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.

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Induction</td>
<td>Coinduction</td>
</tr>
<tr>
<td>Least fixed point</td>
<td>Greatest fixed point</td>
</tr>
<tr>
<td>Recursive functions</td>
<td>Corecursive functions</td>
</tr>
<tr>
<td>consume data</td>
<td>produce data</td>
</tr>
</tbody>
</table>
Coinductive Types in Coq

- Streams:

  \[
  \text{ColInductive} \; \text{stream} \; A := \text{cons} : A \rightarrow \text{stream} \; A \rightarrow \text{stream} \; A
  \]
Coinductive Types in Coq

CoInductive stream A := cons : A → stream A → stream A
Coinductive Types in Coq

- Streams:

  **CoInductive** \( \text{stream} A := \text{cons} : A \to \text{stream} A \to \text{stream} A \)

- Corecursive functions produce streams:

  \[
  \text{zeroes} \overset{\text{def}}{=} \text{cofix} \ Z := \text{cons}(0, Z)
  \]

  zeroes produce the stream:

  \[
  \text{cons}(0, \text{cons}(0, \text{cons}(0, \ldots)))
  \]
Coinductive types

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

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

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

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is productivity.
## Coinductive types

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

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

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

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

- In proof assistants, termination of recursive functions is essential to ensure logical consistency and decidability of type checking.
- For corecursive functions, the dual condition to termination is **productivity**.
- In the case of streams, productivity means that we can compute any element of the stream in finite time:
  
  $$\text{cofix } Z_1 := \text{cons}(0, Z_1)$$

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

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

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

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

  \[
  \text{nats} \overset{\text{def}}{=} \text{cofix nats} := \lambda n. \text{cons}(n, \text{nats} (1 + n))
  \]
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.,
  
  \[ \text{every corecursive call is performed directly under a constructor} \]

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

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

  ![✓]
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:

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

\[
\text{nats} \overset{\text{def}}{=} \lambda n. \text{cofix nats} := \text{cons}(n, \text{map} (1+) \text{nats})
\]
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:

  \[
  \text{nats} \overset{\text{def}}{=} \text{cofix nats} := \lambda n. \text{cons}(n, \text{nats} (1 + n)) \checkmark
  \]
  \[
  \text{nats} \overset{\text{def}}{=} \lambda n. \text{cofix nats} := \text{cons}(n, \text{map} (1+) \text{nats}) \times
  \]
## Syntactic Methods for Termination and Productivity

<table>
<thead>
<tr>
<th>Inductive types</th>
<th>Coinductive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Termination</td>
<td>Productivity</td>
</tr>
</tbody>
</table>
### Syntactic Methods for Termination and Productivity

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

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

- Difficult to understand.
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

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

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 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.
Types annotated with size information

\[ \text{stream}^s A \]

is the type of streams (of type \( A \)) where \textit{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:

\[ s ::= \nu \mid \hat{s} \mid \infty \]
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:
  \[ s ::= \nu \mid \hat{s} \mid \infty \]
- Substage relation (reflexive and transitive):
  \[ s \sqsubseteq \infty \quad s \sqsubseteq \hat{s} \]
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:
  
  \[ s ::= i \mid \hat{s} \mid \infty \]

- Substage relation (reflexive and transitive):
  
  \[ s \sqsubseteq \infty \quad s \sqsubseteq \hat{s} \]

  but stage variables are incomparable; i.e. \(i \nsubseteq j\)
The substage relation induces a subtype relation:

\[ r \sqsubseteq s \quad T \leq U \]

\[ \text{stream}^s T \leq \text{stream}^r U \]
Type-Based Productivity

- The substage relation induces a subtype relation:

\[ r \sqsubseteq s \quad T \leq U \quad \Rightarrow \quad \text{stream}^s T \leq \text{stream}^r U \]

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

\[ U_1 \leq T_1 \quad T_2 \leq U_2 \quad \Rightarrow \quad \Pi x : T_1. T_2 \leq \Pi x : U_1. U_2 \]
Type-Based Productivity

- Typing rules for constructors:

\[ \vdash N : A \quad \vdash M : \text{stream}^s A \]

\[ \vdash \text{cons}(N, M) : \text{stream}^\sim A \]
Type-Based Productivity

- Typing rules for constructors:

\[
\begin{align*}
\vdash N : A & \quad \vdash M : \text{stream}^s A \\
\hline
\vdash \text{cons}(N, M) : \text{stream}^{\hat{s}} A
\end{align*}
\]

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

\[
\begin{align*}
(f : \text{stream} \; A) \vdash M : \text{stream} \; A \\
\hline
\vdash (\text{cofix } f := M) : \text{stream} \; A
\end{align*}
\]
Type-Based Productivity

- Typing rules for constructors:

\[
\begin{align*}
\Gamma &\vdash N : A & \Gamma &\vdash M : \text{stream}^s A \\
\hline
\Gamma &\vdash \text{cons}(N, M) : \text{stream}^{\hat{s}} A
\end{align*}
\]

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

\[
\begin{align*}
(\ell : \text{stream}^{\ell} A) &\vdash M : \text{stream}^{\hat{\ell}} A \\
\hline
\Gamma &\vdash (\text{cofix } f := M) : \text{stream}^s A & \ell &\text{ fresh}
\end{align*}
\]
Type-Based Productivity

- Typing rules for constructors:

\[ \vdash N : A \quad \vdash M : \text{stream}^s A \]
\[ \vdash \text{cons}(N, M) : \text{stream}^\sim A \]

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

\[ (f : \text{stream}^\iota A) \vdash M : \text{stream}^\iota A \]
\[ \vdash (\text{cofix } f := M) : \text{stream}^s A \] \[ \iota \text{ fresh} \]

- The (almost) full rule:

\[ T \equiv \Pi \Delta. \text{stream}^\iota U \quad \iota \text{ neg } \Delta \quad \iota \notin \Gamma, U, M \]
\[ \Gamma \vdash T : \text{Type} \quad \Gamma(f : T) \vdash M : T[^\iota/\iota] \]
\[ \Gamma \vdash (\text{cofix } f : |T| := M) : T[^s/\iota] \]
Type-Based Productivity

- Typing rules for constructors:

\[ \vdash N : A \quad \vdash M : \text{stream}^s A \]

\[ \vdash \text{cons}(N, M) : \text{stream}^\hat{s} A \]

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

\[ (f : \text{stream}^\iota A) \vdash M : \text{stream}^{\hat{\iota}} A \]

\[ \vdash (\text{cofix } f := M) : \text{stream}^s A \quad \iota \text{ fresh} \]

- The (almost) full rule:

\[ T \equiv \Pi \Delta \text{.stream}^\iota U \quad \iota \text{ neg } \Delta \quad \iota \notin \Gamma, U, M \]

\[ \Gamma \vdash T : \text{Type} \quad \Gamma(f : T) \vdash M : T^{\hat{\iota}/\iota} \]

\[ \Gamma \vdash (\text{cofix } f : |T| := M) : T[s/\iota] \]
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

\[ T ::= \text{Type}_i \mid \text{Prop} \mid \Pi x : T. T \]
\[ \mid \lambda x : T^\circ. T \mid TT \]
\[ \mid \text{stream}^s T \mid \text{cons}(T, T) \mid \]
\[ \mid \text{case}T T \text{ of cons}(x, y) \Rightarrow T \]
\[ \mid \text{cofix } f : T^* ::= T \]

- 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:

\[
\text{cofix } f := M \rightarrow M[\text{cofix } f := M / f]
\]
CC with Type-Based Productivity

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

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

- Unfolding is restricted to occur inside case analysis:

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

  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:

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

- Unfolding is restricted to occur inside case analysis:

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

  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:

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

- Unfolding is restricted to occur inside case analysis:

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

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:

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

- Unfolding is restricted to occur inside case analysis:

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

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

\[
\text{map} : (A \rightarrow B) \rightarrow \text{stream}^s A \rightarrow \text{stream}^s B
\]

\[
\text{map} \overset{\text{def}}{=} \lambda f. \text{cofix } M : \text{stream}^* A \rightarrow \text{stream}^* B := \\
\lambda x. \text{case } x \text{ of } (h, t) \Rightarrow \text{cons}(f h, M t)
\]
Type-Based Productivity

Examples

map : \((A \rightarrow B) \rightarrow \text{stream}^s A \rightarrow \text{stream}^s B\)  

\[
\text{map} \overset{\text{def}}{=} \lambda f. \text{cofix } M : \text{stream}^* A \rightarrow \text{stream}^* B \\
\lambda x. \text{case } x \text{ of } (h, t) \Rightarrow \text{cons}(f h, M t)
\]
Type-Based Productivity

Examples

\[
\text{map : } (A \to B) \to \text{stream}^\infty A \to \text{stream}^s B
\]

\[
\text{map} \overset{\text{def}}{=} \lambda f. \text{cofix } M : \text{stream } A \to \text{stream}^* B := \\
\lambda x. \text{case } x \text{ of } (h, t) \Rightarrow \text{cons}(f h, M t)
\]
Type-Based Productivity

Examples

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

\[
\text{merge} : \text{stream}^s \text{nat} \to \text{stream}^s \text{nat} \to \text{stream}^s \text{nat}
\]
map : \((A \rightarrow B) \rightarrow \text{stream}^s A \rightarrow \text{stream}^s B\)

merge : \text{stream}^s \text{nat} \rightarrow \text{stream}^s \text{nat} \rightarrow \text{stream}^s \text{nat}

\[
\text{merge} \stackrel{\text{def}}{=} \text{cofix}\ M : \text{stream}^* \text{nat} \rightarrow \text{stream}^* \text{nat} \rightarrow \text{stream}^* \text{nat} := \\
\lambda x_1 \ x_2. \ \text{case } x_1 \text{ of cons}(h_1, t_1) \Rightarrow \\
\text{case } x_2 \text{ of cons}(h_2, t_2) \Rightarrow \\
\quad \text{if } h_1 \leq h_2 \text{ then cons}(h_1, M \ t_1 \ x_2) \\
\quad \text{else cons}(h_2, M \ x_1 \ t_2) 
\]
Type-Based Productivity

Examples

\[ \text{map} : (A \rightarrow B) \rightarrow \text{stream}^s A \rightarrow \text{stream}^s B \]

\[ \text{merge} : \text{stream}^s \text{nat} \rightarrow \text{stream}^s \text{nat} \rightarrow \text{stream}^s \text{nat} \]

\[ \text{merge} \overset{\text{def}}{=} \text{cofix } M : \text{stream}^* \text{nat} \rightarrow \text{stream}^* \text{nat} \rightarrow \text{stream}^* \text{nat} := \]

\[ \lambda x_1 x_2. \text{case } x_1 \text{ of cons}(h_1, t_1) \Rightarrow \]

\[ \text{case } x_2 \text{ of cons}(h_2, t_2) \Rightarrow \]

\[ \text{if } h_1 \leq h_2 \text{ then cons}(h_1, M t_1 x_2) \]

\[ \text{else cons}(h_2, M x_1 t_2) \]

Guarded
Type-Based Productivity

Examples

\[
\begin{align*}
\text{map} &: (A \rightarrow B) \rightarrow \text{stream}^S A \rightarrow \text{stream}^S B \\
\text{merge} &: \text{stream}^S \text{nat} \rightarrow \text{stream}^S \text{nat} \rightarrow \text{stream}^S \text{nat} \\
\text{hamming} &: \text{stream}^S \text{nat}
\end{align*}
\]
Type-Based Productivity

Examples

\[
\begin{align*}
\text{map} : (A \to B) &\to \text{stream}^s A \to \text{stream}^s B \\
\text{merge} : \text{stream}^s \text{nat} &\to \text{stream}^s \text{nat} \to \text{stream}^s \text{nat} \\
\text{hamming} : \text{stream}^s \text{nat} \\
\text{hamming} \overset{\text{def}}{=} \text{cofix } H : \text{stream}^* \text{nat} := \\
&\quad \text{cons}(1, \text{merge} (\text{map} (2\times) H) \\
&\quad \quad (\text{merge} (\text{map} (3\times) H) \\
&\quad \quad \quad (\text{map} (5\times) H))))
\end{align*}
\]
Type-Based Productivity

Examples

map : \((A \rightarrow B) \rightarrow \text{stream}^s A \rightarrow \text{stream}^s B\)

merge : \text{stream}^s \text{nat} \rightarrow \text{stream}^s \text{nat} \rightarrow \text{stream}^s \text{nat}

hamming : \text{stream}^s \text{nat}

hamming \overset{\text{def}}{=} \text{cofix } H : \text{stream}^* \text{nat} :=

\begin{align*}
\text{cons}(1, \text{merge} (\text{map} (2\times) H)) \\
(\text{merge} (\text{map} (3\times) H) \\
(\text{map} (5\times) H)))
\end{align*}
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 \( \Lambda \)-sets (Altenkirch).
- Relatively easy to understand and expressive
  - Impredicativity (not treated in this work for space reasons).
  - Inductive and coinductive types.
- Types are interpreted as \( \Lambda \)-sets:

\[
(X, \models)
\]

- \( X \) is a set
- \( \models \subseteq \text{SN} \times X \) is a realizability relation
Strong Normalization

- Model based on $\Lambda$-sets (Altenkirch).
- Relatively easy to understand and expressive
  - Impredicativity (not treated in this work for space reasons).
  - Inductive and coinductive types.
- Types are interpreted as $\Lambda$-sets:
  \[
  (X, \models)
  \]
  - $X$ is a set
  - $\models \subseteq \text{SN} \times X$ is a realizability relation

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

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:

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

::
Strong Normalization

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

\[ \lambda x : T . M \gamma = \alpha \in [T] \mapsto [M]_{\gamma, \alpha} \]

\[ [M \cdot N]_\gamma = [M]_\gamma [N]_\gamma \]

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

\[ \vdots \]

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

- \(\Lambda\)-set interpretation ignores sizes.
- Set-theoretical interpretation:

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

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

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

- $\Lambda$-set interpretation ignores sizes.

- Set-theoretical interpretation:

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

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

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

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

\[
[\text{cons}(M, N)]_\gamma = \langle [M]_\gamma, [N]_\gamma \rangle
\]

\[
[\text{cofix } f : T := M]_\gamma = \epsilon(F, P(F))
\]

where \( P(F \in [T]_\gamma) \) is \( F = [M]_{\gamma,F} \)
Strong Normalization

- \( \Lambda \)-set interpretation ignores sizes.
- Realizability interpretation:

\[
M \models [\prod x : T. u] \ f \iff N \models [T] \ \alpha \Rightarrow M \ N \models [u]_{(\alpha)} \ f(\alpha)
\]

\[
M \models [\text{stream } T] \ (\alpha_0, \alpha_1, \ldots) \iff \downarrow M \to^* \ \text{cons}(N_1, N_2)
\]

\[
\land \ N_1 \models [T] \ \alpha_0
\]

\[
\land \ N_2 \models [\text{stream } T] \ (\alpha_1, \ldots)
\]
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):

  \[ \llbracket \text{stream}^s T \rrbracket = \{ (\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in \llbracket T \rrbracket \} \]

- 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):

\[
\llbracket \text{stream}^s T \rrbracket = \{ (\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in \llbracket T \rrbracket \}
\]

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

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

\[
\frac{(f : \text{stream}^i T) \vdash M : \text{stream}^\hat{i} T}{\vdash \text{cofix } f := M : \text{stream}^s T}
\]
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):

\[
\llbracket \text{stream}^s T \rrbracket = \{ (\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in \llbracket T \rrbracket \}
\]

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

\[
(f : \text{stream}^i T) \vdash M : \text{stream}^{\hat{i}} T \\
\vdash \text{cofix } f := M : \text{stream}^s T
\]

\((\alpha_1, \alpha_2)\)
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):

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

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

Cofixpoint rule defines a contractive function in the relational model:

\[
(f : \text{stream}^r T) \vdash M : \text{stream}^\wedge T \\
\vdash \text{cofix } f ::= M : \text{stream}^s T
\]

Coincide up to \( \nu \)

\((\alpha_1, \alpha_2)\)
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):

\[
\begin{align*}
\llbracket \text{stream}^s T \rrbracket &= \{ (\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in \llbracket T \rrbracket \} \\
\end{align*}
\]

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

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

\[
\begin{align*}
(f : \text{stream}^i T) &\vdash M : \text{stream}^\hat{i} T \\
&\vdash \text{cofix } f := M : \text{stream}^s T \\
(\alpha_1, \alpha_2) &\sim \sim ([M](\alpha_1), [M](\alpha_2))
\end{align*}
\]
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):

  \[ \left[ \text{stream}^s T \right] = \{(\alpha, \beta) : i < [s] \Rightarrow (\alpha_i, \beta_i) \in \left[ T \right] \} \]

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

  \[
  \frac{(f : \text{stream}^i T) \vdash M : \text{stream}^\hat{i} T}{\vdash \text{cofix } f := M : \text{stream}^s T}
  \]

  Coincide up to \( i + 1 \)

  \[
  (\alpha_1, \alpha_2) \rightsquigarrow ([M](\alpha_1), [M](\alpha_2))
  \]
Strong Normalization

- \([M]\) is a contractive function

\[
\frac{(f : \text{stream}^i T) \vdash M : \text{stream}^\hat{i} T}{\vdash \text{cofix } f := M : \text{stream}^s T}
\]

- Basis for interpreting cofix

\[
\alpha = \alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \alpha_4 \ldots \\
\beta = \beta_1 \quad \beta_2 \quad \beta_3 \quad \beta_4 \ldots
\]
Strong Normalization

- $[M]$ is a contractive function

$$
(f : \text{stream}^i T) \vdash M : \text{stream}^\hat{i} T
\quad \vdash \text{cofix } f := M : \text{stream}^s T
$$

- Basis for interpreting $\text{cofix}$

\[
\alpha = \alpha_1 \alpha_2 \alpha_3 \alpha_4 \ldots
\]
\[
\beta = \beta_1 \beta_2 \beta_3 \beta_4 \ldots
\]

\[
[M](\alpha) = \alpha_1 \alpha_2 \alpha_3 \alpha_4 \ldots
\]
\[
[M](\beta) = \alpha_{11} \beta_2 \beta_3 \beta_4 \ldots
\]

Same first element
Strong Normalization

- $[M]$ is a contractive function

\[
\frac{(f : \text{stream}^\iota T) \vdash M : \text{stream}^\iota T}{\vdash \text{cofix } f := M : \text{stream}^s T}
\]

- Basis for interpreting cofix

\[
\begin{align*}
\alpha &= \alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \alpha_4 \ldots \\
\beta &= \beta_1 \quad \beta_2 \quad \beta_3 \quad \beta_4 \ldots
\end{align*}
\]

\[
\begin{align*}
[M]^2(\alpha) &= \alpha_{21} \quad \alpha_{22} \quad \alpha_{23} \quad \alpha_{24} \ldots \\
[M]^2(\beta) &= \alpha_{21} \quad \alpha_{22} \quad \beta_{23} \quad \beta_{24} \ldots
\end{align*}
\]
Strong Normalization

- \([M]\) is a contractive function

\[
\frac{(f : \text{stream}^i T) \vdash M : \text{stream}^\hat{i} T}{\vdash \text{cofix} \ f := M : \text{stream}^s T}
\]

- Basis for interpreting \(\text{cofix}\)

\[
\begin{align*}
\alpha &= \alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \alpha_4 \ldots \\
\beta &= \beta_1 \quad \beta_2 \quad \beta_3 \quad \beta_4 \ldots
\end{align*}
\]

\[
[M]^3 \begin{cases} 
[M]^3(\alpha) &= \alpha_{31} \quad \alpha_{32} \quad \alpha_{33} \quad \alpha_{34} \ldots \\
[M]^3(\beta) &= \alpha_31 \quad \alpha_32 \quad \alpha_33 \quad \beta_{34} \ldots
\end{cases}
\]
Strong Normalization

- $[M]$ is a contractive function

$$
(f : \text{stream}^i T) \vdash M : \text{stream}^\hat{i} T
$$

$$
\vdash \text{cofix} \ f := M : \text{stream}^s T
$$

- Basis for interpreting cofix

$$
\begin{align*}
\alpha & = \alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \alpha_4 \quad \ldots \\
\beta & = \beta_1 \quad \beta_2 \quad \beta_3 \quad \beta_4 \quad \ldots
\end{align*}
$$

$$
[M]_\infty
$$

$$
[M]_\infty(\alpha) = \alpha_\infty_1 \quad \alpha_\infty_2 \quad \alpha_\infty_3 \quad \alpha_\infty_4 \quad \ldots
$$

Unique stream
Strong Normalization

- $[M]$ is a contractive function

\[
(f : \text{stream}^\triangledown T) \vdash M : \text{stream}^\triangledown T
\]

\[\vdash \text{cofix } f := M : \text{stream}^s T\]

- Basis for interpreting cofix

\[
\begin{align*}
\alpha &= \alpha_1 & \alpha_2 & \alpha_3 & \alpha_4 & \ldots \\
\beta &= \beta_1 & \beta_2 & \beta_3 & \beta_4 & \ldots
\end{align*}
\]

\[
[M]^{\infty}
\]

\[
[M]^{\infty}(\alpha) = \alpha_{\infty_1} & \alpha_{\infty_2} & \alpha_{\infty_3} & \alpha_{\infty_4} & \ldots
\]

- 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 $(\gamma_1, \gamma_2) \in \llbracket \Gamma \rrbracket$, then $[M]_{\gamma_1, \gamma_2} \in \llbracket T \rrbracket_{\gamma_1, \gamma_2}$.

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

- As a corollary, we can prove logical consistency:

  \[ There \ is \ no \ term \ M \ such \ that \ \vdash M : (\prod X : Type_0. X). \]
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

\[
(f : \text{stream } A \rightarrow \text{stream } A) \\
(y : \text{stream } A), (x : A) \quad \vdash \quad f(f(\text{cons } (x, y)))
\]
Size-inference algorithm

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

\[
(f : \text{stream } A \rightarrow \text{stream } A) \\
(y : \text{stream } A), (x : A) \\
\vdash f(f(\text{cons } (x, y)))
\]

\[
(f : \text{stream}^\alpha A \rightarrow \text{stream}^\beta A) \\
(y : \text{stream}^\gamma A), (x : A) \\
\vdash f(f(\text{cons } (x, y)))
\]
Size-inference algorithm

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

Example: given

\[
(f : \text{stream } A \rightarrow \text{stream } A) \quad \vdash \quad f(f(\text{cons } (x, y)))
\]

\[
(y : \text{stream } A), (x : A)
\]

\[
\left\{ \begin{array}{c}
\text{cons}(x, y) : \text{stream } \tilde{\gamma} A \\
\end{array} \right.
\]

\[
(f : \text{stream}^\alpha A \rightarrow \text{stream}^\beta A) \\
(y : \text{stream} \gamma A), (x : A)
\]

\[
\vdash \quad f(f(\text{cons } (x, y)))
\]
Size-inference algorithm

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

Example: given

\[(f : \text{stream } A \rightarrow \text{stream } A) \quad (y : \text{stream } A), (x : A) \quad \vdash f(f(\text{cons } (x, y)))\]

\[\exists \quad \text{cons}(x, y) : \text{stream } \hat{\gamma} A\]

\[(f : \text{stream}^\alpha A \rightarrow \text{stream}^\beta A) \quad (y : \text{stream}^\gamma A), (x : A) \quad \vdash f(f(\text{cons } (x, y)))\]

\[\alpha \sqsubseteq \hat{\gamma}, \alpha \sqsubseteq \beta\]
Size-inference algorithm

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

\[
(f : \text{stream } A \rightarrow \text{stream } A) \quad \quad (y : \text{stream } A), (x : A) \quad \quad \vdash f(f(\text{cons } (x, y)))
\]

\[
\Downarrow
\]

\[
(f : \text{stream}^\alpha A \rightarrow \text{stream}^\beta A) \quad \quad (y : \text{stream}^\gamma A), (x : A) \quad \quad \vdash f(f(\text{cons } (x, y)))
\]

\[
\alpha \sqsubseteq \gamma, \alpha \sqsubseteq \beta
\]

Any stage substitution \( \rho \) 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

\[
(f : \text{stream } A \rightarrow \text{stream } A) \\
(y : \text{stream } A), (x : A) \\
\vdash f(f(\text{cons } (x, y)))
\]

\[
(f : \text{stream}^\alpha A \rightarrow \text{stream}^\beta A) \\
(y : \text{stream}^\gamma A), (x : A) \\
\vdash f(f(\text{cons } (x, y)))
\]

\[
\alpha \sqsubseteq \hat{\gamma}, \alpha \sqsubseteq \beta
\]

Any stage substitution \( \rho \) satisfying these constraints gives a valid typing judgment. The substitution \( \rho(\alpha) = \infty, \forall \alpha \), satisfies any constraint set.
The algorithm is given by two judgments (inputs, outputs):

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

2. $C, \Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C', M$:
   
   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:

  \[
  \begin{align*}
  C, \Gamma & \vdash M^\circ \Leftarrow T \rightsquigarrow \\
  \end{align*}
  \]
Size-inference algorithm

The algorithm is given by two judgments (inputs, outputs):

1. \( C, \Gamma \vdash M^\circ \leadsto C', M \Rightarrow T \):
   
   For any \( \rho \models C' \), \( \rho \Gamma \vdash \rho M : \rho T \), where \( |M| \equiv M^\circ \) and \( C \subseteq C' \)

2. \( C, \Gamma \vdash M^\circ \Leftarrow T \leadsto C', M \):
   
   For any \( \rho \models C' \), \( \rho \Gamma \vdash \rho M : \rho T \), where \( |M| \equiv M^\circ \) and \( C \subseteq C' \)

Check rule:

\[
C, \Gamma \vdash M^\circ \leadsto C_1, M \Rightarrow T_1
\]

\[
\therefore C, \Gamma \vdash M^\circ \Leftarrow T \leadsto
\]
Size-inference algorithm

- The algorithm is given by two judgments (inputs, outputs):
  - $\Gamma \vdash M^\circ \rightsquigarrow C', M \Rightarrow T$:
    
    for any $\rho \models C', \rho \Gamma \vdash \rho M : \rho T$, where $|M| \equiv M^\circ$ and $C \subseteq C'$
  - $\Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow C', M$:
    
    for any $\rho \models C', \rho \Gamma \vdash \rho M : \rho T$, where $|M| \equiv M^\circ$ and $C \subseteq C'$

- Check rule:
  
  $\Gamma \vdash M^\circ \rightsquigarrow C_1, M \Rightarrow T_1$

  $\Gamma \vdash M^\circ \Leftarrow T \rightsquigarrow$
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:
  \[
  \begin{align*}
  &C, \Gamma \vdash M^\circ \rightsquigarrow C_1, M \Rightarrow T_1 \\
  &C_2 = C_1 \cup T_1 \leq T \\
  \hline
  &C, \Gamma \vdash M^\circ \leftarrow T \rightsquigarrow
  \end{align*}
  \]

- \(T_1 \leq T\) 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:

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

- $T_1 \leq T$ 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):

\[ C, \Gamma \vdash M_1^\circ \leadsto C_1, M_1 \Rightarrow T_1 \]

\[ C, \Gamma \vdash \text{cons}(M_1^\circ, M_2^\circ) \leadsto \]
Size-inference algorithm

- Rule for cons (simplified):

\[ C, \Gamma \vdash M_1^\circ \leadsto C_1, M_1 \Rightarrow T_1 \]

\[ C, \Gamma \vdash \text{cons}(M_1^\circ, M_2^\circ) \leadsto \]
Size-inference algorithm

- Rule for cons (simplified):

\[
\begin{align*}
C, \Gamma \vdash M_1^\circ & \leadsto C_1, M_1 \Rightarrow T_1 \\
C, \Gamma \vdash M_2^\circ & \leadsto C_2, M_2 \Rightarrow \text{stream}' T_2
\end{align*}
\]

\[
C, \Gamma \vdash \text{cons}(M_1^\circ, M_2^\circ) \leadsto
\]
Size-inference algorithm

- Rule for cons (simplified):

\[
\begin{align*}
C, \Gamma &\vdash M_1^\circ \rightsquigarrow C_1, M_1 \Rightarrow T_1 \\
C, \Gamma &\vdash M_2^\circ \rightsquigarrow C_2, M_2 \Rightarrow \text{stream}^r T_2
\end{align*}
\]

\[
C, \Gamma \vdash \text{cons}(M_1^\circ, M_2^\circ) \rightsquigarrow
\]
Size-inference algorithm

- Rule for cons (simplified):

\[
\begin{align*}
    C, \Gamma \vdash M_1^\circ & \rightsquigarrow C_1, M_1 \Rightarrow T_1 \\
    C, \Gamma \vdash M_2^\circ & \rightsquigarrow C_2, M_2 \Rightarrow \text{stream}^r T_2 \\
    T_1 \sqcup T_2 & \rightsquigarrow C_3, T
\end{align*}
\]

\[
C, \Gamma \vdash \text{cons}(M_1^\circ, M_2^\circ) \rightsquigarrow
\]

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

\[
\text{stream}^s T_1 \sqcup \text{stream}^r T_2 = C \cup \{ \alpha \sqsubseteq s, \alpha \sqsubseteq r \}, \text{stream}^\alpha T
\]

where \( T_1 \sqcup T_2 = C, T \)

\[
\vdots
\]
Size-inference algorithm

- Rule for cons (simplified):

\[
\begin{align*}
C, \Gamma & \vdash M_1^\circ \rightsquigarrow C_1, M_1 \Rightarrow T_1 \\
C, \Gamma & \vdash M_2^\circ \rightsquigarrow C_2, M_2 \Rightarrow \text{stream}^r T_2 \\
T_1 \sqcup T_2 & \rightsquigarrow C_3, T \\
\hline
C, \Gamma & \vdash \text{cons}(M_1^\circ, M_2^\circ) \rightsquigarrow
\end{align*}
\]

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

\[
\text{stream}^s T_1 \sqcup \text{stream}^r T_2 = C \cup \{ \alpha \sqsubseteq s, \alpha \sqsubseteq r \}, \text{stream}^\alpha T
\]

where \( T_1 \sqcup T_2 = C, T \)

\[
\vdots
\]
Size-inference algorithm

- Rule for \text{cons} (simplified):

\[
\begin{align*}
C, \Gamma \vdash M_1^\circ & \rightsquigarrow C_1, M_1 \Rightarrow T_1 \\
C, \Gamma \vdash M_2^\circ & \rightsquigarrow C_2, M_2 \Rightarrow \text{stream}^r T_2 \\
T_1 \sqcup T_2 & \rightsquigarrow C_3, T
\end{align*}
\]

\[
C, \Gamma \vdash \text{cons}(M_1^\circ, M_2^\circ) \rightsquigarrow C_3, \text{cons}(M_1, M_2) \Rightarrow \text{stream}^\hat{r} T
\]

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

\[
\text{stream}^s T_1 \sqcup \text{stream}^r T_2 = C \cup \{ \alpha \sqsubseteq s, \alpha \sqsubseteq r \}, \text{stream}^\alpha T
\]

where \( T_1 \sqcup T_2 = C, T \)

\[\vdots\]
Size-inference algorithm

- Rule for cofix (simplified):

\[ C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow \]
Size-inference algorithm

- Rule for cofix (simplified):

\[ T^* \equiv \Pi \Delta^* . \text{stream}^* U^* \]

\[ C, \Gamma \vdash \text{cofix } f : T^* \coloneqq M^\circ \rightsquigarrow \]
Size-inference algorithm

- Rule for cofix (simplified):

\[ T^* \equiv \Pi \Delta^*. \text{stream}^* \ U^* \]

\[ C, \Gamma \vdash \text{cofix } f : T^* := \mathcal{M}^\circ \rightsquigarrow \]
Size-inference algorithm

- Rule for cofix (simplified):

\[ T^* \equiv \Pi \Delta^*.\text{stream}^* \ U^* \]

\[ C, \Gamma \vdash T^* \rightsquigarrow C_1, \Pi \Delta.\text{stream}^\alpha U \Rightarrow^* \text{Type} \]

\[
C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow
\]
Size-inference algorithm

- Rule for `cofix` (simplified):

\[
T^* \equiv \Pi \Delta^*.\text{stream}^* \ U^*
\]
\[
C, \Gamma \vdash T^* \leadsto C_1, \Pi \Delta.\text{stream}^\alpha U \Rightarrow^* \text{Type}
\]

\[
C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \leadsto
\]
Rule for cofix (simplified):

\[
T^* \equiv \Pi \Delta^* . \text{stream}^* U^* \\
C, \Gamma \vdash T^* \leadsto C_1, \Pi \Delta . \text{stream}^\alpha U \Rightarrow^* \text{Type} \\
C_1, \Gamma (f : \Pi \Delta . \text{stream}^\alpha U) \vdash M^\circ \iff \Pi \hat{\Delta} . \text{stream}^\alpha U \leadsto C_2, M
\]

\[
C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \leadsto
\]
Size-inference algorithm

- Rule for cofix (simplified):

\[
T^* \equiv \Pi \Delta^*.\text{stream}^*\ U^*
\]

\[
C, \Gamma \vdash T^* \rightsquigarrow C_1, \Pi \Delta.\text{stream}^\alpha U \Rightarrow^* \text{Type}
\]

\[
C_1, \Gamma(f : \Pi \Delta.\text{stream}^\alpha U) \vdash M^\circ \iff \Pi \hat{\Delta}.\text{stream}^{\hat{\alpha}} U \rightsquigarrow C_2, M
\]

\[
C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow
\]
Size-inference algorithm

- Rule for cofix (simplified):

\[
T^* \equiv \Pi \Delta^*.\text{stream}^* U^*
\]

\[
C, \Gamma \vdash T^* \rightsquigarrow C_1, \Pi \Delta.\text{stream}^\alpha U \Rightarrow^* \text{Type}
\]

\[
C_1, \Gamma(f : \Pi \Delta.\text{stream}^\alpha U) \vdash M^\circ \iff \Pi \hat{\Delta}.\text{stream}^{\hat{\alpha}} U \rightsquigarrow C_2, M
\]

RecCheck(\alpha, V^*, C_2 \cup \hat{\Delta} \leq \Delta) = C_3

\[
C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow
\]

- 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.

\[
T \equiv \Pi \Delta.\text{stream}^i U \quad i \text{ neg } \Delta \quad i \notin \Gamma, U, M
\]

\[
\Gamma \vdash T : \text{Type} \quad \Gamma(f : T) \vdash M : T[\hat{i}/i]
\]

\[
\Gamma \vdash (\text{cofix } f : \mid T \mid := M) : T[s/i]
\]
Size-inference algorithm

- Rule for cofix (simplified):

\[
T^* \equiv \Pi \Delta^*.\text{stream}^* U^*
\]

\[
C, \Gamma \vdash T^* \rightsquigarrow C_1, \Pi \Delta.\text{stream}^\alpha U \Rightarrow^* \text{Type}
\]

\[
C_1, \Gamma(f : \Pi \Delta.\text{stream}^\alpha U) \vdash M^\circ \iff \Pi \hat{\Delta}.\text{stream}^\alpha U \rightsquigarrow C_2, M
\]

\[
\text{RecCheck}(\alpha, V^*, C_2 \cup \hat{\Delta} \leq \Delta) = C_3
\]

\[
C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow
\]

- 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.

\[
T \equiv \Pi \Delta.\text{stream}^i U \quad i \text{ neg } \Delta \quad i \notin \Gamma, U, M
\]

\[
\Gamma \vdash T : \text{Type} \quad \Gamma(f : T) \vdash M : T[\hat{i}/i]
\]

\[
\Gamma \vdash (\text{cofix } f : |T| := M) : T[s/i]
\]
Size-inference algorithm

- Rule for cofix (simplified):

\[ T \equiv \Pi \Delta.\text{stream}^\alpha U \quad \alpha \text{ must be assigned to a fresh variable } i \]

\[
\begin{align*}
C, \Gamma(F : \Pi \Delta.\text{stream}^\alpha U) - M^\circ \iff \Pi \hat{\Delta}.\text{stream}^\hat{\alpha} U \rightsquigarrow C_2, M \\
\text{RecCheck}(\alpha, V^*, C_2 \cup \hat{\Delta} \leq \Delta) = C_3 \\
C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow
\end{align*}
\]

- 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.

\[
T \equiv \Pi \Delta.\text{stream}^i U \quad i \text{ neg } \Delta \quad i \notin \Gamma, U, M \\
\Gamma \vdash T : \text{Type} \quad \Gamma(F : T) \vdash M : T[\hat{i}/i] \\
\Gamma \vdash (\text{cofix } f : |T| : = M) : T[s/i]
\]
Size-inference algorithm

- Rule for \texttt{cofix} (simplified):

\[
\begin{align*}
C, \Gamma \vdash & \text{cofix } f : T^* := M^\circ \rightsquigarrow \\
& \text{RecCheck(} \alpha, V^*, C_2 \cup \hat{\Delta} \leq \Delta) = C_3
\end{align*}
\]

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

\[
T \equiv \Pi \Delta.\text{stream}^\alpha U \quad \neg \Delta \ni \Delta \ni \not\in \Gamma, U, M \\
\Gamma \vdash T : \text{Type} \quad \Gamma(f : T) \vdash M : T[\hat{\alpha}/\alpha] \\
\Gamma \vdash (\text{cofix } f : |T| := M) : T[s/\alpha]
\]
Size-inference algorithm

- Rule for cofix (simplified):

\[ T^* \equiv \Pi \Delta^* \]

\[ C, \Gamma \vdash T^* \rightsquigarrow C_1, \Pi \Delta.\text{stream} \alpha U \Rightarrow \text{Type} \]

\[ C_1, \Gamma(f : \Pi \Delta.\text{stream}^\alpha U) \vdash M^\circ \iff \Pi \Delta.\text{stream} \ulcorner \alpha \urcorner U \rightsquigarrow C_2, M \]

RecCheck(\alpha, V^*, C_2 \cup \hat{\Delta} \leq \Delta) = C_3

\[ C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow \]

- 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.

\[ T \equiv \Pi \Delta.\text{stream}^\iota U \quad \iota \text{ neg } \Delta \quad \iota \notin \Gamma, U, M \]

\[ \Gamma \vdash T : \text{Type} \quad \Gamma(f : T) \vdash M : T[\hat{\iota} / \iota] \]

\[ \Gamma \vdash (\text{cofix } f : |T| := M) : T[s / \iota] \]
Size-inference algorithm

- Rule for cofix (simplified):

\[
\begin{align*}
T^* & \\ 
C, \Gamma \vdash T^* \rightsquigarrow C_1, \forall \ii \vdash T^* : Type \\
C_1, \Gamma(f : \Pi \Delta. \text{stream}^{\alpha} U) \vdash M^\circ \leq \Pi \Delta. \text{stream}^{\hat{\alpha}} U \rightsquigarrow C_2, M \\
\text{RecCheck}(\alpha, V^*, C_2 \cup \hat{\Delta} \leq \Delta) = C_3
\end{align*}
\]

\[
C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow
\]

- 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.

\[
\begin{align*}
T \equiv \Pi \Delta. \text{stream}^{\iota} U & \quad \iota \text{ neg } \Delta \quad \iota \notin \Gamma, U, M \\
\Gamma \vdash T : \text{Type} & \quad \Gamma(f : T) \vdash M : T[\hat{\iota}/\iota] \\
\Gamma \vdash \text{cofix } f : |T| := M : T[s/\iota]
\end{align*}
\]
Size-inference algorithm

- Rule for cofix (simplified):

\[ T^* \equiv \Pi \Delta^*.\text{stream}^* U^* \]
\[ C, \Gamma \vdash T^* \rightsquigarrow C_1, \Pi \Delta.\text{stream}^\alpha U \Rightarrow^* \text{Type} \]
\[ C_1, \Gamma(f : \Pi \Delta.\text{stream}^\alpha U) \vdash M^\circ \iff \Pi \hat{\Delta}.\text{stream}^\alpha U \rightsquigarrow C_2, M \]
\[ \text{RecCheck}(\alpha, V^*, C_2 \cup \hat{\Delta} \leq \Delta) = C_3 \]
\[ C, \Gamma \vdash \text{cofix } f : T^* := M^\circ \rightsquigarrow C_3, \text{cofix } f : T^* := M \Rightarrow \text{stream}^\alpha U \]

- 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.

\[ T \equiv \Pi \Delta.\text{stream}^i U \quad \neg \Delta \quad \forall \Gamma, U, M \]
\[ \Gamma \vdash T : \text{Type} \quad \Gamma(f : T) \vdash M : T[\hat{i}/i] \]
\[ \Gamma \vdash (\text{cofix } f : |T| := M) : T[s/i] \]
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: $\forall \tau. T(\tau)$.
- 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: $\forall \nu. T(\nu)$.
- Mid-term goal: reimplement termination and productivity checking in Coq using sized types.

Thank you!