Type-Based Methods for Termination and Productivity in Coq

Bruno Barras\textsuperscript{1}  Jorge Luis Sacchini\textsuperscript{2}

\textsuperscript{1}INRIA Saclay & LIX
\textsuperscript{2}Carnegie Mellon University – Qatar

July 22, 2013
Coq

- Coq is a **total** dependently-typed programming language
- **Totality** means:
  - Functions must be defined in their entire domain (no partial functions)
  - Recursive functions must be **terminating**
  - Co-recursive functions must be **productive**
- Non-terminations leads to inconsistencies
  - Ex: \((\text{let } f \ x = f \ x \text{ in } f \ 0) : 0 = 1\)
- Totality ensures logical consistency and decidability of type checking
Coq

- Termination and productivity are undecidable problems
- Approximate the answer
- Coq imposes **syntactic restrictions** on (co-)recursive definitions
- For termination: guarded-by-destructors
- Recursive calls performed only on **structurally smaller terms**

\[
\Gamma(f : I \rightarrow T) \vdash M : I \rightarrow T
\]

\[
\frac{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \rightarrow T}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \rightarrow T}
\]
Coq

- Termination and productivity are undecidable problems
- Approximate the answer
- Coq imposes *syntactic restrictions* on (co-)recursive definitions
- For termination: guarded-by-destructors
- Recursive calls performed only on *structurally smaller terms*

\[
\Gamma(f : I \rightarrow T) \vdash M : I \rightarrow T \\
\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \rightarrow T
\]

- The predicate \( G(f, M) \) checks that all recursive calls of \( f \) in \( M \) are guarded by destructors
Termination and productivity are undecidable problems
Approximate the answer
Coq imposes syntactic restrictions on (co-)recursive definitions
For termination: guarded-by-destructors
Recursive calls performed only on structurally smaller terms

\[
\begin{align*}
\Gamma(f : I \to T) & \vdash M : I \to T & G(f, M) \\
\Gamma & \vdash (\text{fix } f : I \to T := M) : I \to T
\end{align*}
\]

The predicate \(G(f, M)\) checks that all recursive calls of \(f\) in \(M\) are guarded by destructors
Actually, the guard condition is checked on a normal form of the body

\[
\begin{align*}
\Gamma(f : I \to T) & \vdash M : I \to T & M \rightarrow^* N & G(f, N) \\
\Gamma & \vdash (\text{fix } f : I \to T := M) : I \to T
\end{align*}
\]
Termination in Coq

- Typical example:

```coq
fix half : nat → nat := λx. case x of
  | O ⇒ O
  | S O ⇒ O
  | S (S p) ⇒ S(half p)
```

Recursive call is **guarded**. The recursive argument is smaller.

- The initial implementation of $G$ (due to Eduardo Giménez around 1994) has been extended over the years to allow more functions.

- Most recent extension: commutative cuts (due to Pierre Boutillier).
Termination in Coq

- Typical example:

```coq
fix half : nat → nat := λx. case x of
  | O ⇒ O
  | S O ⇒ O
  | S (S p) ⇒ S(half p)
```

Recursive call is **guarded**. The recursive argument is smaller.

- The initial implementation of $\mathcal{G}$ (due to Eduardo Giménez around 1994) has been extended over the years to allow more functions.

- Most recent extension: commutative cuts (due to Pierre Boutillier).
Termination in Coq

Subterm relation

Subtraction:

\[
\text{fix } \textit{minus} : \textit{nat} \rightarrow \textit{nat} \rightarrow \textit{nat} := \lambda xy. \ \text{case } x, y \text{ of} \\
| O, _ \Rightarrow x \\
| S x_1, O \Rightarrow S x_1 \\
| S x_1, S y_1 \Rightarrow \textit{minus } x_1 \ y_1
\]
Termination in Coq

Subterm relation

Subtraction:

\[
\text{fix } \text{minus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} := \lambda xy. \text{case } x, y \text{ of} \\
| \text{O, } _{-} \Rightarrow x \\
| S x_1, \text{O} \Rightarrow S x_1 \\
| S x_1, S y_1 \Rightarrow \text{minus } x_1 y_1
\]

\(x_1 \prec x\) \((x_1\text{ is a strict subterm of } S x_1 \equiv x)\)
Termination in Coq

Subterm relation

Subtraction:

\[
\text{fix } \text{minus} : \text{nat} \to \text{nat} \to \text{nat} := \lambda xy. \text{case } x, y \text{ of }
\]
\[
| \ O, \_ \Rightarrow x \n| \ S \ x_1, O \Rightarrow S \ x_1 \n| \ S \ x_1, S \ y_1 \Rightarrow \text{minus} \ x_1 \ y_1
\]

\(x_1 \prec x\) \((x_1 \text{ is a strict subterm of } S \ x_1 \equiv x)\)

Division:

\[
\text{div} \ x \ y = \left\lfloor \frac{x}{y+1} \right\rfloor
\]

\[
\text{fix } \text{div} : \text{nat} \to \text{nat} \to \text{nat} := \lambda xy. \text{case } x \text{ of }
\]
\[
| \ O \Rightarrow O \n| \ S \ x_1 \Rightarrow S(\text{div}(\text{minus} \ x_1 \ y) \ y)
\]
Termination in Coq

Subterm relation

Subtraction:

```
fix minus : nat → nat → nat := λxy. case x, y of
  | O, _ ⇒ x
  | S x₁, O ⇒ S x₁
  | S x₁, S y₁ ⇒ minus x₁ y₁
```

\[ x₁ \prec x \text{ (} x₁ \text{ is a strict subterm of } S x₁ ≡ x \text{)} \]

Division: \[ \text{div } x \; y = \left\lfloor \frac{x}{y+1} \right\rfloor \]

```
fix div : nat → nat → nat := λxy. case x of
  | O ⇒ O
  | S x₁ ⇒ S(div(minus x₁ y) y)
```

\[ \text{minus } x₁ \; y \preceq x₁ \prec S x₁ ≡ x \]
Termination in Coq

Subterm relation

Subtraction:

\[\text{fix } \text{minus} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} := \lambda x, y. \text{case } x, y \text{ of}\]
\[| \text{O, } - \Rightarrow x \]
\[| \text{S } x_1, \text{O} \Rightarrow \text{S } x_1 \]
\[| \text{S } x_1, \text{S } y_1 \Rightarrow \text{minus } x_1 y_1 \]

Division: \[\text{div } x y = \left\lfloor \frac{x}{y+1} \right\rfloor\]

\[\text{fix } \text{div} : \mathbb{N} \to \mathbb{N} \to \mathbb{N} := \lambda x, y. \text{case } x \text{ of}\]
\[| \text{O} \Rightarrow \text{O} \]
\[| \text{S } x_1 \Rightarrow \text{S}(\text{div}(\text{minus } x_1 y) y) \]
Termination in Coq

Subterm relation

Subtraction:

\[
\text{fix } \text{minus} : \text{n}at \to \text{n}at \to \text{n}at \ := \ \lambda x, y. \ \text{case } x, y \ \text{of}
\]
\[
| \ O, _ \ \Rightarrow \ O \\
| \ S x_1, O \ \Rightarrow \ S x_1 \\
| \ S x_1, S y_1 \ \Rightarrow \ \text{minus } x_1 \ y_1
\]

\(x_1 \prec x (x_1 \text{ is a strict subterm of } S x_1 \equiv x)\)

Division: \(\text{div } x \ y = \left\lfloor \frac{x}{y+1} \right\rfloor\)

\[
\text{fix } \text{div} : \text{n}at \to \text{n}at \to \text{n}at \ := \ \lambda x, y. \ \text{case } x \ \text{of}
\]
\[
| \ O \ \Rightarrow \ O \\
| \ S x_1 \ \Rightarrow \ S(\text{div } (\text{minus } x_1 \ y) \ y)
\]
Termination in Coq

Subterm relation

Subtraction:

\[ \text{fix } \text{minus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} := \lambda xy. \text{case } x, y \text{ of} \]
\[ \quad | \quad O, _ \Rightarrow O \]
\[ \quad | \quad S x_1, O \Rightarrow S x_1 \]
\[ \quad | \quad S x_1, S y_1 \Rightarrow \text{minus} x_1 y_1 \]

\( x_1 \prec x \) (\( x_1 \) is a strict subterm of \( S x_1 \equiv x \))

Division: \( \text{div } x \ y = \left\lceil \frac{x}{y+1} \right\rceil \)

\[ \text{fix } \text{div} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} := \lambda xy. \text{case } x \text{ of} \]
\[ \quad | \quad O \Rightarrow O \]
\[ \quad | \quad S x_1 \Rightarrow S(\text{div} (\text{minus} x_1 y) y) \]

\( \text{minus } x_1 y \not\prec x_1 \prec S x_1 \equiv x \)
Inductive rose(A) : Type := node : A → list (rose A) → rose A

rmap := λf : A → B. fix rmap : rose A → rose B :=
λt. case t of
  node x ts ⇒ node (f x) (map rmap ts)

map := λf : A → B. fix map : list A → list B :=
λl. case l of
  nil ⇒ nil
  nil ⇒ nil
  cons x xs ⇒ cons (f x) (map xs)
Inductive rose(A) : Type := node : A → list (rose A) → rose A

rmap := λf : A → B. fix rmap : rose A → rose B := λt. case t of
  node x ts ⇒ node (f x) (map rmap ts)

map := λf : A → B. fix map : list A → list B := λl. case l of
  nil ⇒ nil
  cons x xs ⇒ cons (f x) (map xs)
Inductive rose(A) : Type := node : A → list (rose A) → rose A

rmap := λf : A → B. fix rmap : rose A → rose B :=
    λt. case t of
    node x ts ⇒ node (f x) (map rmap ts)

map := fix map : (A → B) → list A → list B :=
    λf l. case l of
    nil ⇒ nil
    cons x xs ⇒ cons (f x) (map f xs)
Inductive rose(A) : Type := node : A → list (rose A) → rose A

rmap := λf : A → B. fix rmap : rose A → rose B :=
    λt. case t of
    node x ts ⇒ node (f x) (map rmap ts)

map := fix map : (A → B) → list A → list B :=
    λf l. case l of
    nil ⇒ nil
    cons x xs ⇒ cons (f x) (map f xs)
Syntactic criteria

Limitations

- Works on syntax: small changes in code can make functions ill-typed
- Not compositional
- Difficult to understand for users
  - Many questions about termination in the Coq list
  - Error messages not informative
- Difficult to implement: termination checking is the most delicate part of Coq's kernel
- Inefficient: guard condition is checked on the normal form of fixpoints bodies
- Difficult to study
  - Little documentation
  - Complicated to even define
Termination in Coq

- Many ways to get around the guard condition:
  - Adding extra argument to act as measure of termination
  - Wellfounded recursion
  - Ad-hoc predicate (Bove)
  - Tool support (Function, Program)
- But this complicates function definition
- May affect efficiency
Termination using sized types

- Long history: Haskell [Pareto et al.], \( \lambda^\hat{\cdot} \) [Joao Frade et al.], \( F_\omega^\hat{\cdot} \) [Abel], \( \text{CIC}^\hat{\cdot} \) [Barthe et al.], CC+rewriting [Blanqui et al.] …
Termination using sized types

- Long history: Haskell [Pareto et al.], \( \lambda^\wedge \) [Joao Frade et al.], \( F_\omega^\wedge \) [Abel], CIC\(^\wedge \) [Barthe et al.], CC+rewriting [Blanqui et al.] ...

- Basic idea: user-defined datatypes are decorated with size information

\[
\text{nat} ::= O : \text{nat} \mid S : \text{nat} \rightarrow \text{nat}
\]

Intuitive meaning: \([\text{nat}] = \{O, S \ O, S(S \ O), \ldots\}\)
Termination using sized types

- Basic idea: user-defined datatypes are decorated with size information

\[
nat ::= O : \text{nat} \mid S : \text{nat} \to \text{nat}
\]

Intuitive meaning: \([\text{nat}] = \{O, S \ O, S(\ S\ O), \ldots\}\)

- Sized types are approximations

\[
nat{\langle s \rangle}
\]

Intuitive meaning: \([\text{nat} \langle s \rangle] = \{O, S \ O, \ldots, S(\ldots(S\ O)\ldots)\}_{s-1}\)
Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \tau \mid \hat{s} \mid \infty$$
Termination using sized types

- Size annotations keep track of the size of elements.

\[ s ::= \nu \mid \hat{s} \mid \infty \]

\[ \hat{\infty} = \infty \]
Termination using sized types

- Size annotations keep track of the size of elements

\[ s ::= \nu \mid \hat{s} \mid \infty \]

\[ \Gamma \vdash O : \text{nat} \quad \Gamma \vdash M : \text{nat} \]

\[ \Gamma \vdash S M : \text{nat} \]
Termination using sized types

- Size annotations keep track of the size of elements

\[ s ::= n \mid \hat{s} \mid \infty \]

\[ \Gamma \vdash O : \text{nat}(\hat{s}) \]
\[ \Gamma \vdash M : \text{nat}(s) \]
\[ \Gamma \vdash S M : \text{nat}(\hat{s}) \]
Termination using sized types

- Size annotations keep track of the size of elements

\[ s ::= \nu \mid \hat{s} \mid \infty \]

\[ \Gamma \vdash O : \text{nat}\langle \hat{s} \rangle \]
\[ \Gamma \vdash M : \text{nat}\langle s \rangle \]
\[ \Gamma \vdash S M : \text{nat}\langle \hat{s} \rangle \]
Termination using sized types

- Size annotations keep track of the size of elements

\[ s ::= \nu \mid \hat{s} \mid \infty \]

\[
\Gamma \vdash O : \text{nat}\langle \hat{s} \rangle \\
\Gamma \vdash M : \text{nat}\langle s \rangle \\
\Gamma \vdash S M : \text{nat}\langle \hat{s} \rangle
\]

- Substage relation

\[
s \sqsubseteq \hat{s} \quad s \sqsubseteq \infty
\]

defines a subtype relation

\[
s \sqsubseteq r \\
\text{nat}\langle s \rangle \leq \text{nat}\langle r \rangle
\]
Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

\[ \Gamma(f : I \to T) \vdash M : I \to T \]
\[ \Gamma \vdash (\text{fix } f : I \to T := M) : I \to T \]
Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

\[
\frac{\Gamma(f : I \langle i \rangle \rightarrow T) \vdash M : I \langle \hat{i} \rangle \rightarrow T}{\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \langle s \rangle \rightarrow T} \quad i \text{ fresh}
\]

- Recursive calls on terms of smaller size
Recursive functions are defined on approximations of datatypes:

\[ \Gamma(f : \langle i \rangle \rightarrow T) \vdash M : \langle \hat{i} \rangle \rightarrow T \]
\[ \Gamma \vdash (\text{fix } f : \langle i \rangle \rightarrow T := M) : \langle s \rangle \rightarrow T \]

- Recursive calls on terms of smaller size
- Size-preserving functions: return type \( T \) can depend on \( i \)
Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

\[\Gamma(f : I \langle \nu \rangle \rightarrow T) \vdash M : I \langle \hat{\nu} \rangle \rightarrow T\]

\[\Gamma \vdash (\text{fix } f : I \rightarrow T := M) : I \langle s \rangle \rightarrow T\]

- Recursive calls on terms of smaller size
- Size-preserving functions: return type \( T \) can depend on \( \nu \)
- Non-structural recursion
Example: quicksort

Non-structural recursion

\[
\text{filter} \equiv \ldots : \Pi A. (A \rightarrow \text{bool}) \rightarrow \text{list} \quad A \rightarrow \text{list} \quad A \times \text{list} \quad A
\]

\[
(++) \equiv \ldots : \Pi A. \text{list} \quad A \rightarrow \text{list} \quad A \rightarrow \text{list} \quad A
\]
Example: quicksort
Non-structural recursion

\[
\text{filter} \equiv \ldots : \forall A. (A \rightarrow \text{bool}) \rightarrow \text{list } A \rightarrow \text{list } A \times \text{list } A
\]

\[
(\++) \equiv \ldots : \forall A. \text{list } A \rightarrow \text{list } A \rightarrow \text{list } A
\]

**fix qsort : list A \rightarrow list A :=**

\[
\lambda x : \text{list } A. \text{ case } x \text{ of }
\]

\[
| \text{nil} \Rightarrow \text{nil} \\
| \text{cons } h \ t \Rightarrow \text{let } (s, g) = \text{filter } (< h) \ t \text{ in } (\text{qsort } s) \ldots (\text{qsort } g) \ldots ) \\
\]
Example: quicksort
Non-structural recursion

\[
\text{filter} \equiv \ldots : \prod A. (A \to \text{bool}) \to \text{list}(s) A \to \text{list}(s) A \times \text{list}(s) A
\]

\[
(++) \equiv \ldots : \prod A. \text{list}(s) A \to \text{list}(r) A \to \text{list}(\infty) A
\]

\[
\text{fix qsort} : \text{list} A \to \text{list} A := \\
\lambda x : \text{list} A. \text{case } x \text{ of} \\
\quad | \text{nil} \Rightarrow \text{nil} \\
\quad | \text{cons } h t \Rightarrow \text{let } (s, g) = \text{filter } (< h) t \text{ in} \\
\quad \quad (\text{qsort } s) ++ (\text{cons } h (\text{qsort } g))
\]
Example: quicksort
Non-structural recursion

\[
\text{filter} \equiv \ldots : \Pi A. (A \to \text{bool}) \to \text{list}(s) A \to \text{list}(s) A \times \text{list}(s) A
\]

\[
(++) \equiv \ldots : \Pi A. \text{list}(s) A \to \text{list}(r) A \to \text{list}(\infty) A
\]

\[
\text{fix qsort : list } A \to \text{list } A :=
\]

\[
\lambda x : \text{list } A. \ \text{case } x^{\text{list}(\hat{\imath})} \ \text{of}
\]

\[
| \text{nil} \Rightarrow \text{nil}
\]

\[
| \text{cons } h \text{ t}^{\text{list}(\hat{\imath})} \Rightarrow \text{let } (s, g) = \text{filter } (< h) t^{\text{list}(\hat{\imath})} \ \text{in}
\]

\[
(q\text{sort } s^{\text{list}(\hat{\imath})}) ++ (\text{cons } h \ (q\text{sort } g^{\text{list}(\hat{\imath})}))
\]

Bruno Barras, Jorge Luis Sacchini
Example: quicksort
Non-structural recursion

\[
\text{filter} \equiv \ldots : \Pi A. (A \rightarrow \text{bool}) \rightarrow \text{list}\langle s\rangle A \rightarrow \text{list}\langle s\rangle A \times \text{list}\langle s\rangle A \\
(++) \equiv \ldots : \Pi A. \text{list}\langle s\rangle A \rightarrow \text{list}\langle r\rangle A \rightarrow \text{list}\langle \infty\rangle A
\]

fix qsort : list A \rightarrow list A :=
\lambda x : \text{list} A. \text{case } x^{\text{list}\langle \hat{i}\rangle} \text{ of}
| \text{nil} \Rightarrow \text{nil}
| \text{cons } h t^{\text{list}\langle i\rangle} \Rightarrow \text{let } (s, g) = \text{filter } (< h) t^{\text{list}\langle i\rangle} \text{ in}
\quad (\text{qsort } s^{\text{list}\langle i\rangle}) + + (\text{cons } h (\text{qsort } g^{\text{list}\langle i\rangle})))
:\Pi A. \text{list}\langle s\rangle A \rightarrow \text{list}\langle \infty\rangle A
Type-based termination

- Handle higher-order data

\[ \text{node} : \Pi A. A \rightarrow \text{list}(\infty)(\text{rose}(s) A) \rightarrow \text{rose}(\hat{s}) A \]

- Advantages over syntactic criteria
  - Expressiveness
  - Compositional
  - Easier to understand (specially for ill-typed terms)
  - Easier to implement (as shown in prototype implementations)
  - Easier to study (semantically intuitive)
  - Not intrusive for the user (minimal annotations required)

- Good candidate to replace syntactic criterion in Coq
Coinductive Types

- Coinductive types are used to model and reason about infinite data and infinite processes.
- Coinductive types can be seen as the dual of inductive types.

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Induction</td>
<td>Coinduction</td>
</tr>
<tr>
<td>Recursive functions</td>
<td>Corecursive functions</td>
</tr>
<tr>
<td>consume data</td>
<td>produce data</td>
</tr>
</tbody>
</table>

Bruno Barras, Jorge Luis Sacchini
Coinductive Types in Coq

- Streams:

\[
\text{CoInductive } \text{stream } A := \text{scons : } A \rightarrow \text{stream } A \rightarrow \text{stream } A
\]
Coinductive Types in Coq

- Stream

CoInductive stream A := scons : A \rightarrow stream A \rightarrow stream A
Coinductive Types in Coq

- Streams:

  \textbf{CoInductive} \text{ stream } A := \text{scons} : A \rightarrow \text{stream } A \rightarrow \text{stream } A

- Corecursive functions produce streams:

  zeroes := cofix \ Z := \text{scons}(0, \ Z)

  zeroes produce the stream:

  \text{scons}(0, \text{scons}(0, \text{scons}(0, \ldots))))
Coinductive types

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
</tbody>
</table>

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is productivity.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:

\[
\text{cofix } Z_1 := \text{scons}(0, Z_1) \\
\text{cofix } Z_2 := \text{scons}(0, \text{tail } Z_2)
\]
Coinductive types

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
</tbody>
</table>

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is productivity.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:

  \[
  \text{cofix } Z_1 := \text{scons}(0, Z_1) \\
  \text{cofix } Z_2 := \text{scons}(0, \text{tail } Z_2)
  \]
Coinductive types

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
</tbody>
</table>

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is productivity.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:

  \[
  \text{cofix } Z_1 := \text{scons}(0, Z_1) \quad \checkmark \\
  \text{cofix } Z_2 := \text{scons}(0, \text{tail } Z_2) \quad \times
  \]
Coinductive types

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
</tbody>
</table>

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is **productivity**.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:

\[
\text{cofix } Z_1 := \text{scons}(0, Z_1) \\
\text{cofix } Z_2 := \text{scons}(0, \text{tail } Z_2)
\]

(tail \(Z_2\) loops)

Bruno Barras, Jorge Luis Sacchini
## Syntactic-Based Methods for Productivity

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
<tr>
<td>Guarded-by-Destructor</td>
<td>Guarded-by-Constructor</td>
</tr>
</tbody>
</table>

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case
Syntactic-Based Methods for Productivity

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
<tr>
<td>Guarded-by-Destructor</td>
<td>Guarded-by-Constructor</td>
</tr>
</tbody>
</table>

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

\[
nats := \text{cofix } nats := \lambda n. \text{scons}(n, nats(1 + n))
\]
Syntactic-Based Methods for Productivity

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
<tr>
<td>Guarded-by-Destructor</td>
<td>Guarded-by-Constructor</td>
</tr>
</tbody>
</table>

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

\[
nats := \text{cofix} \quad nats := \lambda n. \text{scons}(n, nats(1 + n))
\]
## Syntactic-Based Methods for Productivity

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
<tr>
<td>Guarded-by-Destructor</td>
<td>Guarded-by-Constructor</td>
</tr>
</tbody>
</table>

- **Guarded-by-constructor:** every corecursive call is performed directly under a constructor
- **Same limitations as in the inductive case**

```lambda
let nats := cofix in nats := \n. scons(n, nats (1 + n))
```

```lambda
let nats := \n. cofix in nats := scons(n, map (\x. 1+x) nats)
```
### Syntactic-Based Methods for Productivity

<table>
<thead>
<tr>
<th><strong>Inductive types</strong></th>
<th><strong>Coinductive types</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
<tr>
<td>Guarded-by-Destructor</td>
<td>Guarded-by-Constructor</td>
</tr>
</tbody>
</table>

- **Guarded-by-constructor**: every corecursive call is performed directly under a constructor
- **Same limitations as in the inductive case**

\[
\text{nats} := \cofix \text{nats} := \lambda n. \text{scons}(n, \text{nats} (1 + n)) \quad \checkmark
\]

\[
\text{nats} := \lambda n. \cofix \text{nats} := \text{scons}(n, \text{map } (\lambda x. 1+x) \text{nats})
\]
Syntactic-Based Methods for Productivity

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
<tr>
<td>Guarded-by-Destructor</td>
<td>Guarded-by-Constructor</td>
</tr>
</tbody>
</table>

- Guarded-by-constructor: every corecursive call is performed directly under a constructor
- Same limitations as in the inductive case

\[
\text{nats} := \text{cofix nats} := \lambda n. \text{scons}(n, \text{nats} (1 + n)) \quad \checkmark
\]

\[
\text{nats} := \lambda n. \text{cofix nats} := \text{scons}(n, \text{map} (\lambda x. 1+x) \text{nats}) \quad \times
\]
Sized types can be applied to productivity checking as well!
Sized types can be applied to productivity checking as well!

Dual meaning of size annotations on coinductive types

\[
\text{stream}\langle s \rangle A
\]

is the type of streams of which \textit{at least} \( s \) elements can be produced
Sized types can be applied to productivity checking as well!

- Dual meaning of size annotations on coinductive types

\[
\text{stream}\langle s \rangle \, A
\]

is the type of streams of which \textit{at least} \( s \) elements can be produced.

- Size annotations are contra-variant:

\[
\frac{r \sqsubseteq s}{\text{stream}\langle s \rangle \, T \leq \text{stream}\langle r \rangle \, T}
\]
Sized types can be applied to productivity checking as well!

- Dual meaning of size annotations on coinductive types
  \[
  \text{stream}\langle s \rangle \ A
  \]
  is the type of streams of which \textbf{at least} \( s \) elements can be produced

- Size annotations are contra-variant:
  \[
  \begin{align*}
  r \sqsubseteq s & \quad \Rightarrow \quad \text{stream}\langle s \rangle \ T \leq \text{stream}\langle r \rangle \ T \\
  s \sqsubseteq r & \quad \Rightarrow \quad \text{list}\langle s \rangle \ T \leq \text{list}\langle r \rangle \ T
  \end{align*}
  \]
Type-Based Methods for Productivity

- Typing rules are similar to the inductive case
- Rules for constructors:

\[ \Gamma \vdash M : A \quad \Gamma \vdash N : \text{stream} \langle s \rangle A \]
\[ \Gamma \vdash \text{scons}(M, N) : \text{stream} \langle \hat{s} \rangle A \]

- Cofixpoint definition is also similar to fixpoint definition:

\[ \Gamma(f : \text{stream} \langle i \rangle A) \vdash M : \text{stream} \langle \hat{i} \rangle A \]
\[ \Gamma \vdash \text{cofix } f := M : \text{stream} \langle s \rangle A \quad \hat{i} \text{ fresh} \]
Type-Based Methods for Productivity

- Typing rules are similar to the inductive case
- Rules for constructors:

\[
\Gamma \vdash M : A \quad \Gamma \vdash N : \text{stream}\langle s \rangle A \\
\Gamma \vdash \text{scons}(M, N) : \text{stream}\langle \hat{s} \rangle A \\
\Gamma \vdash M : A \quad \Gamma \vdash N : \text{list}\langle s \rangle A \\
\Gamma \vdash \text{cons}(M, N) : \text{list}\langle \hat{s} \rangle A
\]

- Cofixpoint definition is also similar to fixpoint definition:

\[
\Gamma(f : \text{stream}\langle i \rangle A) \vdash M : \text{stream}\langle \hat{i} \rangle A \\
\Gamma \vdash \text{cofix } f := M : \text{stream}\langle s \rangle A \\
\hat{i} \text{ fresh}
\]

Bruno Barras, Jorge Luis Sacchini
Type-Based Methods for Productivity

- Typing rules are similar to the inductive case
- Rules for constructors:

\[
\begin{align*}
\Gamma \vdash M : A & \quad \Gamma \vdash N : \text{stream}\langle s \rangle A \\
\Gamma \vdash \text{scons}(M, N) : \text{stream}\langle \hat{s} \rangle A
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash M : A & \quad \Gamma \vdash N : \text{list}\langle s \rangle A \\
\Gamma \vdash \text{cons}(M, N) : \text{list}\langle \hat{s} \rangle A
\end{align*}
\]

- Cofixpoint definition is also similar to fixpoint definition:

\[
\begin{align*}
\Gamma(f : \text{stream}\langle i \rangle A) \vdash M : \text{stream}\langle \hat{i} \rangle A \\
\Gamma \vdash \text{cofix}\ f := M : \text{stream}\langle s \rangle A & \quad i \text{ fresh}
\end{align*}
\]

\[
\begin{align*}
\Gamma(f : \text{list}\langle i \rangle A \rightarrow U) \vdash M : \text{list}\langle \hat{i} \rangle A \rightarrow U \\
\Gamma \vdash \text{fix}\ f := M : \text{list}\langle s \rangle A \rightarrow U & \quad i \text{ fresh}
\end{align*}
\]
Co-recursive definitions

Examples

\[
\begin{align*}
\text{map} : (A \to B) \to \text{stream} & \quad A \to \text{stream} \quad B \\
\text{merge} : \text{stream} \quad \text{nat} \to \text{stream} \quad \text{nat} & \to \text{stream} \quad \text{nat} \\
\text{merge} (1 3 5 \ldots) & (2 4 6 \ldots) = (1 2 3 4 \ldots) \\
\text{ham} := \text{cofix} \quad \text{ham} : \text{stream} \quad \text{nat} := \\
& \quad \text{scons}(1, \text{merge} (\text{map} (\lambda x. 2 \times x) \text{ham} \quad ) \\
& \quad \quad (\text{merge} (\text{map} (\lambda x. 3 \times x) \text{ham} \quad ) \\
& \quad \quad \quad (\text{map} (\lambda x. 5 \times x) \text{ham} \quad ))) \\
\text{ham} = (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 9 \ 10 \ 12 \ 15 \ldots)
\end{align*}
\]
Co-recursive definitions

Examples

\[
\text{map} : (A \to B) \to \text{stream}\langle s\rangle A \to \text{stream}\langle s\rangle B
\]

\[
\text{merge} : \text{stream}\langle s\rangle \text{nat} \to \text{stream}\langle s\rangle \text{nat} \to \text{stream}\langle s\rangle \text{nat}
\]

\[
\text{merge} \ (1 \ 3 \ 5 \ldots) \ (2 \ 4 \ 6 \ldots) = (1 \ 2 \ 3 \ 4 \ldots)
\]

\[
\text{ham} := \text{cofix} \ \text{ham} : \text{stream} \ \text{nate} :=
\]

\[
\text{scons}(1, \ \text{merge} \ (\text{map} \ (\lambda x. 2 \times x) \ \text{ham} )
\]

\[
(\text{merge} \ (\text{map} \ (\lambda x. 3 \times x) \ \text{ham} )
\]

\[
(\text{map} \ (\lambda x. 5 \times x) \ \text{ham} )))
\]

\[
\text{ham} = (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 8 \ 9 \ 10 \ 12 \ 15 \ldots)
\]
Co-recursive definitions

Examples

map : (A → B) → stream<s>A → stream<s>B

merge : stream<s>nat → stream<s>nat → stream<s>nat
merge (1 3 5 . . .) (2 4 6 . . .) = (1 2 3 4 . . .)

ham := cofix ham : stream nat :=
  scons(1, merge (map (λx. 2*x) ham<stream<s>>)
         (merge (map (λx. 3*x) ham<stream<s>>) )
         (map (λx. 5*x) ham<stream<s>>)))

ham = (1 2 3 4 5 6 8 9 10 12 15 . . .)
Sized types for coinduction

- Type-based productivity has several advantages over syntactic-based
  - More expressive
  - Compositional
  - Easier to understand (specially for ill-typed terms)
  - Easier to implement (as shown in prototype implementations)
  - Easier to study (semantically intuitive)
  - Not intrusive for the user (minimal annotations required)

- Furthermore, sized types treat inductive and co-inductive types in a similar way
What’s next?
What’s next?

- Design a type-based termination system for Coq
- Implementation!
What’s next?

- Design a type-based termination system for Coq Implementation!
- Sombrero line (Barthe et al.) : $\lambda^\wedge, F^\wedge, \text{CIC}^\wedge$
- Sizes are declared implicitly (not first class):
  - Size inference: little burden for the user
    - Constraint-based algorithm
    - Treats fixpoints and co-fixpoints in the same way
- Still some issues remain in order to adapt to full Coq
In a future Coq version . . .

Fixpoint map \( f : A \rightarrow B \) \((xs : List\langle \_ \rangle A) : List\langle \_ \rangle B :=
    match xs with
      nil \Rightarrow nil
    cons h t \Rightarrow cons (f h) (map f t)
    end.
In a future Coq version . . .

Fixpoint map ℓ (f : A -> B) (xs : List<ℓ> A) : List<ℓ> B :=
    match xs with
      nil        => nil
    | cons h t   => cons (f h) (map f t)
    end.

Check map.
map : ∀ ℓ. (A -> B) -> List<ℓ> A -> List<ℓ> B.
Fixpoint map \( \diamond \) (f : A -> B) (xs : List\(<\diamond>\) A) : List\(<\diamond>\) B :=
  match xs with
  nil       => nil
  cons h t  => cons (f h) (map f t)
end.

Check map.
map : \( \forall \diamond. (A -> B) -> List\(<\diamond>\) A -> List\(<\diamond>\) B.

Fixpoint ntail \( \diamond \) A (x : nat\(<\diamond>\)) : List A -> List A :=
  ...

In a future Coq version . . .
In a future Coq version ...

Fixpoint map (f : A -> B) (xs : List A) : List B :=
  match xs with
  nil => nil
  cons h t => cons (f h) (map f t)
end.

Check map.
map : \forall A. (A -> B) -> List A -> List B.

Fixpoint ntail (x : nat) : List A -> List A :=
  ...

Check ntail.
In a future Coq version . . .

Fixpoint map (f : A -> B) (xs : List A) : List B :=
    match xs with
    nil => nil
    cons h t => cons (f h) (map f t)
end.

Check map.
map : \forall i. (A -> B) -> List A -> List B.

Fixpoint ntail (x : nat) : List A -> List A :=
    ...

Check ntail.
n tail : \forall i \forall j. forall A, nat -> List A -> List A.
In a future Coq version . . .

Fixpoint map \( f : A \to B \) (xs : List\(<i>\) A) : List\(<i>\) B :=
  match xs with
  | nil => nil
  | cons h t => cons (f h) (map f t)
end.

Check map.
map : \( \forall \ i. \ (A \to B) \to \text{List}\(<i>\) A \to \text{List}\(<i>\) B. \)

Fixpoint ntail \( \iota \ A \ (x : \text{nat}\(<i>\>) \) : \text{List} \ A \to \text{List} \ A :=
  ...

Check ntail.
ntail : \( \forall \ i \ \forall \ J_1 \ \forall \ J_2. \ J_2 \sqsubseteq J_1 \Rightarrow \)
  forall A, nat\(<i>\) \to List\(<j_1>\) A \to List\(<j_2>\) A.
Summary

- Keep extending the guard condition is not sustainable
- Time is right to rethink termination checking in Coq
- Sized types seem to be an ideal candidate
  - More expressive
  - Compositional
  - Easier to study and implement
Summary

- Keep extending the guard condition is not sustainable
- Time is right to rethink termination checking in Coq
- Sized types seem to be an ideal candidate
  - More expressive
  - Compositional
  - Easier to study and implement

- Project to start at CMU-Q in September
  - Careful design before implementation
Summary

- Keep extending the guard condition is not sustainable
- Time is right to rethink termination checking in Coq
- Sized types seem to be an ideal candidate
  - More expressive
  - Compositional
  - Easier to study and implement
- Project to start at CMU-Q in September
  - Careful design before implementation
- Is this an opportunity to rethink coinduction in Coq?
Summary

- Keep extending the guard condition is not sustainable
- Time is right to rethink termination checking in Coq
- Sized types seem to be an ideal candidate
  - More expressive
  - Compositional
  - Easier to study and implement

- Project to start at CMU-Q in September
  - Careful design before implementation

- Is this an opportunity to rethink coinduction in Coq?

Thank you!
A note on coinduction with dependent types

- Coinduction in Coq is broken: it does not satisfy type preservation
- The problem: cofixpoint unfolding is only allowed inside case analysis
  \[
  \text{case (cofix } f := M \text{) of . . . } \rightarrow \text{ case } M[f := (\text{cofix } f := M)] \text{ of . . .}
  \]
- Already observed by Giménez in 1996
- Some promising ideas: OTT (McBride) and copatterns (Abel et al.)
A note on coinduction with dependent types

- Example: consider a co-inductive type $U$ with only one constructor $\text{in} : U \rightarrow U$

  
  \[
  \begin{align*}
  u &: U \\
  \text{force} &: U \rightarrow U \\
  \end{align*}
  \]

  \[
  \begin{align*}
  u &\overset{\text{def}}{=} \text{cofix } u \coloneqq \text{in } u \\
  \text{force} &\overset{\text{def}}{=} \lambda x. \text{case } x \text{ of } \text{in } x' \Rightarrow \text{in } x'
  \end{align*}
  \]

- We can prove that $x = \text{force } x$ for any $x : U$

  \[
  \begin{align*}
  \text{eq} &: \prod x : U. x = \text{force } x \\
  \text{eq} &\overset{\text{def}}{=} \lambda x. \text{case } x \text{ of } \text{in } x' \Rightarrow \text{refl}
  \end{align*}
  \]

- Then, $\text{eq } u : u = \text{force } u$, 

Example: consider a co-inductive type $U$ with only one costructor
\[ \text{in} : U \rightarrow U \]

\[
\begin{align*}
u & : U \\
\text{force} & : U \rightarrow U \\
u \overset{\text{def}}{=} \text{cofix } u := \text{in } u & \quad \text{force} \overset{\text{def}}{=} \lambda x \cdot \text{case } x \text{ of in } x' \Rightarrow \text{in } x'
\end{align*}
\]

We can prove that $x = \text{force } x$ for any $x : U$

\[
\begin{align*}
\text{eq} & : \prod x : U. x = \text{force } x \\
\text{eq} \overset{\text{def}}{=} \lambda x \cdot \text{case } x \text{ of in } x' \Rightarrow \text{refl}
\end{align*}
\]

Then, $\text{eq } u : u = \text{in } u$, 
A note on coinduction with dependent types

- Example: consider a co-inductive type $U$ with only one constructor $\text{in} : U \rightarrow U$

  $$u : U \quad \text{force} : U \rightarrow U$$

  $$u \overset{\text{def}}{=} \text{cofix } u := \text{in } u \quad \text{force} \overset{\text{def}}{=} \lambda x. \text{case } x \text{ of in } x' \Rightarrow \text{in } x'$$

- We can prove that $x = \text{force } x$ for any $x : U$

  $$\text{eq} : \Pi x : U. x = \text{force } x$$

  $$\text{eq} \overset{\text{def}}{=} \lambda x. \text{case } x \text{ of in } x' \Rightarrow \text{refl}$$

- Then, $\text{eq } u : u = \text{in } u$, and $\text{eq } u \rightarrow^* \text{refl}$
A note on coinduction with dependent types

- Example: consider a co-inductive type $U$ with only one constructor
  $\text{in} : U \rightarrow U$

  $$u : U \quad \text{force} : U \rightarrow U$$

  $$u \overset{\text{def}}{=} \text{cofix } u := \text{in } u \quad \text{force} \overset{\text{def}}{=} \lambda x.\text{case } x \text{ of } \text{in } x' \Rightarrow \text{in } x'$$

- We can prove that $x = \text{force } x$ for any $x : U$

  $$\text{eq} : \Pi x : U. x = \text{force } x$$

  $$\text{eq} \overset{\text{def}}{=} \lambda x.\text{case } x \text{ of } \text{in } x' \Rightarrow \text{refl}$$

- Then, $\text{eq } u : u = \text{in } u$, and $\text{eq } u \rightarrow^* \text{refl}$

- But $\text{refl} : u = u$
A note on coinduction with dependent types

- Example: consider a co-inductive type $U$ with only one constructor
  
  $$
  \text{in} : U \rightarrow U
  $$

  $$
  u : U \\
  \text{force} : U \rightarrow U
  $$

  $$
  u \overset{\text{def}}{=} \text{cofix } u := \text{in } u \\
  \text{force} \overset{\text{def}}{=} \lambda x. \text{case } x \text{ of } \text{in } x' \Rightarrow \text{in } x'
  $$

- We can prove that $x = \text{force } x$ for any $x : U$

  $$
  \text{eq} : \Pi x : U. x = \text{force } x
  $$

  $$
  \text{eq} \overset{\text{def}}{=} \lambda x. \text{case } x \text{ of } \text{in } x' \Rightarrow \text{refl}
  $$

- Then, $\text{eq } u : u = \text{in } u$, and $\text{eq } u \rightarrow^* \text{refl}$

- But $\text{refl} : u = u$

- The types $u = u$ and $u = \text{in } u$ are not convertible since there is no case forcing the unfolding of $u$. 