Type-Based Methods for Termination and Productivity in Coq

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

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

July 31, 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-d destructors
- Recursive calls performed only on **structurally smaller terms**

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

\[ \Gamma(f : I \to T) \vdash M : I \to T \quad G(f, M) \]

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

- The predicate \( G(f, M) \) checks that all recursive calls of \( f \) in \( M \) are guarded by destructors
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**

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

- 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

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

- Typical example:

\[
\text{fix } \text{half} : \text{nat } \rightarrow \text{nat} := \lambda x. \text{case } x \text{ of } \\
| \text{O } \Rightarrow \text{O} \\
| \text{S O } \Rightarrow \text{O} \\
| \text{S (S p) } \Rightarrow \text{S(half p)}
\]

Recursive call is \texttt{guarded}. The recursive argument is smaller.

- The initial implementation of \texttt{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 $G$ (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).
Subtraction:

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

Subterm relation

Subtraction:

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

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

Subterm relation

Subtraction:

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

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

Division:

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

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

Subterm relation

Subtraction:

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

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

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

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

\[\text{minus } x_1 \ y \preceq x_1 \prec S x_1 \equiv x\]
Termination in Coq

Subterm relation

Subtraction:

\[
\text{fix } \text{minus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} := \lambda x y. \text{case } x, y \text{ of }
\]

\[
\begin{align*}
| O, _ & \Rightarrow x \\
| S x, O & \Rightarrow S x \\
| S x, S y & \Rightarrow \text{minus } x y
\end{align*}
\]

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

\[
\text{fix } \text{div} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} := \lambda x y. \text{case } x \text{ of }
\]

\[
\begin{align*}
| O & \Rightarrow O \\
| S x & \Rightarrow S(\text{div } (\text{minus } x y) y)
\end{align*}
\]
Termination in Coq

Subterm relation

Subtraction:

\[
\text{fix } \text{minus} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} := \lambda xy. \text{case } x, y \text{ of }
\begin{align*}
&\mid O, - \Rightarrow O \\
&\mid S x_1, O \Rightarrow S x_1 \\
&\mid S x_1, S y_1 \Rightarrow \text{minus } x_1 y_1
\end{align*}
\]

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

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

\[
\text{fix } \text{div} : \text{nat} \rightarrow \text{nat} \rightarrow \text{nat} := \lambda xy. \text{case } x \text{ of }
\begin{align*}
&\mid O \Rightarrow O \\
&\mid S x_1 \Rightarrow S(\text{div}(\text{minus } x_1 y) y)
\end{align*}
\]
Termination in Coq

Subterm relation

Subtraction:

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

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

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

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

\(\text{minus} \ x_1 \ y \not\prec x_1 \prec S x_1 \equiv x\)
Termination in Coq

Nested fixpoints

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)
Termination in Coq

Nested fixpoints

Inductive rose(A) : Type := node : A → list (rose A) → rose A

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

map := \( \lambda f : A \rightarrow B. \) fix map : list A → list B :=
\( \lambda l. \) 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

\textit{rmap} := \lambda f : A \rightarrow B. \text{fix } rmap : \text{rose } A \rightarrow \text{rose } B := \lambda t. \text{case } t \text{ of } \node x ts \Rightarrow \node (f \ x) (\text{map } rmap \ ts)

\textit{map} := \text{fix } map : (A \rightarrow B) \rightarrow \text{list } A \rightarrow \text{list } B := \lambda f \ l. \text{case } l \text{ of } \nil \Rightarrow \nil \text{ cons } x xs \Rightarrow \text{cons } (f \ x) (\text{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.], $\lambda^\hat{}$ [Joao Frade et al.], $F^\hat{}_\omega$ [Abel], CIC$^\hat{}$ [Barthe et al.], CC+rewriting [Blanqui et al.] ...
Termination using sized types

- Long history: Haskell [Pareto et al.], $\lambda^\infty$ [Joao Frade et al.], $F_\omega^\infty$ [Abel], $\text{CIC}^\infty$ [Barthe et al.], CC+rewriting [Blanqui et al.] ...
- Basic idea: user-defined datatypes are decorated with size information

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

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

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

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

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

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

- Sized types are approximations

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

Intuitive meaning: $[\text{nat}^{\langle s \rangle}] = \{O, S\ O, \ldots, S(\ldots (S\ O)\ldots)\}$

for $s=1$
Termination using sized types

- Size annotations keep track of the size of elements

\[ s ::= \nu \mid \mathbf{s} \mid \infty \]
Termination using sized types

- Size annotations keep track of the size of elements.

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

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

- Size annotations keep track of the size of elements

\[ s ::= \_ | \hat{s} | \infty \]

\[
\begin{align*}
\Gamma \vdash O : \text{nat} & \quad \Gamma \vdash M : \text{nat} \\
\Gamma \vdash S M : \text{nat}
\end{align*}
\]
Termination using sized types

- Size annotations keep track of the size of elements

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

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

- Size annotations keep track of the size of elements

\[ s ::= \_ | \hat{s} | \infty \]

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

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

\[ \text{upper bound} \]
Termination using sized types

- Size annotations keep track of the size of elements

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

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

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

- Substage relation

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

defines a subtype relation

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

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

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

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

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

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

\(\nu\) fresh

- Recursive calls on terms of smaller size
Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$
\frac{
\Gamma (f : \mathcal{I} \langle \nu \rangle \rightarrow T) \vdash M : \mathcal{I} \langle \hat{\nu} \rangle \rightarrow T
}{
\Gamma \vdash (\text{fix } f : \mathcal{I} \rightarrow T := M) : \mathcal{I} \langle s \rangle \rightarrow T}
$$  \(\nu\) fresh

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

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

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

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

\[
\begin{align*}
\text{filter} & \equiv \ldots : \Pi A. (A \to \text{bool}) \to \text{list} \quad A \to \text{list} \quad A \times \text{list} \quad A \\
(\text{++}) & \equiv \ldots : \Pi A. \text{list} \quad A \to \text{list} \quad A \to \text{list} \quad A
\end{align*}
\]
Example: quicksort

Non-structural recursion

\[
\begin{align*}
\text{filter} & \equiv \ldots : \Pi A. (A \to \text{bool}) \to \text{list } A \to \text{list } A \times \text{list } A \\
(++) & \equiv \ldots : \Pi A. \text{list } A \to \text{list } A \to \text{list } A
\end{align*}
\]

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

Bruno Barras, Jorge Luis Sacchini
Example: quicksort

Non-structural recursion

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

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

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

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

\[
\text{fix } \text{qsort} : \text{list } A \rightarrow \text{list } A :\equiv \\
\lambda x : \text{list } A. \text{case } x^\text{list}\langle \hat{\imath}\rangle \text{ of} \\
| \text{nil } \Rightarrow \text{nil} \\
| \text{cons } h t^\text{list}\langle \hat{\imath}\rangle \Rightarrow \text{let } (s, g) = \text{filter } (< h) t^\text{list}\langle \hat{\imath}\rangle \text{ in} \\
\hspace{1cm} (\text{qsort } s^\text{list}\langle \hat{\imath}\rangle ) ++ (\text{cons } h (\text{qsort } g^\text{list}\langle \hat{\imath}\rangle ))
\]

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

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

\[
\text{fix qsort} : \text{list} A \to \text{list} A := \\
\lambda x : \text{list} A. \text{case } x^{\text{list}\langle \hat{i}\rangle} \text{ of} \\
| \text{nil} \Rightarrow \text{nil} \\
| \text{cons } h t^{\text{list}\langle i\rangle} \Rightarrow \text{let } (s, g) = \text{filter } (< h) t^{\text{list}\langle i\rangle} \text{ in} \\
\quad (\text{qsort } s^{\text{list}\langle i\rangle}) \text{ ++ (cons } h \text{ (qsort } g^{\text{list}\langle i\rangle}))
\]

: \Pi A. \text{list}\langle s\rangle A \to \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>
Coinductive Types in Coq

Streams:

\textbf{CoInductive} stream \(A := \text{scons} : A \rightarrow \text{stream} A \rightarrow \text{stream} A\)
Coinductive Types in Coq

- Stream
  - Empty as an inductive type

  \textbf{CoInductive} \quad \text{stream} \ A := \ scons : A \to \text{stream} \ A \to \text{stream} \ A
Coinductive Types in Coq

- Streams:

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

- Corecursive functions produce streams:

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

  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:

\[
\begin{align*}
\text{cofix } Z_1 & := \text{scons}(0, Z_1) \\
\text{cofix } Z_2 & := \text{scons}(0, \text{tail } Z_2)
\end{align*}
\]
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 co-recursive 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:

  \[
  \text{cofix } Z_1 := \text{scons}(0, Z_1) \quad \checkmark
  \]

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

  \[
  (\text{tail } Z_2) \text{ loops}
  \]
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**

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

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

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

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

\[
nats := \text{cofix} \quad \text{nats} := \lambda n. \ scons(n, \text{nats}(1 + n)) \\
nats := \lambda n. \ \text{cofix} \quad \text{nats} := \text{scons}(n, \text{map}(\lambda x. 1+x) \text{nats})
\]
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 \textbf{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 at least \( s \) elements can be produced

Size annotations are contra-variant:

$$\begin{align*}
  r \sqsubseteq s \\
  \text{stream} \langle s \rangle T \leq \text{stream} \langle r \rangle T
\end{align*}$$

$$\begin{align*}
  s \sqsubseteq r \\
  \text{list} \langle s \rangle T \leq \text{list} \langle r \rangle T
\end{align*}$$
Typing rules are similar to the inductive case

Rules for constructors:

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

Cofixpoint definition is also similar to fixpoint definition:

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

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

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

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

- Cofixpoint definition is also similar to fixpoint definition:

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

Examples

map : (A → B) → stream  A → stream  B

merge : stream  nat → stream  nat → stream  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)
    (merge (map (λx. 3*x) ham)
    (map (λx. 5*x) ham)))

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

Bruno Barras, Jorge Luis Sacchini
Type-Based Methods for Termination and Productivity in Coq
Co-recursive definitions

Examples

\[
\begin{align*}
\text{map} &: (A \rightarrow B) \rightarrow \text{stream}(s)A \rightarrow \text{stream}(s)B \\
\text{merge} &: \text{stream}(s)\text{nat} \rightarrow \text{stream}(s)\text{nat} \rightarrow \text{stream}(s)\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 nat} := \\
&\quad \text{scons}(1, \text{merge} (\text{map} (\lambda x. 2\times x) \ \text{ham}) \ ) \\
&\quad \quad \quad \quad \quad \quad \quad (\text{merge} (\text{map} (\lambda x. 3\times x) \ \text{ham}) \ ) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad (\text{map} (\lambda x. 5\times x) \ \text{ham}) \ )) \\
\text{ham} &= (1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 10\ 12\ 15\ldots) \\
\end{align*}
\]
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 productivity:
  - 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 -> B) (xs : List A) : List B :=
match xs with
  nil    => nil
  cons h t => cons (f h) (map f t)
end.
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 -> B) -> List A -> List B.
In a future Coq version ...

Fixpoint map \( f : A \rightarrow B \) \( (xs : \text{List} A) : \text{List} B \) :=
  match xs with
  nil => nil
  cons h t => cons (f h) (map f t)
end.

Check \( \text{map} \).
\( \text{map} : \forall \, (A \rightarrow B) \rightarrow \text{List} A \rightarrow \text{List} B \).

Fixpoint \( \text{ntail} \) \( A \) \( (x : \text{nat}) \) \( : \text{List} A \rightarrow \text{List} A \) :=
  ...

Bruno Barras, Jorge Luis Sacchini
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 ntail ℓ A (x : nat<ℓ>) : 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 : ∀ (A -> B) -> List A -> List B.

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

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

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

Check map.

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

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

Check ntail.

ntail : \(\forall\ i \ \forall J_1 \ \forall J_2. \ J_2 \sqsubseteq J_1 \Rightarrow\)
  \(\forall A, \text{nat}<i> \to \text{List}<J_1> A \to \text{List}<J_2> A.\)
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!