Type-Based Methods for Termination and Productivity in Coq

Bruno Barras[1] Jorge Luis Sacchini[2]

[1]INRIA Saclay & LIX

[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: (let $f\ x = f\ x$ 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

$$\frac{\Gamma(f : I \to T) \vdash M : I \to T}{\Gamma \vdash (\mathrm{fix}\ f : I \to T := M) : I \to T}$$

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

$$\frac{\Gamma(f : I \to T) \vdash M : I \to T \qquad \mathcal{G}(f, M)}{\Gamma \vdash (\text{fix } f : I \to T := M) : I \to T}$$

- The predicate $\mathcal{G}(f, M)$ 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

$$\frac{\Gamma(f : I \to T) \vdash M : I \to T \qquad \mathcal{G}(f, M)}{\Gamma \vdash (\text{fix } f : I \to T := M) : I \to T}$$

- The predicate $\mathcal{G}(f, M)$ 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

$$\frac{\Gamma(f : I \to T) \vdash M : I \to T \qquad M \to^* N \qquad \mathcal{G}(f, N)}{\Gamma \vdash (\text{fix } f : I \to T := M) : I \to T}$$

Termination in Coq

- Typical example:

$$\begin{array}{ll} \text{fix } \textsf{half} : \text{nat} \to \text{nat} := \lambda x.\ \text{case } x \text{ of} & \\ & \mid\ O \Rightarrow O \\ & \mid\ S\, O \Rightarrow O \\ & \mid\ S\,(S\, p) \Rightarrow S(\textsf{half}\ p) \end{array}$$

  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

- Typical example:

$$
\begin{array}{ll}
\text{fix } \textcolor{blue}{\text{half}} : \text{nat} \to \text{nat} := \lambda x.\ \text{case } x \text{ of} & \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid\ O \Rightarrow O & \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid\ S\,O \Rightarrow O & \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid\ S\,(S\,p) \Rightarrow S(\textcolor{blue}{\text{half}}\ p) &
\end{array}
$$

$p \prec S\,(S\,p)$

  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:

$$
\begin{array}{l}
\text{fix } \textcolor{blue}{\text{minus}} : \textcolor{green}{\text{nat}} \to \text{nat} \to \text{nat} := \lambda xy.\ \text{case } x, y \text{ of} \\
\qquad \mid\ O, \_ \Rightarrow x \\
\qquad \mid\ S\,x_1, O \Rightarrow S\,x_1 \\
\qquad \mid\ S\,x_1, S\,y_1 \Rightarrow \textcolor{blue}{\text{minus}}\ x_1\ y_1
\end{array}
$$

Termination in Coq

Subterm relation

Subtraction:

$$\begin{aligned}\text{fix } \textcolor{blue}{\text{minus}} : \textcolor{green}{\text{nat}} \to \text{nat} \to \text{nat} := \lambda xy.\ \text{case } x, y \text{ of}\ & \\ & \mid\ O, \_ \Rightarrow x \\ & \mid\ S\,x_1, O \Rightarrow S\,x_1 \\ & \mid\ \textcolor{red}{S\,x_1}, S\,y_1 \Rightarrow \textcolor{blue}{\text{minus}}\ \textcolor{red}{x_1}\ y_1\end{aligned}$$

✔

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

Termination in Coq

Subterm relation

Subtraction:

$$\begin{array}{ll} \text{fix } \mathsf{minus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} := \lambda xy.\ \mathsf{case}\ x, y\ \mathsf{of} & \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid O, \_ \Rightarrow x & \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid S\,x_1, O \Rightarrow S\,x_1 & \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid S\,x_1, S\,y_1 \Rightarrow \mathsf{minus}\ x_1\ y_1 & \end{array}$$

✔

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

Division: $\mathrm{div}\ x\ y = \left\lceil \frac{x}{y+1} \right\rceil$

$$\begin{array}{l} \text{fix } \mathsf{div} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} := \lambda xy.\ \mathsf{case}\ x\ \mathsf{of} \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid O \Rightarrow O \\ \qquad\qquad\qquad\qquad\qquad\qquad \mid S\,x_1 \Rightarrow S(\mathsf{div}\,(\mathsf{minus}\ x_1\ y)\ y) \end{array}$$

Termination in Coq

Subterm relation

Subtraction:

$$\begin{array}{l} \text{fix minus} : \text{nat} \to \text{nat} \to \text{nat} := \lambda xy.\ \text{case } x, y \text{ of} \\ \qquad \mid\ O, \_ \Rightarrow x \\ \qquad \mid\ S\,x_1, O \Rightarrow S\,x_1 \\ \qquad \mid\ S\,x_1, S\,y_1 \Rightarrow \text{minus } x_1\ y_1 \end{array}$$

✔

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

Division: $\mathrm{div}\ x\ y = \left\lceil \frac{x}{y+1} \right\rceil$

$$\begin{array}{l} \text{fix div} : \text{nat} \to \text{nat} \to \text{nat} := \lambda xy.\ \text{case } x \text{ of} \\ \qquad \mid\ O \Rightarrow O \\ \qquad \mid\ S\,x_1 \Rightarrow S(\text{div}\,(\text{minus } x_1\ y)\ y) \end{array}$$

✔

$\text{minus } x_1\ y \preceq x_1 \prec S\,x_1 \equiv x$

Termination in Coq

Subterm relation

Subtraction:

$$\begin{array}{l} \text{fix minus} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} := \lambda xy.\ \text{case } x, y \text{ of} \\ \qquad \mid\ O, \_ \Rightarrow x \\ \qquad \mid\ S\,x_1, O \Rightarrow S\,x_1 \\ \qquad \mid\ S\,x_1, S\,y_1 \Rightarrow \text{minus } x_1\ y_1 \end{array}$$

Division: $\mathrm{div}\ x\ y = \left\lceil \frac{x}{y+1} \right\rceil$

$$\begin{array}{l} \text{fix div} : \mathsf{nat} \to \mathsf{nat} \to \mathsf{nat} := \lambda xy.\ \text{case } x \text{ of} \\ \qquad \mid\ O \Rightarrow O \\ \qquad \mid\ S\,x_1 \Rightarrow S(\text{div}\,(\text{minus } x_1\ y)\ y) \end{array}$$

Termination in Coq

Subterm relation

Subtraction:

$$\text{fix } \textsf{minus} : \textsf{nat} \to \textsf{nat} \to \textsf{nat} := \lambda xy.\ \textsf{case } x, y \textsf{ of } \begin{array}{l} \mid\ O, \_ \Rightarrow O \\ \mid\ S\,x_1, O \Rightarrow S\,x_1 \\ \mid\ S\,x_1, S\,y_1 \Rightarrow \textsf{minus}\ x_1\ y_1 \end{array}$$

✔

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

Division: $\mathrm{div}\ x\ y = \left\lceil \frac{x}{y+1} \right\rceil$

$$\text{fix } \textsf{div} : \textsf{nat} \to \textsf{nat} \to \textsf{nat} := \lambda xy.\ \textsf{case } x \textsf{ of } \begin{array}{l} \mid\ O \Rightarrow O \\ \mid\ S\,x_1 \Rightarrow S(\textsf{div}\,(\textsf{minus}\ x_1\ y)\ y) \end{array}$$

Termination in Coq

Subterm relation

Subtraction:

$$\text{fix } \color{blue}{\text{minus}} : \color{green}{\text{nat}} \to \text{nat} \to \text{nat} := \lambda xy.\ \text{case } x, y \text{ of } \begin{array}{l} \mid\ O, \_ \Rightarrow \color{red}{O} \\ \mid\ S\,x_1, O \Rightarrow S\,x_1 \\ \mid\ S\,x_1, S\,y_1 \Rightarrow \color{blue}{\text{minus}}\ x_1\ y_1 \end{array}$$

✔

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

Division: $\mathrm{div}\ x\ y = \left\lceil \frac{x}{y+1} \right\rceil$

$$\text{fix } \color{blue}{\text{div}} : \color{green}{\text{nat}} \to \text{nat} \to \text{nat} := \lambda xy.\ \text{case } x \text{ of } \begin{array}{l} \mid\ O \Rightarrow O \\ \mid\ S\,x_1 \Rightarrow S(\color{blue}{\text{div}}\,(\color{blue}{\text{minus}}\ x_1\ y)\ y) \end{array}$$

✘

$\color{blue}{\text{minus}}\ x_1\ y \color{red}{\not\prec} x_1 \prec S\,x_1 \equiv x$

Termination in Coq

Nested fixpoints

$$\text{Inductive rose}(A) : \text{Type} := \text{node} : A \to \text{list}\,(\text{rose}\,A) \to \text{rose}\,A$$

$$\begin{aligned}
\text{rmap} := {} & \lambda f : A \to B.\ \text{fix } \text{rmap} : \text{rose}\,A \to \text{rose}\,B := \\
& \quad \lambda t.\ \text{case } t \text{ of} \\
& \qquad \text{node}\,x\,ts \Rightarrow \text{node}\,(f\ x)\ (\text{map rmap } ts)
\end{aligned}$$

$$\begin{aligned}
\text{map} := {} & \lambda f : A \to B.\ \text{fix map} : \text{list}\,A \to \text{list}\,B := \\
& \quad \lambda l.\ \text{case } l \text{ of} \\
& \qquad \text{nil} \Rightarrow \text{nil} \\
& \qquad \text{cons}\,x\,xs \Rightarrow \text{cons}\,(f\ x)\,(\text{map } xs)
\end{aligned}$$

Termination in Coq

Nested fixpoints

$$\text{Inductive rose}(A) : \text{Type} := \text{node} : A \to \text{list}\,(\text{rose}\,A) \to \text{rose}\,A$$

$$\begin{aligned}
&\text{rmap} := \lambda f : A \to B.\ \text{fix}\ \text{rmap} : \text{rose}\,A \to \text{rose}\,B := \\
&\qquad \lambda t.\ \text{case}\ t\ \text{of} \\
&\qquad\quad \text{node}\,x\,ts \Rightarrow \text{node}\,(f\ x)\ (\text{map}\ \text{rmap}\ ts)
\end{aligned}$$

✔

$$\begin{aligned}
&\text{map} := \lambda f : A \to B.\ \text{fix}\ \text{map} : \text{list}\,A \to \text{list}\,B := \\
&\qquad \lambda l.\ \text{case}\ l\ \text{of} \\
&\qquad\quad \text{nil} \Rightarrow \text{nil} \\
&\qquad\quad \text{cons}\,x\,xs \Rightarrow \text{cons}\,(f\ x)\,(\text{map}\ xs)
\end{aligned}$$

✔

Termination in Coq

Nested fixpoints

$$\text{Inductive rose}(A) : \text{Type} := \text{node} : A \to \text{list}\,(\text{rose}\,A) \to \text{rose}\,A$$

$$\begin{aligned}
\text{rmap} := \lambda f : A \to B.\ &\text{fix rmap} : \text{rose}\,A \to \text{rose}\,B := \\
&\lambda t.\ \text{case}\ t\ \text{of} \\
&\quad \text{node}\,x\,ts \Rightarrow \text{node}\,(f\ x)\ (\text{map rmap}\ ts)
\end{aligned}$$

$$\begin{aligned}
\text{map} :=\ &\text{fix map} : (A \to B) \to \text{list}\,A \to \text{list}\,B := \\
&\lambda f\ l.\ \text{case}\ l\ \text{of} \\
&\quad \text{nil} \Rightarrow \text{nil} \\
&\quad \text{cons}\,x\,xs \Rightarrow \text{cons}\,(f\ x)\,(\text{map}\ f\ xs)
\end{aligned}$$

✔

Termination in Coq

Nested fixpoints

$$\text{Inductive rose}(A) : \text{Type} := \text{node} : A \to \text{list}\,(\text{rose}\,A) \to \text{rose}\,A$$

$$\begin{aligned}
\text{rmap} := \lambda f : A \to B.\ &\text{fix}\ \text{rmap} : \text{rose}\,A \to \text{rose}\,B := \\
&\lambda t.\ \text{case}\ t\ \text{of} \\
&\quad \text{node}\,x\,ts \Rightarrow \text{node}\,(f\ x)\ (\text{map}\ \text{rmap}\ ts)
\end{aligned}$$

✖

$$\begin{aligned}
\text{map} :=\ &\text{fix}\ \text{map} : (A \to B) \to \text{list}\,A \to \text{list}\,B := \\
&\lambda f\ l.\ \text{case}\ l\ \text{of} \\
&\quad \text{nil} \Rightarrow \text{nil} \\
&\quad \text{cons}\,x\,xs \Rightarrow \text{cons}\,(f\ x)\,(\text{map}\ f\ xs)
\end{aligned}$$

✔

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.], $\lambda\widehat{\ }$ [Joao Frade et al.], $F_\omega\widehat{\ }$ [Abel], $\mathrm{CIC}\widehat{\ }$ [Barthe et al.], CC+rewriting [Blanqui et al.] ...

Termination using sized types

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

$$\mathsf{nat} ::= O : \mathsf{nat} \mid S : \mathsf{nat} \to \mathsf{nat}$$

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

Termination using sized types

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

$$\mathsf{nat} ::= O : \mathsf{nat} \mid S : \mathsf{nat} \to \mathsf{nat}$$

  Intuitive meaning: $[\mathsf{nat}] = \{O, S\,O, S(S\,O), \ldots\}$
- Sized types are approximations

$$\mathsf{nat}\langle s \rangle$$

  Intuitive meaning: $[\mathsf{nat}\langle s \rangle] = \{O, S\,O, \ldots, \underbrace{S(\ldots(S\,O)\ldots)}_{s-1}\}$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \imath \mid \widehat{s} \mid \infty$$

Termination using sized types

- Size annotations keep track of the size

$\widehat{\infty} = \infty$

$$s ::= \imath \mid \widehat{s} \mid \infty$$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \imath \mid \widehat{s} \mid \infty$$

$$\frac{}{\Gamma \vdash O : \mathsf{nat}} \qquad \frac{\Gamma \vdash M : \mathsf{nat}}{\Gamma \vdash S\, M : \mathsf{nat}}$$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \imath \mid \widehat{s} \mid \infty$$

$$\frac{}{\Gamma \vdash O : \mathsf{nat}\langle \widehat{s} \rangle} \qquad \frac{\Gamma \vdash M : \mathsf{nat}\langle s \rangle}{\Gamma \vdash S\, M : \mathsf{nat}\langle \widehat{s} \rangle}$$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \imath \mid \widehat{s} \mid \infty$$

upper bound

$$\frac{}{\Gamma \vdash O : \mathsf{nat}\langle \widehat{s} \rangle} \qquad \frac{\Gamma \vdash M : \mathsf{nat}\langle s \rangle}{\Gamma \vdash S\, M : \mathsf{nat}\langle \widehat{s} \rangle}$$

Termination using sized types

- Size annotations keep track of the size of elements

$$s ::= \imath \mid \widehat{s} \mid \infty$$

$$\frac{}{\Gamma \vdash O : \mathsf{nat}\langle \widehat{s} \rangle} \qquad \frac{\Gamma \vdash M : \mathsf{nat}\langle s \rangle}{\Gamma \vdash S\, M : \mathsf{nat}\langle \widehat{s} \rangle}$$

- Substage relation

$$\frac{}{s \sqsubseteq \widehat{s}} \qquad \frac{}{s \sqsubseteq \infty}$$

defines a subtype relation

$$\frac{s \sqsubseteq r}{\mathsf{nat}\langle s \rangle \leq \mathsf{nat}\langle r \rangle}$$

Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$\frac{\Gamma(f : I \quad\quad \to T) \vdash M : I \quad\quad \to T}{\Gamma \vdash (\text{fix } f : I \to T := M) : I \quad\quad \to T}$$

Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$\frac{\Gamma(f : I\langle\imath\rangle \to T) \vdash M : I\langle\widehat{\imath}\rangle \to T}{\Gamma \vdash (\mathsf{fix}\ f : I \to T := M) : I\langle s\rangle \to T}\ \ \imath \text{ fresh}$$

- Recursive calls on terms of smaller size

Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$\frac{\Gamma(f : I\langle\imath\rangle \to T) \vdash M : I\langle\widehat{\imath}\rangle \to T}{\Gamma \vdash (\mathrm{fix}\ f : I \to T := M) : I\langle s\rangle \to T}\ \imath \text{ fresh}$$

- Recursive calls on terms of smaller size
- Size-preserving functions: return type T can depend on $\imath$

Termination using sized types

Fixpoint rule

Recursive functions are defined on approximations of datatypes:

$$\frac{\Gamma(f : I\langle\imath\rangle \to T) \vdash M : I\langle\widehat{\imath}\rangle \to T}{\Gamma \vdash (\text{fix } f : I \to T := M) : I\langle s\rangle \to T} \quad \imath \text{ fresh}$$

- Recursive calls on terms of smaller size
- Size-preserving functions: return type T can depend on $\imath$
- Non-structural recursion

Example: quicksort

Non-structural recursion

$$\mathsf{filter} \equiv \ldots : \Pi A.(A \to \mathsf{bool}) \to \mathsf{list} \quad A \to \mathsf{list} \quad A \times \mathsf{list} \quad A$$

$$(++) \equiv \ldots : \Pi A.\mathsf{list} \quad A \to \mathsf{list} \quad A \to \mathsf{list} \quad A$$

Example: quicksort

Non-structural recursion

$$\text{filter} \equiv \ldots : \Pi A.(A \to \text{bool}) \to \text{list} \quad A \to \text{list} \quad A \times \text{list} \quad A$$

$$(++) \equiv \ldots : \Pi A.\text{list} \quad A \to \text{list} \quad A \to \text{list} \quad A$$

$$
\begin{array}{l}
\text{fix } \text{qsort} : \text{list}\, A \to \text{list}\, A := \\
\lambda x : \text{list}\, A.\ \text{case}\ x \qquad \text{of} \\
\qquad\qquad \mid \text{nil} \Rightarrow \text{nil} \\
\qquad\qquad \mid \text{cons}\, h\, t \qquad \Rightarrow \text{let}\ (s, g) = \text{filter}\,(< h)\, t \qquad \text{in} \\
\qquad\qquad\qquad\qquad\qquad (\text{qsort}\, s \qquad) ++ (\text{cons}\, h\, (\text{qsort}\, g \qquad))
\end{array}
$$

Example: quicksort

Non-structural recursion

$$\mathsf{filter} \equiv \ldots : \Pi A.(A \to \mathsf{bool}) \to \mathsf{list}\langle s\rangle\, A \to \mathsf{list}\langle s\rangle\, A \times \mathsf{list}\langle s\rangle\, A$$

$$(++) \equiv \ldots : \Pi A.\mathsf{list}\langle s\rangle\, A \to \mathsf{list}\langle r\rangle\, A \to \mathsf{list}\langle \infty\rangle\, A$$

$$\begin{array}{l}
\mathsf{fix}\ \mathsf{qsort} : \mathsf{list}\, A \to \mathsf{list}\, A := \\
\lambda x : \mathsf{list}\, A.\ \mathsf{case}\ x\ \ \mathsf{of} \\
\qquad\qquad \mid \mathsf{nil} \Rightarrow \mathsf{nil} \\
\qquad\qquad \mid \mathsf{cons}\, h\, t \Rightarrow \mathsf{let}\ (s, g) = \mathsf{filter}\,(< h)\, t\ \ \mathsf{in} \\
\qquad\qquad\qquad\qquad (\mathsf{qsort}\, s) ++ (\mathsf{cons}\, h\, (\mathsf{qsort}\, g))
\end{array}$$

Example: quicksort

Non-structural recursion

$$\text{filter} \equiv \ldots : \Pi A.(A \to \text{bool}) \to \text{list}\langle s \rangle\, A \to \text{list}\langle s \rangle\, A \times \text{list}\langle s \rangle\, A$$

$$(++) \equiv \ldots : \Pi A.\text{list}\langle s \rangle\, A \to \text{list}\langle r \rangle\, A \to \text{list}\langle \infty \rangle\, A$$

$$\begin{array}{l} \text{fix } \text{qsort} : \text{list}\, A \to \text{list}\, A := \\ \lambda x : \text{list}\, A.\ \text{case } x^{\text{list}\langle \widehat{\imath} \rangle} \text{ of} \\ \qquad\qquad \mid \text{nil} \Rightarrow \text{nil} \\ \qquad\qquad \mid \text{cons}\, h\, t^{\text{list}\langle \imath \rangle} \Rightarrow \text{let } (s, g) = \text{filter}\,(< h)\, t^{\text{list}\langle \imath \rangle} \text{ in} \\ \qquad\qquad\qquad\qquad (\text{qsort}\, s^{\text{list}\langle \imath \rangle}) \mathbin{++} (\text{cons}\, h\, (\text{qsort}\, g^{\text{list}\langle \imath \rangle})) \end{array}$$

Example: quicksort

Non-structural recursion

$$\text{filter} \equiv \ldots : \Pi A.(A \to \text{bool}) \to \text{list}\langle s \rangle\, A \to \text{list}\langle s \rangle\, A \times \text{list}\langle s \rangle\, A$$

$$(++) \equiv \ldots : \Pi A.\text{list}\langle s \rangle\, A \to \text{list}\langle r \rangle\, A \to \text{list}\langle \infty \rangle\, A$$

$$\begin{array}{l}
\text{fix}\ \text{qsort} : \text{list}\, A \to \text{list}\, A := \\
\lambda x : \text{list}\, A.\ \text{case}\ x^{\text{list}\langle \hat{\imath} \rangle}\ \text{of} \\
\qquad\qquad \mid \text{nil} \Rightarrow \text{nil} \\
\qquad\qquad \mid \text{cons}\, h\, t^{\text{list}\langle \imath \rangle} \Rightarrow \text{let}\ (s, g) = \text{filter}\,(< h)\, t^{\text{list}\langle \imath \rangle}\ \text{in} \\
\qquad\qquad\qquad\qquad (\text{qsort}\, s^{\text{list}\langle \imath \rangle}) ++ (\text{cons}\, h\, (\text{qsort}\, g^{\text{list}\langle \imath \rangle})) \\
\qquad\qquad : \Pi A.\text{list}\langle s \rangle\, A \to \text{list}\langle \infty \rangle\, A
\end{array}$$

Type-based termination

- Handle higher-order data

$$\mathsf{node} : \Pi A.A \to \mathsf{list}\langle\infty\rangle\,(\mathsf{rose}\langle s\rangle\,A) \to \mathsf{rose}\langle\,\widehat{s}\,\rangle\,A$$

- 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.

Inductive types	**Coinductive types**
Induction	Coinduction
Recursive functions consume data	Corecursive functions produce data

Coinductive Types in Coq

- Streams:

$$\textbf{CoInductive}\ \mathsf{stream}\, A := \mathsf{scons} : A \to \mathsf{stream}\, A \to \mathsf{stream}\, A$$

Coinductive Types in Coq

- Strea

Empty as an inductive type

$$\textbf{CoInductive } \mathsf{stream}\, A := \mathsf{scons} : A \to \mathsf{stream}\, A \to \mathsf{stream}\, A$$

Coinductive Types in Coq

- Streams:

$$\textbf{CoInductive}\ \mathsf{stream}\, A := \mathsf{scons} : A \to \mathsf{stream}\, A \to \mathsf{stream}\, A$$

- Corecursive functions produce streams:

$$\mathsf{zeroes} \;:=\; \mathsf{cofix}\ Z := \mathsf{scons}(0, Z)$$

zeroes produce the stream:

$$\mathsf{scons}(0, \mathsf{scons}(0, \mathsf{scons}(0, \ldots)))$$

Coinductive types

Inductive types	**Coinductive types**
Termination	Productivity

- 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:

$$\mathsf{cofix}\ Z_1 := \mathsf{scons}(0, Z_1)$$

$$\mathsf{cofix}\ Z_2 := \mathsf{scons}(0, \mathsf{tail}\ Z_2)$$

Coinductive types

Inductive types	**Coinductive types**
Termination	Productivity

- 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:

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

$$\text{cofix } Z_2 := \text{scons}(0, \text{tail } Z_2)$$

Coinductive types

Inductive types	**Coinductive types**
Termination	Productivity

- 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:

$$\text{cofix } Z_1 := \text{scons}(0, Z_1) \quad ✔$$
$$\text{cofix } Z_2 := \text{scons}(0, \text{tail } Z_2) \quad ✖$$

Coinductive types

Inductive types	**Coinductive types**
Termination	Productivity

- 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:

(tail Z_2) loops

$$\text{cofix } Z_1 := \text{scons}(0, Z_1) \quad ✔$$

$$\text{cofix } Z_2 := \text{scons}(0, \text{tail } Z_2) \quad ✖$$

Syntactic-Based Methods for Productivity

Inductive types	**Coinductive types**
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

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

Syntactic-Based Methods for Productivity

Inductive types	**Coinductive types**
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

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

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

Syntactic-Based Methods for Productivity

Inductive types	**Coinductive types**
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

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

$$\mathsf{nats} \;:=\; \mathsf{cofix}\ \mathsf{nats} := \lambda n.\ \mathsf{scons}(n, \mathsf{nats}\ (1+n)) \quad \checkmark$$

Syntactic-Based Methods for Productivity

Inductive types	**Coinductive types**
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

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

$$\text{nats} \;:=\; \text{cofix}\ \text{nats} := \lambda n.\ \text{scons}(n, \text{nats}\ (1+n)) \quad ✔$$

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

Syntactic-Based Methods for Productivity

Inductive types	Coinductive types
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

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

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

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

Syntactic-Based Methods for Productivity

Inductive types	**Coinductive types**
Termination	Productivity
Guarded-by-Destructor	Guarded-by-Constructor

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

$$\text{nats} \;:=\; \text{cofix } \text{nats} := \lambda n.\, \text{scons}(n, \text{nats } (1+n))$$ ✔

$$\text{nats} \;:=\; \lambda n.\, \text{cofix } \text{nats} := \text{scons}(n, \text{map } (\lambda x.\, 1+x) \text{ nats})$$ ✖

Type-Based Methods for Productivity

- Sized types can be applied to productivity checking as well!

Type-Based Methods for Productivity

- Sized types can be applied to productivity checking as well!
- Dual meaning of size annotations on coinductive types

$$\mathsf{stream}\langle s \rangle\, A$$

  is the type of streams of which at least s elements can be produced

Type-Based Methods for Productivity

- Sized types can be applied to productivity checking as well!
- Dual meaning of size annotations on coinductive types

$$\mathsf{stream}\langle s\rangle\ A$$

  is the type of streams of which at least s elements can be produced
- Size annotations are contra-variant:

$$\frac{r \sqsubseteq s}{\mathsf{stream}\langle s\rangle\ T \leq \mathsf{stream}\langle r\rangle\ T}$$

Type-Based Methods for Productivity

- Sized types can be applied to productivity checking as well!
- Dual meaning of size annotations on coinductive types

$$\mathsf{stream}\langle s \rangle\ A$$

  is the type of streams of which at least s elements can be produced
- Size annotations are contra-variant:

$$\frac{r \sqsubseteq s}{\mathsf{stream}\langle s \rangle\ T \leq \mathsf{stream}\langle r \rangle\ T} \qquad \frac{s \sqsubseteq r}{\mathsf{list}\langle s \rangle\ T \leq \mathsf{list}\langle r \rangle\ T}$$

Type-Based Methods for Productivity

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

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : \mathsf{stream}\langle s \rangle\, A}{\Gamma \vdash \mathsf{scons}(M, N) : \mathsf{stream}\langle \widehat{s} \rangle\, A}$$

- Cofixpoint definition is also similar to fixpoint definition:

$$\frac{\Gamma(f : \mathsf{stream}\langle \imath \rangle\, A) \vdash M : \mathsf{stream}\langle \widehat{\imath} \rangle\, A}{\Gamma \vdash \mathsf{cofix}\ f := M : \mathsf{stream}\langle s \rangle\, A}\ \imath \text{ fresh}$$

Type-Based Methods for Productivity

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

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : \mathsf{stream}\langle s \rangle\, A}{\Gamma \vdash \mathsf{scons}(M, N) : \mathsf{stream}\langle \widehat{s} \rangle\, A} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : \mathsf{list}\langle s \rangle\, A}{\Gamma \vdash \mathsf{cons}(M, N) : \mathsf{list}\langle \widehat{s} \rangle\, A}$$

- Cofixpoint definition is also similar to fixpoint definition:

$$\frac{\Gamma(f : \mathsf{stream}\langle \imath \rangle\, A) \vdash M : \mathsf{stream}\langle \widehat{\imath} \rangle\, A}{\Gamma \vdash \mathsf{cofix}\ f := M : \mathsf{stream}\langle s \rangle\, A}\ \imath \text{ fresh}$$

Type-Based Methods for Productivity

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

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : \mathsf{stream}\langle s \rangle\ A}{\Gamma \vdash \mathsf{scons}(M, N) : \mathsf{stream}\langle \widehat{s} \rangle\ A} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : \mathsf{list}\langle s \rangle\ A}{\Gamma \vdash \mathsf{cons}(M, N) : \mathsf{list}\langle \widehat{s} \rangle\ A}$$

- Cofixpoint definition is also similar to fixpoint definition:

$$\frac{\Gamma(f : \mathsf{stream}\langle \imath \rangle\ A) \vdash M : \mathsf{stream}\langle \widehat{\imath} \rangle\ A}{\Gamma \vdash \mathsf{cofix}\ f := M : \mathsf{stream}\langle s \rangle\ A}\ \imath \text{ fresh}$$

$$\frac{\Gamma(f : \mathsf{list}\langle \imath \rangle\ A \to U) \vdash M : \mathsf{list}\langle \widehat{\imath} \rangle\ A \to U}{\Gamma \vdash \mathsf{fix}\ f := M : \mathsf{list}\langle s \rangle\ A \to U}\ \imath \text{ fresh}$$

Co-recursive definitions

Examples

$$\mathrm{map} : (A \to B) \to \mathrm{stream} \quad A \to \mathrm{stream} \quad B$$

$$\mathrm{merge} : \mathrm{stream} \quad \mathrm{nat} \to \mathrm{stream} \quad \mathrm{nat} \to \mathrm{stream} \quad \mathrm{nat}$$

$$\mathrm{merge}\ (1\ 3\ 5\ldots)\ (2\ 4\ 6\ldots) = (1\ 2\ 3\ 4\ldots)$$

$$\begin{aligned}\mathrm{ham} := \ & \mathrm{cofix}\ \mathrm{ham} : \mathrm{stream}\ \mathit{nat} := \\ & \quad \mathrm{scons}(1, \mathrm{merge}\ (\mathrm{map}\ (\lambda x.\, 2{*}x)\ \mathrm{ham}\quad) \\ & \qquad\qquad (\mathrm{merge}\ (\mathrm{map}\ (\lambda x.\, 3{*}x)\ \mathrm{ham}\quad) \\ & \qquad\qquad\qquad (\mathrm{map}\ (\lambda x.\, 5{*}x)\ \mathrm{ham}\quad)))\end{aligned}$$

$$\mathrm{ham} = (1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 10\ 12\ 15\ldots)$$

Co-recursive definitions

Examples

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

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

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

$$\begin{aligned}
\text{ham} := {} & \text{cofix}\ \text{ham} : \text{stream}\ \mathit{nat} := \\
& \text{scons}(1, \text{merge}\ (\text{map}\ (\lambda x.\, 2{*}x)\ \text{ham}\quad) \\
& \qquad (\text{merge}\ (\text{map}\ (\lambda x.\, 3{*}x)\ \text{ham}\quad) \\
& \qquad\qquad (\text{map}\ (\lambda x.\, 5{*}x)\ \text{ham}\quad)))
\end{aligned}$$

$$\text{ham} = (1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 10\ 12\ 15 \ldots)$$

Co-recursive definitions

Examples

$$\mathsf{map} : (A \to B) \to \mathsf{stream}\langle s \rangle A \to \mathsf{stream}\langle s \rangle B$$

$$\mathsf{merge} : \mathsf{stream}\langle s \rangle \mathsf{nat} \to \mathsf{stream}\langle s \rangle \mathsf{nat} \to \mathsf{stream}\langle s \rangle \mathsf{nat}$$

$$\mathsf{merge}\ (1\ 3\ 5 \ldots)\ (2\ 4\ 6 \ldots) = (1\ 2\ 3\ 4 \ldots)$$

$$\begin{aligned}\mathsf{ham} := {} & \mathsf{cofix}\ \mathsf{ham} : \mathsf{stream}\ \mathit{nat} := \\ & \mathsf{scons}(1, \mathsf{merge}\ (\mathsf{map}\ (\lambda x.\, 2{*}x)\ \mathsf{ham}^{\mathsf{stream}\langle \imath \rangle}) \\ & \qquad\qquad (\mathsf{merge}\ (\mathsf{map}\ (\lambda x.\, 3{*}x)\ \mathsf{ham}^{\mathsf{stream}\langle \imath \rangle}) \\ & \qquad\qquad\qquad (\mathsf{map}\ (\lambda x.\, 5{*}x)\ \mathsf{ham}^{\mathsf{stream}\langle \imath \rangle})))\end{aligned}$$

$$\mathsf{ham} = (1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 10\ 12\ 15 \ldots)$$

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.) : $\lambda\widehat{\ }$, $F\widehat{\ }$, $\mathrm{CIC}\widehat{\ }$
- 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 ı (f : A -> B) (xs : List<ı> A) : List<ı> B :=
  match xs with
    nil      => nil
    cons h t => cons (f h) (map f t)
  end.
```

In a future Coq version . . .

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

Check map.
map : ∀ ı. (A -> B) -> List<ı> A -> List<ı> B.
```

In a future Coq version . . .

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

Check map.
map : ∀ ı. (A -> B) -> List<ı> A -> List<ı> B.

Fixpoint ntail ı A (x : nat<ı>) : List A → List A :=
  ...
```

In a future Coq version . . .

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

Check map.
map : ∀ ı. (A -> B) -> List<ı> A -> List<ı> B.

Fixpoint ntail ı A (x : nat<ı>) : List A → List A :=
  ...

Check ntail.
```

In a future Coq version . . .

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

Check map.
map : ∀ ı. (A -> B) -> List<ı> A -> List<ı> B.

Fixpoint ntail ı A (x : nat<ı>) : List A → List A :=
  ...

Check ntail.
ntail : ∀ ı ∀ ȷ. forall A, nat<ı> -> List<ȷ> A -> List<ȷ> A.
```

In a future Coq version . . .

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

Check map.
map : ∀ ı. (A -> B) -> List<ı> A -> List<ı> B.

Fixpoint ntail ı A (x : nat<ı>) : List A → List A :=
  ...

Check ntail.
ntail : ∀ ı ∀ ȷ1 ∀ ȷ2. ȷ2 ⊑ ȷ1 =>
        forall A, nat<ı> -> List<ȷ1> A -> List<ȷ2> 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

$$\text{case } (\text{cofix } f := M) \text{ of} \ldots \to \text{case } M[f := (\text{cofix } f := M)] \text{ of} \ldots$$

- 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 costructor $\mathsf{in} : U \to U$

$$u : U \qquad \mathrm{force} : U \to U$$

$$u \stackrel{\mathrm{def}}{=} \mathsf{cofix}\ u := \mathsf{in}\ u \qquad \mathrm{force} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\ \mathsf{in}\, x' \Rightarrow \mathsf{in}\, x'$$

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

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

$$\mathrm{eq} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\ \mathsf{in}\, x' \Rightarrow \mathrm{refl}$$

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

A note on coinduction with dependent types

- Example: consider a co-inductive type U with only one costructor $\mathsf{in} : U \to U$

$$u : U \qquad\qquad \mathrm{force} : U \to U$$

$$u \stackrel{\mathrm{def}}{=} \mathsf{cofix}\ u := \mathsf{in}\, u \qquad \mathrm{force} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, \mathsf{in}\, x' \Rightarrow \mathsf{in}\, x'$$

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

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

$$\mathrm{eq} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, \mathsf{in}\, x' \Rightarrow \mathrm{refl}$$

- Then, $\mathrm{eq}\, u : u = \mathsf{in}\, u$,

A note on coinduction with dependent types

- Example: consider a co-inductive type U with only one costructor $\mathsf{in} : U \to U$

$$u : U \qquad\qquad \mathrm{force} : U \to U$$

$$u \stackrel{\mathrm{def}}{=} \mathsf{cofix}\ u := \mathsf{in}\, u \qquad\qquad \mathrm{force} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, \mathsf{in}\, x' \Rightarrow \mathsf{in}\, x'$$

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

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

$$\mathrm{eq} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\, \mathsf{in}\, x' \Rightarrow \mathrm{refl}$$

- Then, $\mathrm{eq}\, u : u = \mathsf{in}\, u$, and $\mathrm{eq}\, u \to^* \mathrm{refl}$

A note on coinduction with dependent types

- Example: consider a co-inductive type U with only one costructor $\mathsf{in} : U \to U$

$$u : U \qquad \mathrm{force} : U \to U$$

$$u \stackrel{\mathrm{def}}{=} \mathsf{cofix}\ u := \mathsf{in}\ u \qquad \mathrm{force} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\ \mathsf{in}\, x' \Rightarrow \mathsf{in}\, x'$$

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

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

$$\mathrm{eq} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\ \mathsf{in}\, x' \Rightarrow \mathrm{refl}$$

- Then, $\mathrm{eq}\, u : u = \mathsf{in}\, u$, and $\mathrm{eq}\, u \to^* \mathrm{refl}$
- But $\mathrm{refl} : u = u$

A note on coinduction with dependent types

- Example: consider a co-inductive type U with only one costructor $\mathsf{in} : U \to U$

$$u : U \qquad\qquad \mathrm{force} : U \to U$$

$$u \stackrel{\mathrm{def}}{=} \mathsf{cofix}\ u := \mathsf{in}\ u \qquad\qquad \mathrm{force} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\ \mathsf{in}\, x' \Rightarrow \mathsf{in}\, x'$$

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

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

$$\mathrm{eq} \stackrel{\mathrm{def}}{=} \lambda x.\mathsf{case}\, x\, \mathsf{of}\ \mathsf{in}\, x' \Rightarrow \mathrm{refl}$$

- Then, $\mathrm{eq}\, u : u = \mathsf{in}\, u$, and $\mathrm{eq}\, u \to^* \mathrm{refl}$
- But $\mathrm{refl} : u = u$
- The types $u = u$ and $u = \mathsf{in}\, u$ are not convertible since there is no case forcing the unfolding of u.