eventually_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 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)).
```

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)).
```

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)).
```

Coinductive Predicate infinite

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.
```

The same predicate for dyn

```
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_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

```
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 dyn

```
Theorem dyn_equation :
forall x i: infinite_dyn x, 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).
```

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?

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 $x - 1$ gives rise to the totally productive function `f: stream nat -> stream nat:`

$$\texttt{f (x::y::tl)} = \begin{cases} \texttt{x::f(y::tl)} & \text{if } x \leq y \\ \texttt{f((x-1)::y::tl)} & \text{otherwise.} \end{cases}$$

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 $\rightarrow$ further automatisation.

Thank you!