Type-Based Productivity of Stream Definitions in the Calculus of Constructions

Jorge Luis Sacchini

Carnegie Mellon University

June 26, 2013

(Co-)Inductive Types in Type Theory

- Inductive types (e.g. lists, trees) are essential in Type Theory to model and reason about systems.
- 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
Least fixed point	Greatest fixed point
Recursive functions consume data	Corecursive functions produce data

Coinductive Types in Coq

- Streams:

$$\textbf{CoInductive}\ \text{stream}\ A := \text{cons} : A \to \text{stream}\ A \to \text{stream}\ A$$

Coinductive Types in Coq

- Strea

Empty as an inductive type

CoInductive stream $A :=$ cons : $A \to$ stream $A \to$ stream A

Coinductive Types in Coq

- Streams:

$$\textbf{CoInductive}\ \text{stream}\ A := \text{cons} : A \to \text{stream}\ A \to \text{stream}\ A$$

- Corecursive functions produce streams:

$$\text{zeroes} \stackrel{\text{def}}{=} \text{cofix}\ Z := \text{cons}(0, Z)$$

zeroes produce the stream:

$$\text{cons}(0, \text{cons}(0, \text{cons}(0, \ldots)))$$

Coinductive types

Inductive types	**Coinductive types**
Termination	

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.

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.

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:

$$\text{cofix } Z_1 := \text{cons}(0, Z_1)$$

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

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:

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

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

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:

$$\text{cofix } Z_1 := \text{cons}(0, Z_1)$$

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

Productivity

- Productivity checking is undecidable.

Productivity

- Productivity checking is undecidable.
- Proof assistants (Coq, Agda, ...) typically use syntactic methods to ensure productivity (and termination).

Productivity

- Productivity checking is undecidable.
- Proof assistants (Coq, Agda, ...) typically use syntactic methods to ensure productivity (and termination).
- In Coq, a corecursive function is productive if it is guarded, i.e.,

  every corecursive call is performed directly under a constructor

Productivity

- Productivity checking is undecidable.
- Proof assistants (Coq, Agda, ...) typically use syntactic methods to ensure productivity (and termination).
- In Coq, a corecursive function is productive if it is guarded, i.e.,

  every corecursive call is performed directly under a constructor

- Syntactic methods are limited in practice. E.g. in Coq:

Productivity

- Productivity checking is undecidable.
- Proof assistants (Coq, Agda, ...) typically use syntactic methods to ensure productivity (and termination).
- In Coq, a corecursive function is productive if it is guarded, i.e.,

  every corecursive call is performed directly under a constructor

- Syntactic methods are limited in practice. E.g. in Coq:

$$\mathsf{nats} \stackrel{\mathsf{def}}{=} \mathsf{cofix}\ \mathsf{nats} := \lambda n.\ \mathsf{cons}(n, \mathsf{nats}\ (1 + n))$$

Productivity

- Productivity checking is undecidable.
- Proof assistants (Coq, Agda, ...) typically use syntactic methods to ensure productivity (and termination).
- In Coq, a corecursive function is productive if it is guarded, i.e.,

  every corecursive call is performed directly under a constructor

- Syntactic methods are limited in practice. E.g. in Coq:

$$\text{nats} \stackrel{\text{def}}{=} \text{cofix nats} := \lambda n.\ \text{cons}(n, \text{nats}\ (1+n))$$ ✔

Productivity

- Productivity checking is undecidable.
- Proof assistants (Coq, Agda, ...) typically use syntactic methods to ensure productivity (and termination).
- In Coq, a corecursive function is productive if it is guarded, i.e.,

  every corecursive call is performed directly under a constructor

- Syntactic methods are limited in practice. E.g. in Coq:

$$\text{nats} \stackrel{\text{def}}{=} \text{cofix nats} := \lambda n.\ \text{cons}(n, \text{nats}\ (1+n)) \quad \checkmark$$

$$\text{nats} \stackrel{\text{def}}{=} \lambda n.\ \text{cofix nats} := \text{cons}(n, \text{map}\ (1+)\ \text{nats})$$

Productivity

- Productivity checking is undecidable.
- Proof assistants (Coq, Agda, ...) typically use syntactic methods to ensure productivity (and termination).
- In Coq, a corecursive function is productive if it is guarded, i.e.,

  every corecursive call is performed directly under a constructor

- Syntactic methods are limited in practice. E.g. in Coq:

$$\text{nats} \stackrel{\text{def}}{=} \text{cofix nats} := \lambda n.\ \text{cons}(n, \text{nats}\ (1+n)) \quad \checkmark$$

$$\text{nats} \stackrel{\text{def}}{=} \lambda n.\ \text{cofix nats} := \text{cons}(n, \text{map}\ (1+)\ \text{nats}) \quad \times$$

Syntactic Methods for Termination and Productivity

Inductive types	**Coinductive types**
Termination	Productivity

Syntactic Methods for Termination and Productivity

Inductive types	**Coinductive types**
Termination	Productivity

Syntactic methods in Coq have several limitations that often appear in practice.

Syntactic Methods for Termination and Productivity

Inductive types	**Coinductive types**
Termination	Productivity

Syntactic methods in Coq have several limitations that often appear in practice.

- Difficult to understand.

Syntactic Methods for Termination and Productivity

Inductive types	**Coinductive types**
Termination	Productivity

Syntactic methods in Coq have several limitations that often appear in practice.

- Difficult to understand.
- Difficult to implement (the termination and productivity checker in Coq is one of the weakest points in the kernel).

Syntactic Methods for Termination and Productivity

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

Syntactic methods in Coq have several limitations that often appear in practice.

- Difficult to understand.
- Difficult to implement (the termination and productivity checker in Coq is one of the weakest points in the kernel).
- Different procedures for inductive and coinductive types.

Syntactic Methods for Termination and Productivity

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

Syntactic methods in Coq have several limitations that often appear in practice.

- Difficult to understand.
- Difficult to implement (the termination and productivity checker in Coq is one of the weakest points in the kernel).
- Different procedures for inductive and coinductive types.

Type-based methods provide a better framework for termination and productivity checking.

Type-Based Productivity

- Types annotated with size information

$$\text{stream}^s\ A$$

is the type of streams (of type A) where at least s elements can be produced.

Type-Based Productivity

- Types annotated with size information

$$\text{stream}^{s}\ A$$

  is the type of streams (of type A) where at least s elements can be produced.

- In this work, sizes (stages) are defined by a simple algebra:

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

Type-Based Productivity

- Types annotated with size information

$$\mathsf{stream}^{s}\ A$$

  is the type of streams (of type A) where at least s elements can be produced.

- In this work, sizes (stages) are defined by a simple algebra:

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

- Substage relation (reflexive and transitive):

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

Type-Based Productivity

- Types annotated with size information

$$\mathsf{stream}^{s}\ A$$

  is the type of streams (of type A) where at least s elements can be produced.

- In this work, sizes (stages) are defined by a simple algebra:

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

- Substage relation (reflexive and transitive):

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

  but stage variables are incomparable; i.e. $\imath \not\sqsubseteq \jmath$

Type-Based Productivity

- The substage relation induces a subtype relation:

$$\frac{r \sqsubseteq s \qquad T \leq U}{\mathsf{stream}^s\ T \leq \mathsf{stream}^r\ U}$$

Type-Based Productivity

- The substage relation induces a subtype relation:

$$\frac{r \sqsubseteq s \qquad T \leq U}{\mathsf{stream}^s\ T \leq \mathsf{stream}^r\ U}$$

- Subtyping is contravariant on products domain and covariant in their codomain

$$\frac{U_1 \leq T_1 \qquad T_2 \leq U_2}{\Pi x : T_1.T_2 \leq \Pi x : U_1.U_2}$$

Type-Based Productivity

- Typing rules for constructors:

$$\frac{\vdash N : A \qquad \vdash M : \mathsf{stream}^{s}\ A}{\vdash \mathsf{cons}(N, M) : \mathsf{stream}^{\widehat{s}}\ A}$$

Type-Based Productivity

- Typing rules for constructors:

$$\frac{\vdash N : A \qquad \vdash M : \mathsf{stream}^{s}\ A}{\vdash \mathsf{cons}(N, M) : \mathsf{stream}^{\widehat{s}}\ A}$$

- Corecursive functions produce at least one more element in each iteration:

$$\frac{(f : \mathsf{stream}\ \ A) \vdash M : \mathsf{stream}\ \ A}{\vdash (\mathsf{cofix}\ f := M) : \mathsf{stream}\ \ A}$$

Type-Based Productivity

- Typing rules for constructors:

$$\frac{\vdash N : A \qquad \vdash M : \mathsf{stream}^{s}\ A}{\vdash \mathsf{cons}(N, M) : \mathsf{stream}^{\widehat{s}}\ A}$$

- Corecursive functions produce at least one more element in each iteration:

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

Type-Based Productivity

- Typing rules for constructors:

$$\frac{\vdash N : A \qquad \vdash M : \mathsf{stream}^{s}\ A}{\vdash \mathsf{cons}(N, M) : \mathsf{stream}^{\widehat{s}}\ A}$$

- Corecursive functions produce at least one more element in each iteration:

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

- The (almost) full rule:

$$\frac{\begin{array}{c} T \equiv \Pi\Delta.\mathsf{stream}^{\imath}\ U \qquad \imath\ \mathsf{neg}\ \Delta \qquad \imath \notin \Gamma, U, M \\ \Gamma \vdash T : \mathsf{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath] \end{array}}{\Gamma \vdash (\mathsf{cofix}\ f : |T| := M) : T[s/\imath]}$$

Type-Based Productivity

- Typing rules for constructors:

$$\frac{\vdash N : A \qquad \vdash M : \mathsf{stream}^{s}\ A}{\vdash \mathsf{cons}(N, M) : \mathsf{stream}^{\widehat{s}}\ A}$$

- Corecursive functions produce at least one more element in each iteration:

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

- The (almost) full rule:

$\mathsf{stream}^{\imath}\ A \to \mathsf{stream}^{\imath}\ A$

$$\frac{\begin{array}{c} T \equiv \Pi\Delta.\mathsf{stream}^{\imath}\ U \qquad \imath \text{ neg } \Delta \qquad \imath \notin \Gamma, U, M \\ \Gamma \vdash T : \mathsf{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath] \end{array}}{\Gamma \vdash (\mathsf{cofix}\ f : |T| := M) : T[s/\imath]}$$

Type-Based Productivity

Type-based approaches to productivity (and termination) have several advantages over syntactic-based methods:

- More expressive.
- Easier to understand.
- Easier to implement.
- Treats inductive and coinductive types uniformly.

Contributions

- We define an extension of CC+universes and a stream type (a subset of Coq) with a type-based criterion for ensuring productivity.
- We prove strong normalization and logical consistency using a Λ-set model.
- We define a size-inferring algorithm (constraint-based) following previous work by Barthe et al.

Contributions

- We define an extension of CC+universes and a stream type (a subset of Coq) with a type-based criterion for ensuring productivity.
- We prove strong normalization and logical consistency using a Λ-set model.
- We define a size-inferring algorithm (constraint-based) following previous work by Barthe et al.

CC with Type-Based Productivity

$$
\begin{aligned}
T ::= \ & \mathsf{Type}_i \mid \mathsf{Prop} \mid \Pi x : T.T \\
& \mid \lambda x : T^{\circ}.T \mid TT \\
& \mid \mathsf{stream}^s\ T \mid \mathsf{cons}(T, T) \mid \\
& \mid \mathsf{case}_T\ T\ \mathsf{of}\ \mathsf{cons}(x, y) \Rightarrow T \\
& \mid \mathsf{cofix}\ f : T^{\star} := T
\end{aligned}
$$

- Based on a long-history of type-based systems (Barthe et al.).
- Coinductive types as in Coq.
- Inherits some good properties of these systems ...

CC with Type-Based Productivity

- . . . and some bad ones.

CC with Type-Based Productivity

- . . . and some bad ones.
- Subject reduction is not valid (like in Coq).

CC with Type-Based Productivity

- . . . and some bad ones.
- Subject reduction is not valid (like in Coq).
- Unrestricted unfolding is not strongly normalizing:

$$\text{cofix } f := M \quad \rightarrow \quad M[\text{cofix } f := M/f]$$

CC with Type-Based Productivity

- . . . and some bad ones.
- Subject reduction is not valid (like in Coq).
- Unrestricted unfolding is not strongly normalizing:

$$\text{cofix } f := M \quad \to \quad M[\text{cofix } f := M/f]$$

- Unfolding is restricted to occur inside case analysis:

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

Does not satisfy SR, but it is SN.

CC with Type-Based Productivity

- . . . and some bad ones.
- Subject reduction is not valid (like in Coq).
- Unrestricted unfolding is not strongly normalizing:

$$\text{cofix } f := M \quad \rightarrow \quad M[\text{cofix } f := M/f]$$

- Unfolding is restricted to occur inside case analysis:

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

  Does not satisfy SR, but it is SN.
- (Giménez) Conversion is done using the unrestricted unfolding, and evaluation using the restricted unfolding.

CC with Type-Based Productivity

- . . . and some bad ones.
- Subject reduction is not valid (like in Coq).
- Unrestricted unfolding is not strongly normalizing:

$$\text{cofix } f := M \quad \rightarrow \quad M[\text{cofix } f := M/f]$$

- Unfolding is restricted to occur inside case analysis:

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

  Does not satisfy SR, but it is SN.
- (Giménez) Conversion is done using the unrestricted unfolding, and evaluation using the restricted unfolding.
- Satisfies SR, SN, but not decidability of type-checking.

CC with Type-Based Productivity

- ... and some bad ones.
- Subject reduction is not valid (like in Coq).
- Unrestricted unfolding is not strongly normalizing:

$$\text{cofix } f := M \quad \rightarrow \quad M[\text{cofix } f := M/f]$$

- Unfolding is restricted to occur inside case analysis:

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

  Does not satisfy SR, but it is SN.
- (Giménez) Conversion is done using the unrestricted unfolding, and evaluation using the restricted unfolding.
- Satisfies SR, SN, but not decidability of type-checking.
- In an implementation (e.g. Coq), we use restricted unfolding exclusively. We lose SR, but gain decidability.

Type-Based Productivity

Examples

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

$$\mathsf{map} \stackrel{\mathsf{def}}{=} \lambda f.\mathsf{cofix}\ M : \mathsf{stream}^\star\ A \to \mathsf{stream}^\star\ B := \lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ (h, t) \Rightarrow \mathsf{cons}(f\ h, M\ t)$$

Type-Based Productivity

Examples

Size-preserving functions

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

Guarded

$$\mathsf{map} \stackrel{\mathsf{def}}{=} \lambda f.\mathsf{cofix}\ M : \mathsf{stream}^\star\ A \to \mathsf{stream}^\star\ B$$
$$\lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ (h, t) \Rightarrow \mathsf{cons}(f\ h, M\ t)$$

Type-Based Productivity

Examples

$$\mathsf{map} : (A \to B) \to \mathsf{stream}^{\infty}\ A \to \mathsf{stream}^{s}\ B$$

$$\mathsf{map} \stackrel{\mathsf{def}}{=} \lambda f.\mathsf{cofix}\ M : \mathsf{stream}\ A \to \mathsf{stream}^{\star}\ B := \lambda x.\ \mathsf{case}\ x\ \mathsf{of}\ (h, t) \Rightarrow \mathsf{cons}(f\ h, M\ t)$$

Type-Based Productivity

Examples

$$\mathsf{map} : (A \to B) \to \mathsf{stream}^s\, A \to \mathsf{stream}^s\, B$$

$$\mathsf{merge} : \mathsf{stream}^s\, \mathsf{nat} \to \mathsf{stream}^s\, \mathsf{nat} \to \mathsf{stream}^s\, \mathsf{nat}$$

Type-Based Productivity

Examples

$$\mathsf{map} : (A \to B) \to \mathsf{stream}^s\, A \to \mathsf{stream}^s\, B$$

$$\mathsf{merge} : \mathsf{stream}^s\, \mathsf{nat} \to \mathsf{stream}^s\, \mathsf{nat} \to \mathsf{stream}^s\, \mathsf{nat}$$

$$\begin{aligned}\mathsf{merge} \stackrel{\mathsf{def}}{=} \mathsf{cofix}\ M : \mathsf{stream}^\star\, \mathsf{nat} \to \mathsf{stream}^\star\, \mathsf{nat} \to \mathsf{stream}^\star\, \mathsf{nat} :=\\ \lambda x_1\, x_2.\ \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{cons}(h_1, t_1) \Rightarrow\\ \mathsf{case}\ x_2\ \mathsf{of}\ \mathsf{cons}(h_2, t_2) \Rightarrow\\ \mathsf{if}\ h_1 \leq h_2\ \mathsf{then}\ \mathsf{cons}(h_1, M\, t_1\, x_2)\\ \mathsf{else}\ \mathsf{cons}(h_2, M\, x_1\, t_2)\end{aligned}$$

Type-Based Productivity

Examples

$$\mathsf{map} : (A \to B) \to \mathsf{stream}^s\, A \to \mathsf{stream}^s\, B$$

$$\mathsf{merge} : \mathsf{stream}^s\, \mathsf{nat} \to \mathsf{stream}^s\, \mathsf{nat} \to \mathsf{stream}^s\, \mathsf{nat}$$

$$\begin{aligned}\mathsf{merge} \stackrel{\mathsf{def}}{=} \mathsf{cofix}\ M : \mathsf{stream}^\star\, \mathsf{nat} \to \mathsf{stream}^\star\, \mathsf{nat} \to \mathsf{stream}^\star\, \mathsf{nat} :=\\ \lambda x_1\, x_2.\ \mathsf{case}\ x_1\ \mathsf{of}\ \mathsf{cons}(h_1, t_1) \Rightarrow\\ \mathsf{case}\ x_2\ \mathsf{of}\ \mathsf{cons}(h_2, t_2) \Rightarrow\\ \mathsf{if}\ h_1 \leq h_2\ \mathsf{then}\ \mathsf{cons}(h_1, M\, t_1\, x_2)\\ \mathsf{else}\ \mathsf{cons}(h_2, M\, x_1\, t_2)\end{aligned}$$

Guarded

Type-Based Productivity

Examples

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

$$\mathsf{merge} : \mathsf{stream}^s\ \mathsf{nat} \to \mathsf{stream}^s\ \mathsf{nat} \to \mathsf{stream}^s\ \mathsf{nat}$$

$$\mathsf{hamming} : \mathsf{stream}^s\ \mathsf{nat}$$

Type-Based Productivity

Examples

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

$$\mathsf{merge} : \mathsf{stream}^s\ \mathsf{nat} \to \mathsf{stream}^s\ \mathsf{nat} \to \mathsf{stream}^s\ \mathsf{nat}$$

$$\mathsf{hamming} : \mathsf{stream}^s\ \mathsf{nat}$$

$$\begin{aligned}\mathsf{hamming} \stackrel{\mathsf{def}}{=}\ & \mathsf{cofix}\ H : \mathsf{stream}^\star \mathsf{nat} := \\ & \mathsf{cons}(1, \mathsf{merge}\ (\mathsf{map}\ (2*)\ H) \\ & \qquad (\mathsf{merge}\ (\mathsf{map}\ (3*)\ H) \\ & \qquad\quad (\mathsf{map}\ (5*)\ H)))\end{aligned}$$

Type-Based Productivity

Examples

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

$$\mathsf{merge} : \mathsf{stream}^s\ \mathsf{nat} \to \mathsf{stream}^s\ \mathsf{nat} \to \mathsf{stream}^s\ \mathsf{nat}$$

$$\mathsf{hamming} : \mathsf{stream}^s\ \mathsf{nat}$$

$$\mathsf{hamming} \stackrel{\mathsf{def}}{=} \mathsf{cofix}\ H : \mathsf{stream}^\star \mathsf{nat} := \mathsf{cons}(1, \mathsf{merge}\ (\mathsf{map}\ (2*)\ H)\ (\mathsf{merge}\ (\mathsf{map}\ (3*)\ H)\ (\mathsf{map}\ (5*)\ H))))$$

Not guarded

Contributions

- We define an extension of CC+universes and a stream type (a subset of Coq) with a type-based criterion for ensuring productivity.
- We prove strong normalization and logical consistency using a Λ-set model.
- We define a size-inferring algorithm (constraint-based) following previous work by Barthe et al.

Contributions

- We define an extension of CC+universes and a stream type (a subset of Coq) with a type-based criterion for ensuring productivity.
- We prove strong normalization and logical consistency using a Λ-set model.
- We define a size-inferring algorithm (constraint-based) following previous work by Barthe et al.

Strong Normalization

- Model based on Λ-sets (Altenkirch).
- Relatively easy to understand and expressiveL
  - Impredicativity (not treated in this work for space reasons).
  - Inductive and coinductive types.
- Types are interpreted as Λ-sets:

$$(X, \models)$$

  - X is a set
  - $\models \subseteq \mathsf{SN} \times X$ is a realizability relation

Strong Normalization

- Model based on Λ-sets (Altenkirch).
- Relatively easy to understand and expressiveL
  - Impredicativity (not treated in this work for space reasons).
  - Inductive and coinductive types.
- Types are interpreted as Λ-sets:

$$(X, \models)$$

  - X is a set
  - $\models \subseteq \mathsf{SN} \times X$ is a realizability relation

$$\text{Soundness} : \Gamma \vdash M : T \Rightarrow \forall \gamma \in [\Gamma].[M](\gamma) \in [T](\gamma)$$

$$\text{Realizability:}\ \Gamma \vdash M : T \Rightarrow \exists \gamma . M \models [M](\gamma)$$

(realizers are SN by definition)

Strong Normalization

- Λ-set interpretation ignores sizes.
- Set-theoretical interpretation:

$$
\begin{aligned}
[\lambda x : T.M]_\gamma &= \alpha \in [T] \mapsto [M]_{\gamma,\alpha} \\
[M\ N]_\gamma &= [M]_\gamma\,[N]_\gamma \\
[\Pi x : T.U]_\gamma &= \prod_{\alpha \in [T](\gamma)} [U]_{\gamma,\alpha} \\
&\vdots
\end{aligned}
$$

Strong Normalization

- Λ-set interpretation ignores sizes.
- Set-theoretical interpretation:

$$
\begin{aligned}
[\lambda x : T.M]_\gamma &= \alpha \in [T] \mapsto [M]_{\gamma,\alpha} \\
[M\ N]_\gamma &= [M]_\gamma\,[N]_\gamma \\
[\Pi x : T.U]_\gamma &= \prod_{\alpha \in [T](\gamma)} [U]_{\gamma,\alpha} \\
&\vdots \\
[\text{stream } T]_\gamma &= \{(\alpha_0, \alpha_1, \ldots) : \alpha_i \in [T]_\gamma\}
\end{aligned}
$$

Strong Normalization

- Λ-set interpretation ignores sizes.
- Set-theoretical interpretation:

$$[\lambda x : T.M]_\gamma = \alpha \in [T] \mapsto [M]_{\gamma,\alpha}$$
$$[M\ N]_\gamma = [M]_\gamma\,[N]_\gamma$$
$$[\Pi x : T.U]_\gamma = \prod_{\alpha \in [T](\gamma)} [U]_{\gamma,\alpha}$$

Finite and infinite sequences

$$[\text{stream } T]_\gamma = \{(\alpha_0, \alpha_1, \ldots) : \alpha_i \in [T]_\gamma\}$$

Strong Normalization

- Λ-set interpretation ignores sizes.
- Set-theoretical interpretation:

$$
\begin{aligned}
[\lambda x : T.M]_\gamma &= \alpha \in [T] \mapsto [M]_{\gamma,\alpha} \\
[M\ N]_\gamma &= [M]_\gamma\,[N]_\gamma \\
[\Pi x : T.U]_\gamma &= \prod_{\alpha \in [T](\gamma)} [U]_{\gamma,\alpha} \\
&\vdots \\
[\text{stream } T]_\gamma &= \{(\alpha_0, \alpha_1, \ldots) : \alpha_i \in [T]_\gamma\} \\
[\text{cons}(M, N)]_\gamma &= \langle [M]_\gamma, [N]_\gamma \rangle
\end{aligned}
$$

Strong Normalization

- Λ-set interpretation ignores sizes.
- Set-theoretical interpretation:

$$
\begin{aligned}
[\lambda x : T.M]_\gamma &= \alpha \in [T] \mapsto [M]_{\gamma,\alpha} \\
[M\ N]_\gamma &= [M]_\gamma\,[N]_\gamma \\
[\Pi x : T.U]_\gamma &= \prod_{\alpha \in [T](\gamma)} [U]_{\gamma,\alpha} \\
&\vdots \\
[\text{stream } T]_\gamma &= \{(\alpha_0, \alpha_1, \ldots) : \alpha_i \in [T]_\gamma\} \\
[\text{cons}(M,N)]_\gamma &= \langle [M]_\gamma, [N]_\gamma \rangle \\
[\text{cofix } f : T := M]_\gamma &= \epsilon(F, P(F)) \\
&\qquad \text{where } P(F \in [T]_\gamma) \text{ is } F = [M]_{\gamma,F}
\end{aligned}
$$

Strong Normalization

- Λ-set interpretation ignores sizes.
- Set-theoretical interpretation:

$$[\lambda x : T.M]_\gamma = \alpha \in [T] \mapsto [M]_{\gamma,\alpha}$$
$$[M\ N]_\gamma = [M]_\gamma\,[N]_\gamma$$
$$[\Pi x : T.U]_\gamma = \prod_{\alpha \in [T](\gamma)} [U]_{\gamma,\alpha}$$
$$\vdots$$
$$[\text{stream } T]_\gamma = \{(\alpha_0, \alpha_1, \ldots) : \alpha_i \in [T]_\gamma\}$$
$$[\text{cofix } f : T := M]_\gamma = \epsilon(F, P(F))$$
$$\text{where } P(F \in [T]_\gamma) \text{ is } F = [M]_{\gamma,F}$$

Hilbert's choice operator

Strong Normalization

- Λ-set interpretation ignores sizes.
- Realizability interpretation:

$$M \models_{[\Pi x{:}T.U]} f \iff N \models_{[T]} \alpha \Rightarrow M\,N \models_{[U](\alpha)} f(\alpha)$$

$$\vdots$$

$$M \models_{[\mathsf{stream}\ T]} (\alpha_0, \alpha_1, \ldots) \iff \downarrow M \to^* \mathsf{cons}(N_1, N_2) \wedge N_1 \models_{[T]} \alpha_0 \wedge N_2 \models_{[\mathsf{stream}\ T]} (\alpha_1, \ldots)$$

Strong Normalization

- We extend the model with a relational interpretation for types.
- Makes sense of size annotations.

Strong Normalization

- We extend the model with a relational interpretation for types.
- Makes sense of size annotations.
- Streams are represented by pairs of (set-theoretical) sequences (finite and infinite):

$$[\![\mathsf{stream}^s\ T]\!] = \{(\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in [\![T]\!]\}$$

If $s = \infty$, the relation reduces to the identity relation.

Strong Normalization

- We extend the model with a relational interpretation for types.
- Makes sense of size annotations.
- Streams are represented by pairs of (set-theoretical) sequences (finite and infinite):

$$[\![\mathsf{stream}^s\ T]\!] = \{(\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in [\![T]\!]\}$$

  If $s = \infty$, the relation reduces to the identity relation.
- Cofixpoint rule defines a contractive function in the relational model:

$$\frac{(f : \mathsf{stream}^{\imath}\ T) \vdash M : \mathsf{stream}^{\widehat{\imath}}\ T}{\vdash \mathsf{cofix}\ f := M : \mathsf{stream}^{s}\ T}$$

Strong Normalization

- We extend the model with a relational interpretation for types.
- Makes sense of size annotations.
- Streams are represented by pairs of (set-theoretical) sequences (finite and infinite):

$$[\![\mathsf{stream}^s\ T]\!] = \{(\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in [\![T]\!]\}$$

  If $s = \infty$, the relation reduces to the identity relation.
- Cofixpoint rule defines a contractive function in the relational model:

$$\frac{(f : \mathsf{stream}^{\imath}\ T) \vdash M : \mathsf{stream}^{\widehat{\imath}}\ T}{\vdash \mathsf{cofix}\ f := M : \mathsf{stream}^{s}\ T}$$

$$(\alpha_1, \alpha_2)$$

Strong Normalization

- We extend the model with a relational interpretation for types.
- Makes sense of size annotations.
- Streams are represented by pairs of (set-theoretical) sequences (finite and infinite):

$$[\![\mathsf{stream}^s\ T]\!] = \{(\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in [\![T]\!]\}$$

If $s = \infty$, the relation reduces to the identity relation.

- Cofixpoint rule defines a contractive function in the relational model:

$$\frac{(f : \mathsf{stream}^{\imath}\ T) \vdash M : \mathsf{stream}^{\widehat{\imath}}\ T}{\vdash \mathsf{cofix}\ f := M : \mathsf{stream}^{s}\ T}$$

Coincide up to $\imath$

(α_1, α_2)

Strong Normalization

- We extend the model with a relational interpretation for types.
- Makes sense of size annotations.
- Streams are represented by pairs of (set-theoretical) sequences (finite and infinite):

$$[\![\mathsf{stream}^s\ T]\!] = \{(\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in [\![T]\!]\}$$

  If $s = \infty$, the relation reduces to the identity relation.
- Cofixpoint rule defines a contractive function in the relational model:

$$\frac{(f : \mathsf{stream}^{\imath}\ T) \vdash M : \mathsf{stream}^{\widehat{\imath}}\ T}{\vdash \mathsf{cofix}\ f := M : \mathsf{stream}^{s}\ T}$$

$$(\alpha_1, \alpha_2) \rightsquigarrow ([M](\alpha_1), [M](\alpha_2))$$

Strong Normalization

- We extend the model with a relational interpretation for types.
- Makes sense of size annotations.
- Streams are represented by pairs of (set-theoretical) sequences (finite and infinite):

$$[\![\mathsf{stream}^s\ T]\!] = \{(\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in [\![T]\!]\}$$

  If $s = \infty$, the relation reduces to the identity relation.
- Cofixpoint rule defines a contractive function in the relational model:

$$\frac{(f : \mathsf{stream}^{\imath}\ T) \vdash M : \mathsf{stream}^{\widehat{\imath}}\ T}{\vdash \mathsf{cofix}\ f := M : \mathsf{stream}^{s}\ T}$$

Coincide up to $\imath + 1$

$$(\alpha_1, \alpha_2) \rightsquigarrow ([M](\alpha_1), [M](\alpha_2))$$

Strong Normalization

- $[M]$ is a contractive function

$$\frac{(f : \mathsf{stream}^{\imath}\ T) \vdash M : \mathsf{stream}^{\widehat{\imath}}\ T}{\vdash \mathsf{cofix}\ f := M : \mathsf{stream}^{s}\ T}$$

- Basis for interpreting cofix

$$\begin{array}{lcccccc} \alpha & = & \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \ldots \\ \beta & = & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots \end{array}$$

Strong Normalization

- $[M]$ is a contractive function

$$\frac{(f : \text{stream}^{\imath}\ T) \vdash M : \text{stream}^{\widehat{\imath}}\ T}{\vdash \text{cofix}\ f := M : \text{stream}^{s}\ T}$$

- Basis for interpreting cofix

$$\begin{array}{lcccccc}
\alpha & = & \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \ldots \\
\beta & = & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots \\
 & & & \wr\,[M] & & & \\
[M](\alpha) & = & & \alpha_{12} & \alpha_{13} & \alpha_{14} & \ldots \\
[M](\beta) & = & \alpha_{11} & \beta_{12} & \beta_{13} & \beta_{14} & \ldots
\end{array}$$

Same first element

Strong Normalization

- $[M]$ is a contractive function

$$\frac{(f : \text{stream}^{\imath}\ T) \vdash M : \text{stream}^{\widehat{\imath}}\ T}{\vdash \text{cofix}\ f := M : \text{stream}^{s}\ T}$$

- Basis for interpreting cofix

$$\begin{array}{lcccccc}
\alpha & = & \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \ldots \\
\beta & = & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots \\
 & & & \wr\, [M]^2 & & & \\
[M]^2(\alpha) & = & & & \alpha_{23} & \alpha_{24} & \ldots \\
[M]^2(\beta) & = & \alpha_{21} & \alpha_{22} & \beta_{23} & \beta_{24} & \ldots
\end{array}$$

Strong Normalization

- $[M]$ is a contractive function

$$\frac{(f : \text{stream}^{\imath}\ T) \vdash M : \text{stream}^{\widehat{\imath}}\ T}{\vdash \text{cofix}\ f := M : \text{stream}^{s}\ T}$$

- Basis for interpreting cofix

$$\begin{array}{lcccccc}
\alpha & = & \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \ldots \\
\beta & = & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots \\
 & & & \wr\, [M]^3 & & & \\
[M]^3(\alpha) & = & & & & \alpha_{34} & \ldots \\
[M]^3(\beta) & = & \alpha_{31} & \alpha_{32} & \alpha_{33} & \beta_{34} & \ldots
\end{array}$$

Strong Normalization

- $[M]$ is a contractive function

$$\frac{(f : \text{stream}^{\imath}\ T) \vdash M : \text{stream}^{\widehat{\imath}}\ T}{\vdash \text{cofix}\ f := M : \text{stream}^{s}\ T}$$

- Basis for interpreting cofix

$$\begin{array}{rcccccc} \alpha & = & \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \ldots \\ \beta & = & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots \\ & & & \downarrow [M]^{\infty} & & & \\ [M]^{\infty}(\alpha) & = & \alpha_{\infty_1} & \alpha_{\infty_2} & \alpha_{\infty_3} & \alpha_{\infty_4} & \ldots \end{array}$$

Unique stream

Strong Normalization

- $[M]$ is a contractive function

$$\frac{(f : \text{stream}^{\imath}\ T) \vdash M : \text{stream}^{\widehat{\imath}}\ T}{\vdash \text{cofix}\ f := M : \text{stream}^{s}\ T}$$

- Basis for interpreting cofix

$$\begin{array}{rcccccc} \alpha & = & \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \ldots \\ \beta & = & \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots \\ & & \multicolumn{4}{c}{\wr\ [M]^{\infty}} & \\ [M]^{\infty}(\alpha) & = & \alpha_{\infty_1} & \alpha_{\infty_2} & \alpha_{\infty_3} & \alpha_{\infty_4} & \ldots \end{array}$$

Unique stream

- This unique stream has all the desired properties, including invariance under reduction.

Strong Normalization

- The Λ-set model is sound.

  Soundness If $\Gamma \vdash M : T$ and $(\gamma_1, \gamma_2) \in [\![\Gamma]\!]$, then $[M]_{\gamma_1,\gamma_2} \in [\![T]\!]_{\gamma_1,\gamma_2}$.

  Realizability If $\Gamma \vdash M : T$ and $(\gamma_1, \gamma_2) \in [\![\Gamma]\!]$ and $\theta \models (\gamma_1, \gamma_2)$, then $\theta M \models [M]_{\gamma_1,\gamma_2}$ in $[\![T]\!]_{\gamma_1,\gamma_2}$.

- As a corollary, we can prove logical consistency:

  There is no term M such that $\vdash M : (\Pi X : \mathrm{Type}_0.X)$.

Contributions

- We define an extension of CC+universes and a stream type (a subset of Coq) with a type-based criterion for ensuring productivity.
- We prove strong normalization and logical consistency using a Λ-set model.
- We define a size-inferring algorithm (constraint-based) following previous work by Barthe et al.

Contributions

- We define an extension of CC+universes and a stream type (a subset of Coq) with a type-based criterion for ensuring productivity.
- We prove strong normalization and logical consistency using a Λ-set model.
- We define a size-inferring algorithm (constraint-based) following previous work by Barthe et al.

Size-inference algorithm

- Given a term without annotations, the algorithm returns an annotated term and a set of constraints on those annotations

Size-inference algorithm

- Given a term without annotations, the algorithm returns an annotated term and a set of constraints on those annotations
- Example: given

$$\begin{array}{r} (f : \mathsf{stream}\, A \to \mathsf{stream}\, A) \\ (y : \mathsf{stream}\, A), (x : A) \end{array} \quad \vdash \quad \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ (\mathsf{x},\ \mathsf{y})))$$

Size-inference algorithm

- Given a term without annotations, the algorithm returns an annotated term and a set of constraints on those annotations
- Example: given

$$\begin{array}{rcl} (f : \mathsf{stream}\, A \to \mathsf{stream}\, A) \\ (y : \mathsf{stream}\, A), (x : A) \end{array} \quad \vdash \quad \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ (\mathsf{x},\ \mathsf{y})))$$

$$\Big\downarrow$$

$$\begin{array}{rcl} (f : \mathsf{stream}^{\alpha}\, A \to \mathsf{stream}^{\beta}\, A) \\ (y : \mathsf{stream}^{\gamma}\, A), (x : A) \end{array} \quad \vdash \quad \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ \ (\mathsf{x},\ \mathsf{y})))$$

Size-inference algorithm

- Given a term without annotations, the algorithm returns an annotated term and a set of constraints on those annotations
- Example: given

$$(f : \mathsf{stream}\, A \to \mathsf{stream}\, A),\ (y : \mathsf{stream}\, A), (x : A) \quad \vdash \quad \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ (\mathsf{x},\ \mathsf{y})))$$

$$\Big\downarrow$$

$$\mathsf{cons}(x, y) : \mathsf{stream}^{\widehat{\gamma}}\, A$$

$$(f : \mathsf{stream}^{\alpha}\, A \to \mathsf{stream}^{\beta}\, A),\ (y : \mathsf{stream}^{\gamma}\, A), (x : A) \quad \vdash \quad \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ (\mathsf{x},\ \mathsf{y})))$$

Size-inference algorithm

- Given a term without annotations, the algorithm returns an annotated term and a set of constraints on those annotations
- Example: given

$$(f : \mathsf{stream}\, A \to \mathsf{stream}\, A),\ (y : \mathsf{stream}\, A), (x : A) \quad \vdash \quad \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ (\mathsf{x},\ \mathsf{y})))$$

$$\Big\downarrow$$

$$\mathsf{cons}(x, y) : \mathsf{stream}^{\widehat{\gamma}}\, A$$

$$(f : \mathsf{stream}^{\alpha}\, A \to \mathsf{stream}^{\beta}\, A),\ (y : \mathsf{stream}^{\gamma}\, A), (x : A) \quad \vdash \quad \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ (\mathsf{x},\ \mathsf{y})))$$

$$\alpha \sqsubseteq \widehat{\gamma}, \alpha \sqsubseteq \beta$$

Size-inference algorithm

- Given a term without annotations, the algorithm returns an annotated term and a set of constraints on those annotations
- Example: given

$$\begin{array}{rcl} (f : \mathsf{stream}\, A \to \mathsf{stream}\, A) & & \\ (y : \mathsf{stream}\, A), (x : A) & \vdash & \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ (\mathsf{x},\ \mathsf{y}))) \end{array}$$

$$\Big\downarrow$$

$$\begin{array}{rcl} (f : \mathsf{stream}^{\alpha}\, A \to \mathsf{stream}^{\beta}\, A) & & \\ (y : \mathsf{stream}^{\gamma}\, A), (x : A) & \vdash & \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ \ (\mathsf{x},\ \mathsf{y}))) \\ & & \alpha \sqsubseteq \widehat{\gamma}, \alpha \sqsubseteq \beta \end{array}$$

Any stage substitution ρ satisfying these constraints gives a valid typing judgment.

Size-inference algorithm

- Given a term without annotations, the algorithm returns an annotated term and a set of constraints on those annotations
- Example: given

$$\begin{array}{rcl} (f : \mathsf{stream}\, A \to \mathsf{stream}\, A) \\ (y : \mathsf{stream}\, A), (x : A) \end{array} \vdash \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ (\mathsf{x},\ \mathsf{y})))$$

$$\Big\downarrow$$

$$\begin{array}{rcl} (f : \mathsf{stream}^{\alpha}\, A \to \mathsf{stream}^{\beta}\, A) \\ (y : \mathsf{stream}^{\gamma}\, A), (x : A) \end{array} \vdash \mathsf{f}\,(\mathsf{f}\,(\mathsf{cons}\ \ (\mathsf{x},\ \mathsf{y})))$$

$$\alpha \sqsubseteq \widehat{\gamma}, \alpha \sqsubseteq \beta$$

Any stage substitution ρ satisfying these constraints gives a valid typing judgment.
The substitution $\rho(\alpha) = \infty$, $\forall \alpha$, satisfies any constraint set.

Size-inference algorithm

- The algorithm is given by two judgments (inputs, outputs):
  - $C, \Gamma \vdash M^\circ \rightsquigarrow C', M \Rightarrow T$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
  - $C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C', M$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$

Size-inference algorithm

- The algorithm is given by two judgments (inputs, outputs):
  - $C, \Gamma \vdash M^\circ \rightsquigarrow C', M \Rightarrow T$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
  - $C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C', M$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
- Check rule:

$$\frac{}{C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow}$$

Size-inference algorithm

- The algorithm is given by two judgments (inputs, outputs):
  - $C, \Gamma \vdash M^\circ \rightsquigarrow C', M \Rightarrow T$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
  - $C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C', M$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
- Check rule:

$$\frac{C, \Gamma \vdash M^\circ \rightsquigarrow C_1, M \Rightarrow T_1}{C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow}$$

Size-inference algorithm

- The algorithm is given by two judgments (inputs, outputs):
  - $C, \Gamma \vdash M^\circ \rightsquigarrow C', M \Rightarrow T$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
  - $C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C', M$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
- Check rule:

$$\frac{C, \Gamma \vdash M^\circ \rightsquigarrow C_1, M \Rightarrow T_1}{C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow}$$

Size-inference algorithm

- The algorithm is given by two judgments (inputs, outputs):
  - $C, \Gamma \vdash M^\circ \rightsquigarrow C', M \Rightarrow T$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
  - $C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C', M$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
- Check rule:

$$\frac{C, \Gamma \vdash M^\circ \rightsquigarrow C_1, M \Rightarrow T_1 \qquad C_2 = C_1 \cup T_1 \leq T}{C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow}$$

- $T_1 \leq T$ computes a constraints set such that T_1 is a subtype of T:

$$\text{stream}^s\ T \leq \text{stream}^r\ U \ = \ \{r \sqsubseteq s\} \cup T \leq U$$

$$\vdots$$

Size-inference algorithm

- The algorithm is given by two judgments (inputs, outputs):
  - $C, \Gamma \vdash M^\circ \rightsquigarrow C', M \Rightarrow T$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
  - $C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C', M$:

    for any $\rho \models C'$, $\rho\Gamma \vdash \rho M : \rho T$, *where* $|M| \equiv M^\circ$ *and* $C \subseteq C'$
- Check rule:

$$\frac{C, \Gamma \vdash M^\circ \rightsquigarrow C_1, M \Rightarrow T_1 \qquad C_2 = C_1 \cup T_1 \leq T}{C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C_2, M}$$

- $T_1 \leq T$ computes a constraints set such that T_1 is a subtype of T:

$$\text{stream}^s\ T \leq \text{stream}^r\ U \ = \ \{r \sqsubseteq s\} \cup T \leq U$$

$$\vdots$$

Size-inference algorithm

- Rule for cons (simplified):

$$C, \Gamma \vdash M_1{}^{\circ} \rightsquigarrow C_1, M_1 \Rightarrow T_1$$

$$\frac{}{C, \Gamma \vdash \mathsf{cons}(M_1{}^{\circ}, M_2{}^{\circ}) \rightsquigarrow}$$

Size-inference algorithm

- Rule for cons (simplified):

$$C, \Gamma \vdash M_1{}^{\circ} \rightsquigarrow C_1, M_1 \Rightarrow T_1$$

$$\frac{}{C, \Gamma \vdash \mathsf{cons}(M_1{}^{\circ}, M_2{}^{\circ}) \rightsquigarrow}$$

Size-inference algorithm

- Rule for cons (simplified):

$$\frac{C, \Gamma \vdash M_1{}^{\circ} \rightsquigarrow C_1, M_1 \Rightarrow T_1 \quad C, \Gamma \vdash M_2{}^{\circ} \rightsquigarrow C_2, M_2 \Rightarrow \mathsf{stream}^r\ T_2}{C, \Gamma \vdash \mathsf{cons}(M_1{}^{\circ}, M_2{}^{\circ}) \rightsquigarrow}$$

Size-inference algorithm

- Rule for cons (simplified):

$$\frac{\begin{array}{c} C, \Gamma \vdash M_1{}^{\circ} \rightsquigarrow C_1, M_1 \Rightarrow T_1 \\ C, \Gamma \vdash M_2{}^{\circ} \rightsquigarrow C_2, M_2 \Rightarrow \mathsf{stream}^r\ T_2 \end{array}}{C, \Gamma \vdash \mathsf{cons}(M_1{}^{\circ}, M_2{}^{\circ}) \rightsquigarrow}$$

Size-inference algorithm

- Rule for cons (simplified):

$$\frac{\begin{array}{c} C, \Gamma \vdash M_1{}^{\circ} \rightsquigarrow C_1, M_1 \Rightarrow T_1 \\ C, \Gamma \vdash M_2{}^{\circ} \rightsquigarrow C_2, M_2 \Rightarrow \mathsf{stream}^{r}\ T_2 \\ T_1 \sqcup T_2 \rightsquigarrow C_3, T \end{array}}{C, \Gamma \vdash \mathsf{cons}(M_1{}^{\circ}, M_2{}^{\circ}) \rightsquigarrow}$$

- where $T_1 \sqcup T_2 \rightsquigarrow C, T$ computes the *least upper bound* of T_1 and T_2: for any $\rho \models C$, $\rho T_1, \rho T_2 \leq \rho T$

$$\mathsf{stream}^{s}\ T_1 \sqcup \mathsf{stream}^{r}\ T_2 \ = \ C \cup \{\alpha \sqsubseteq s, \alpha \sqsubseteq r\}, \mathsf{stream}^{\alpha}\ T \quad \text{where } T_1 \sqcup T_2 = C, T$$

$$\vdots$$

Size-inference algorithm

- Rule for cons (simplified):

$$\frac{\begin{array}{c} C, \Gamma \vdash M_1{}^{\circ} \rightsquigarrow C_1, M_1 \Rightarrow T_1 \\ C, \Gamma \vdash M_2{}^{\circ} \rightsquigarrow C_2, M_2 \Rightarrow \mathsf{stream}^{r}\ T_2 \\ T_1 \sqcup T_2 \rightsquigarrow C_3, T \end{array}}{C, \Gamma \vdash \mathsf{cons}(M_1{}^{\circ}, M_2{}^{\circ}) \rightsquigarrow}$$

- where $T_1 \sqcup T_2 \rightsquigarrow C, T$ computes the *least upper bound* of T_1 and T_2: for any $\rho \models C$, $\rho T_1, \rho T_2 \leq \rho T$

$$\mathsf{stream}^{s}\ T_1 \sqcup \mathsf{stream}^{r}\ T_2 \ = \ C \cup \{\alpha \sqsubseteq s, \alpha \sqsubseteq r\}, \mathsf{stream}^{\alpha}\ T \quad \text{where } T_1 \sqcup T_2 = C, T$$

$$\vdots$$

Size-inference algorithm

- Rule for cons (simplified):

$$\frac{\begin{array}{c} C, \Gamma \vdash M_1{}^{\circ} \rightsquigarrow C_1, M_1 \Rightarrow T_1 \\ C, \Gamma \vdash M_2{}^{\circ} \rightsquigarrow C_2, M_2 \Rightarrow \mathsf{stream}^{r}\ T_2 \\ T_1 \sqcup T_2 \rightsquigarrow C_3, T \end{array}}{C, \Gamma \vdash \mathsf{cons}(M_1{}^{\circ}, M_2{}^{\circ}) \rightsquigarrow C_3, \mathsf{cons}(M_1, M_2) \Rightarrow \mathsf{stream}^{\widehat{r}}\ T}$$

- where $T_1 \sqcup T_2 \rightsquigarrow C, T$ computes the *least upper bound* of T_1 and T_2: for any $\rho \models C$, $\rho T_1, \rho T_2 \leq \rho T$

$$\mathsf{stream}^{s}\ T_1 \sqcup \mathsf{stream}^{r}\ T_2 \;=\; C \cup \{\alpha \sqsubseteq s, \alpha \sqsubseteq r\}, \mathsf{stream}^{\alpha}\ T \quad \text{where } T_1 \sqcup T_2 = C, T$$

$$\vdots$$

Size-inference algorithm

- Rule for cofix (simplified):

$$\frac{}{C, \Gamma \vdash \text{cofix } f : T^{\star} := M^{\circ} \rightsquigarrow}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star$$

$$\frac{}{C, \Gamma \vdash \mathsf{cofix}\ f : T^\star := M^\circ \rightsquigarrow}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star$$

$$\frac{}{C, \Gamma \vdash \mathsf{cofix}\ f : T^\star := M^\circ \rightsquigarrow}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star$$
$$C, \Gamma \vdash T^\star \rightsquigarrow C_1, \Pi\Delta.\mathsf{stream}^\alpha\ U \Rightarrow^* \mathsf{Type}$$

$$\frac{}{C, \Gamma \vdash \mathsf{cofix}\ f : T^\star := M^\circ \rightsquigarrow}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star$$
$$C, \Gamma \vdash T^\star \rightsquigarrow C_1, \Pi\Delta.\mathsf{stream}^\alpha\ U \Rightarrow^* \mathsf{Type}$$

$$C, \Gamma \vdash \mathsf{cofix}\ f : T^\star := M^\circ \rightsquigarrow$$

Size-inference algorithm

- Rule for cofix (simplified):

$$\frac{\begin{array}{c} T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star \\ C, \Gamma \vdash T^\star \rightsquigarrow C_1, \Pi\Delta.\mathsf{stream}^\alpha\ U \Rightarrow^* \mathsf{Type} \\ C_1, \Gamma(f : \Pi\Delta.\mathsf{stream}^\alpha\ U) \vdash M^\circ \Leftarrow \Pi\widehat{\Delta}.\mathsf{stream}^{\widehat{\alpha}}\ U \rightsquigarrow C_2, M \end{array}}{C, \Gamma \vdash \mathsf{cofix}\ f : T^\star := M^\circ \rightsquigarrow}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$\frac{\begin{array}{c} T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star \\ C, \Gamma \vdash T^\star \rightsquigarrow C_1, \Pi\Delta.\mathsf{stream}^\alpha\ U \Rightarrow^* \mathsf{Type} \\ C_1, \Gamma(f : \Pi\Delta.\mathsf{stream}^\alpha\ U) \vdash M^\circ \Leftarrow \Pi\widehat{\Delta}.\mathsf{stream}^{\widehat{\alpha}}\ U \rightsquigarrow C_2, M \end{array}}{C, \Gamma \vdash \mathsf{cofix}\ f : T^\star := M^\circ \rightsquigarrow}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$\frac{\begin{array}{c} T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star \\ C, \Gamma \vdash T^\star \rightsquigarrow C_1, \Pi\Delta.\mathsf{stream}^\alpha\ U \Rightarrow^* \mathsf{Type} \\ C_1, \Gamma(f : \Pi\Delta.\mathsf{stream}^\alpha\ U) \vdash M^\circ \Leftarrow \Pi\widehat{\Delta}.\mathsf{stream}^{\widehat{\alpha}}\ U \rightsquigarrow C_2, M \\ \mathsf{RecCheck}(\alpha, V^\star, C_2 \cup \widehat{\Delta} \leq \Delta) = C_3 \end{array}}{C, \Gamma \vdash \mathsf{cofix}\ f{:}T^\star := M^\circ \rightsquigarrow}$$

- RecCheck (Barthe et al., TLCA 2005) checks that the side conditions in the typing rules of cofixpoints are satisfied. On success, it returns a new set of constraints.

$$\frac{\begin{array}{c} T \equiv \Pi\Delta.\mathsf{stream}^\imath\ U \qquad \imath\ \mathsf{neg}\ \Delta \qquad \imath \notin \Gamma, U, M \\ \Gamma \vdash T : \mathsf{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath] \end{array}}{\Gamma \vdash (\mathsf{cofix}\ f : |T| := M) : T[s/\imath]}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$\frac{\begin{array}{c} T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star \\ C, \Gamma \vdash T^\star \rightsquigarrow C_1, \Pi\Delta.\mathsf{stream}^\alpha\ U \Rightarrow^* \mathsf{Type} \\ C_1, \Gamma(f : \Pi\Delta.\mathsf{stream}^\alpha\ U) \vdash M^\circ \Leftarrow \Pi\widehat{\Delta}.\mathsf{stream}^{\widehat{\alpha}}\ U \rightsquigarrow C_2, M \\ \mathsf{RecCheck}(\alpha, V^\star, C_2 \cup \widehat{\Delta} \leq \Delta) = C_3 \end{array}}{C, \Gamma \vdash \mathsf{cofix}\ f{:}T^\star := M^\circ \rightsquigarrow}$$

- RecCheck (Barthe et al., TLCA 2005) checks that the side conditions in the typing rules of cofixpoints are satisfied. On success, it returns a new set of constraints.

$$\frac{\begin{array}{c} T \equiv \Pi\Delta.\mathsf{stream}^\imath\ U \qquad \imath\ \mathsf{neg}\ \Delta \qquad \imath \notin \Gamma, U, M \\ \Gamma \vdash T : \mathsf{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath] \end{array}}{\Gamma \vdash (\mathsf{cofix}\ f : |T| := M) : T[s/\imath]}$$

Size-inference algorithm

- Rule for cofix (simplified):

α must be assigned to a fresh variable $\imath$

$$\frac{C, \Gamma \vdash \quad , \quad : \mathsf{Type} \qquad C_1, \Gamma(f : \Pi\Delta.\mathsf{stream}^{\alpha}\, U) \vdash M^{\circ} \Leftarrow \Pi\widehat{\Delta}.\mathsf{stream}^{\widehat{\alpha}}\, U \rightsquigarrow C_2, M \qquad \mathsf{RecCheck}(\alpha, V^{\star}, C_2 \cup \widehat{\Delta} \leq \Delta) = C_3}{C, \Gamma \vdash \mathsf{cofix}\ f{:}T^{\star} := M^{\circ} \rightsquigarrow}$$

- RecCheck (Barthe et al., TLCA 2005) checks that the side conditions in the typing rules of cofixpoints are satisfied. On success, it returns a new set of constraints.

$$\frac{T \equiv \Pi\Delta.\mathsf{stream}^{\imath}\, U \qquad \imath\ \mathsf{neg}\ \Delta \qquad \imath \notin \Gamma, U, M \qquad \Gamma \vdash T : \mathsf{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath]}{\Gamma \vdash (\mathsf{cofix}\ f : |T| := M) : T[s/\imath]}$$

Size-inference algorithm

- Rule for cofix (simplified):

Variables must be assigned to a stage containing $\imath$

$$\frac{C, \Gamma \quad C_1, \Gamma(f : \Pi\Delta.\mathsf{stream}^{\alpha}\, U) \vdash M^{\circ} \Leftarrow \Pi\Delta.\mathsf{stream}^{\widehat{\alpha}}\, U \rightsquigarrow C_2, M \quad \mathsf{RecCheck}(\alpha, V^{\star}, C_2 \cup \widehat{\Delta} \leq \Delta) = C_3}{C, \Gamma \vdash \mathsf{cofix}\ f{:}T^{\star} := M^{\circ} \rightsquigarrow}$$

- RecCheck (Barthe et al., TLCA 2005) checks that the side conditions in the typing rules of cofixpoints are satisfied. On success, it returns a new set of constraints.

$$\frac{T \equiv \Pi\Delta.\mathsf{stream}^{\imath}\, U \qquad \imath\ \mathsf{neg}\ \Delta \qquad \imath \notin \Gamma, U, M \qquad \Gamma \vdash T : \mathsf{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath]}{\Gamma \vdash (\mathsf{cofix}\ f : |T| := M) : T[s/\imath]}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$\frac{\begin{array}{c} T^\star \equiv \Pi\Delta.\ldots \\ C, \Gamma \vdash T^\star \rightsquigarrow C_1, \Pi\Delta.\text{stream}^{\ldots} U \to \ldots \to \text{Type} \\ C_1, \Gamma(f : \Pi\Delta.\text{stream}^{\alpha} U) \vdash M^\circ \Leftarrow \Pi\widehat{\Delta}.\text{stream}^{\widehat{\alpha}} U \rightsquigarrow C_2, M \\ \text{RecCheck}(\alpha, V^\star, C_2 \cup \widehat{\Delta} \leq \Delta) = C_3 \end{array}}{C, \Gamma \vdash \text{cofix } f{:}T^\star := M^\circ \rightsquigarrow}$$

Ensures that $\imath$ neg Δ

- RecCheck (Barthe et al., TLCA 2005) checks that the side conditions in the typing rules of cofixpoints are satisfied. On success, it returns a new set of constraints.

$$\frac{\begin{array}{c} T \equiv \Pi\Delta.\text{stream}^{\imath} U \qquad \imath \text{ neg } \Delta \qquad \imath \notin \Gamma, U, M \\ \Gamma \vdash T : \text{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath] \end{array}}{\Gamma \vdash (\text{cofix } f : |T| := M) : T[s/\imath]}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$\frac{\begin{array}{c} T^\star \\ C, \Gamma \vdash T^\star \rightsquigarrow C_1, \quad \text{ype} \\ C_1, \Gamma(f : \Pi\Delta.\mathsf{stream}^\alpha\ U) \vdash M^\circ \lessdot \Pi\Delta.\mathsf{stream}^{\widehat{\alpha}}\ U \rightsquigarrow C_2, M \\ \mathsf{RecCheck}(\alpha, V^\star, C_2 \cup \widehat{\Delta} \leq \Delta) = C_3 \end{array}}{C, \Gamma \vdash \mathsf{cofix}\ f{:}T^\star := M^\circ \rightsquigarrow}$$

No other variable is assigned to $\imath$

- RecCheck (Barthe et al., TLCA 2005) checks that the side conditions in the typing rules of cofixpoints are satisfied. On success, it returns a new set of constraints.

$$\frac{\begin{array}{c} T \equiv \Pi\Delta.\mathsf{stream}^{\imath}\ U \qquad \imath\ \mathsf{neg}\ \Delta \qquad \imath \notin \Gamma, U, M \\ \Gamma \vdash T : \mathsf{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath] \end{array}}{\Gamma \vdash (\mathsf{cofix}\ f : |T| := M) : T[s/\imath]}$$

Size-inference algorithm

- Rule for cofix (simplified):

$$\frac{\begin{array}{c} T^\star \equiv \Pi\Delta^\star.\mathsf{stream}^\star\ U^\star \\ C, \Gamma \vdash T^\star \rightsquigarrow C_1, \Pi\Delta.\mathsf{stream}^\alpha\ U \Rightarrow^* \mathsf{Type} \\ C_1, \Gamma(f : \Pi\Delta.\mathsf{stream}^\alpha\ U) \vdash M^\circ \Leftarrow \Pi\widehat{\Delta}.\mathsf{stream}^{\widehat{\alpha}}\ U \rightsquigarrow C_2, M \\ \mathsf{RecCheck}(\alpha, V^\star, C_2 \cup \widehat{\Delta} \leq \Delta) = C_3 \end{array}}{C, \Gamma \vdash \mathsf{cofix}\ f{:}T^\star := M^\circ \rightsquigarrow C_3, \mathsf{cofix}\ f{:}T^\star := M \Rightarrow \mathsf{stream}^\alpha\ U}$$

- RecCheck (Barthe et al., TLCA 2005) checks that the side conditions in the typing rules of cofixpoints are satisfied. On success, it returns a new set of constraints.

$$\frac{\begin{array}{c} T \equiv \Pi\Delta.\mathsf{stream}^\imath\ U \qquad \imath\ \mathsf{neg}\ \Delta \qquad \imath \notin \Gamma, U, M \\ \Gamma \vdash T : \mathsf{Type} \qquad \Gamma(f : T) \vdash M : T[\widehat{\imath}/\imath] \end{array}}{\Gamma \vdash (\mathsf{cofix}\ f : |T| := M) : T[s/\imath]}$$

Conclusions

- We developed an extension of CC with streams (a subset of Coq) with a type-based productivity checker for stream definitions.
  - More expressive than Coq's syntactic methods.
  - Easier to understand and implement.
  - Treats inductive and coinductive types uniformly.
- We showed SN and logical consistency based on a Λ-set model.
  - Can be easily extended to cover general coinductive types.
- We developed a size-inferring algorithm.
  - Type-based productivity poses little burden for the user.

Future work

- Complete the extension of the model to all coinductive types.
- Type system extensions. E.g. ML-style polymorphism: $\forall \imath.\, T(\imath)$.
- Mid-term goal: reimplement termination and productivity checking in Coq using sized types.

Future work

- Complete the extension of the model to all coinductive types.
- Type system extensions. E.g. ML-style polymorphism: $\forall \imath.\, T(\imath)$.
- Mid-term goal: reimplement termination and productivity checking in Coq using sized types.

Thank you!