A Functional Synchronous Language with Time Warps

Adrien Guatto

Inria Paris

Gallium - 06/11/2017

Streams in Programs and Proofs

- Infinite sequences of values

$$\mathsf{Stream}(X) \approx \mathbb{N} \to X$$

- Kahn's insight: a deterministic reactive system can be described as a mathematical function

$$\mathsf{Stream}(X) \to \mathsf{Stream}(Y)$$

- Exploited in various languages and formalisms:
  - lazy functional languages, e.g. Haskell;
  - synchronous dataflow languages, e.g. Lustre;
  - proof assistants based on Type Theory, e.g. Coq.

Recursive Stream Definitions

- Streams, as infinite objects, have to be introduced via self-referential definitions.
- For example, `zeroes` can be characterized as the solution of

$$\texttt{zeroes} = 0 :: \texttt{zeroes}$$

  and defined as such in Haskell, Lustre, and Coq.
- What about equations with several or no solutions?

$$\texttt{weird} = \texttt{weird}$$

  Different languages follow different approaches.

Recursive Stream Definitions

- Streams, as infinite objects, have to be introduced via self-referential definitions.
- For example, `zeroes` can be characterized as the solution of

$$\texttt{zeroes} = 0 :: \texttt{zeroes}$$

  and defined as such in Haskell, Lustre, and Coq.
- What about equations with several or no solutions?

$$\texttt{weird} = \texttt{weird}$$

  Different languages follow different approaches.

Demonstration 1

Try the above in our three prototypical languages.

Productivity and its Enforcement

Productivity

A stream definition is *productive* when any finite prefix of the stream can be computed in finite time.

Productivity can be enforced by:

- Syntactic criteria (e.g., Coq and Lustre)
  - ✓ Simple and well-understood
  - ✗ Anti-modular, inexpressive
- Type systems (e.g., guarded type theories, Lucid Synchrone)
  - ✓ Modular
  - ? Expressive

Productivity and its Enforcement

Productivity

A stream definition is *productive* when any finite prefix of the stream can be computed in finite time.

Productivity can be enforced by:

- Syntactic criteria (e.g., Coq and Lustre)
  - ✓ Simple and well-understood
  - ✗ Anti-modular, inexpressive
- Type systems (e.g., guarded type theories, Lucid Synchrone)
  - ✓ Modular
  - ? Expressive

Guarded Type Theories and the Later Modality

Nakano's Key Idea

In a recursive definition, self-references are only available later.

Guarded Type Theories and the Later Modality

Nakano's Key Idea

In a recursive definition, self-references are only available later.

Guarded Type Theories and the Later Modality

Nakano's Key Idea

In a recursive definition, self-references are only available later.

Formally:

Guarded Type Theories and the Later Modality

Nakano's Key Idea

In a recursive definition, self-references are only available later.

Formally:

- Enrich the type language with a modality

$$\tau ::= \cdots \mid \blacktriangleright \tau$$

  and related operations.

Guarded Type Theories and the Later Modality

Nakano's Key Idea

In a recursive definition, self-references are only available later.

Formally:

- Enrich the type language with a modality

$$\tau ::= \cdots \mid \blacktriangleright \tau$$

  and related operations.

- Give appropriate types to the stream constructor/destructors;

$$(::) : X \to \blacktriangleright \mathsf{Stream}(X) \to \mathsf{Stream}(X)$$

$$\texttt{head} : \mathsf{Stream}(X) \to X \qquad \texttt{tail} : \mathsf{Stream}(X) \to \blacktriangleright \mathsf{Stream}(X)$$

Guarded Type Theories and the Later Modality

Nakano's Key Idea

In a recursive definition, self-references are only available later.

Formally:

- Enrich the type language with a modality

$$\tau ::= \cdots \mid \blacktriangleright \tau$$

  and related operations.

- Give appropriate types to the stream constructor/destructors;

$$(::) : X \to \blacktriangleright \mathsf{Stream}(X) \to \mathsf{Stream}(X)$$

$$\mathtt{head} : \mathsf{Stream}(X) \to X \qquad \mathtt{tail} : \mathsf{Stream}(X) \to \blacktriangleright \mathsf{Stream}(X)$$

- Have a special typing rule for recursive definitions.

Guarded Recursive Definitions

$$\frac{\Gamma, x : \blacktriangleright \tau \vdash t : \tau}{\Gamma \vdash \texttt{rec}\ (x : \tau).t : \tau}$$

Guarded Recursive Definitions

$$\frac{\Gamma, x : \blacktriangleright \tau \vdash t : \tau}{\Gamma \vdash \texttt{rec}\ (x : \tau).t : \tau}$$

- The following definition is well-typed.

$$\texttt{zeroes} = 0 :: \texttt{zeroes}$$

Guarded Recursive Definitions

$$\frac{\Gamma, x : \blacktriangleright \tau \vdash t : \tau}{\Gamma \vdash \texttt{rec}\ (x : \tau).t : \tau}$$

- The following definition is well-typed.

$$\texttt{rec}\ (\texttt{zeroes} : \mathsf{Stream}(\mathsf{Int})).(0 :: \texttt{zeroes})$$

Guarded Recursive Definitions

$$\frac{\Gamma, x : \blacktriangleright \tau \vdash t : \tau}{\Gamma \vdash \texttt{rec}\ (x : \tau).t : \tau}$$

- The following definition is well-typed.

$$\texttt{rec}\ (\texttt{zeroes} : \mathsf{Stream}(\mathsf{Int})).(0 :: \texttt{zeroes})$$

- This one is not:

$$\texttt{rec}\ (\texttt{weird} : \mathsf{Stream}(\mathsf{Int})).\texttt{weird}$$

Guarded Recursive Definitions

$$\frac{\Gamma, x : \blacktriangleright \tau \vdash t : \tau}{\Gamma \vdash \texttt{rec}\ (x : \tau).t : \tau}$$

- The following definition is well-typed.

$$\texttt{rec}\ (\texttt{zeroes} : \mathsf{Stream}(\mathsf{Int})).(0 :: \texttt{zeroes})$$

- This one is not:

$$\texttt{rec}\ (\texttt{weird} : \mathsf{Stream}(\mathsf{Int})).\texttt{weird}$$

"This expression has type $\blacktriangleright$ Stream(Int) but was expected to have type Stream(Int)."

Later and its Limitations

- Following Nakano, many works from Birkedal, Krishnaswami, McBride, Møgelberg, Bizjak and others, studying:
  - powerful (dependent) type systems;
  - denotational and operational semantics;
  - practical and theoretical use cases, from

$$\texttt{nat} = 0 :: \texttt{map}\ (\lambda \texttt{x}.\texttt{x} + 1)\ \texttt{nat}$$

    to step-indexed models of programming languages.

Later and its Limitations

- Following Nakano, many works from Birkedal, Krishnaswami, McBride, Møgelberg, Bizjak and others, studying:
  - powerful (dependent) type systems;
  - denotational and operational semantics;
  - practical and theoretical use cases, from

    $$\texttt{nat} = 0 :: \texttt{map}\ (\lambda\texttt{x}.\texttt{x} + 1)\ \texttt{nat}$$

    to step-indexed models of programming languages.
- However, current guarded type theories struggle with...
  - mutual recursion:

    $$\texttt{nat} = 0 :: \texttt{spos} \qquad \texttt{spos} = \texttt{map}\ (\lambda\texttt{x}.\texttt{x} + 1)\ \texttt{nat}$$

Later and its Limitations

- Following Nakano, many works from Birkedal, Krishnaswami, McBride, Møgelberg, Bizjak and others, studying:
  - powerful (dependent) type systems;
  - denotational and operational semantics;
  - practical and theoretical use cases, from

    $$\texttt{nat} = 0 :: \texttt{map}\ (\lambda \mathrm{x}.\mathrm{x} + 1)\ \texttt{nat}$$

    to step-indexed models of programming languages.
- However, current guarded type theories struggle with...
  - mutual recursion:

    $$\texttt{nat} = 0 :: \texttt{spos} \qquad \texttt{spos} = \texttt{map}\ (\lambda \mathrm{x}.\mathrm{x} + 1)\ \texttt{nat}$$

  - fine-grained dependencies:

    $$\begin{aligned} \texttt{thuemorse} = \ & \texttt{false} :: \texttt{tail}\ (\texttt{h}\ \texttt{thuemorse}) \\ & \texttt{where}\ \texttt{h}\ (\texttt{x} :: \texttt{xs}) = \texttt{x} :: (\texttt{not}\ \texttt{x}) :: \texttt{h}\ \texttt{xs} \end{aligned}$$

Beyond Later: Time Warps

- Models of guarded recursion interpret types by ω-indexed families of sets of observations.

$$(\mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^n$$

- The later modality applies a simple transformation to a type: delaying what can be observed one step into the future.

$$(\blacktriangleright \mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^{n-1}$$

Beyond Later: Time Warps

- Models of guarded recursion interpret types by ω-indexed families of sets of observations.

$$(\mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^n$$

- The later modality applies a simple transformation to a type: delaying what can be observed one step into the future.

$$(\blacktriangleright \mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^{n-1}$$

Main Claims of This Talk

Beyond Later: Time Warps

- Models of guarded recursion interpret types by ω-indexed families of sets of observations.

$$(\mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^n$$

- The later modality applies a simple transformation to a type: delaying what can be observed one step into the future.

$$(\blacktriangleright \mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^{n-1}$$

Main Claims of This Talk

- By considering a large class of transformations, *time warps*, we obtain a very general parametric modality.

Beyond Later: Time Warps

- Models of guarded recursion interpret types by ω-indexed families of sets of observations.

$$(\mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^n$$

- The later modality applies a simple transformation to a type: delaying what can be observed one step into the future.

$$(\blacktriangleright \mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^{n-1}$$

Main Claims of This Talk

- By considering a large class of transformations, *time warps*, we obtain a very general parametric modality.
- It is possible to design a type system around this modality to make it both *usable* and *implementable*.

Beyond Later: Time Warps

- Models of guarded recursion interpret types by ω-indexed families of sets of observations.

$$(\mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^n$$

- The later modality applies a simple transformation to a type: delaying what can be observed one step into the future.

$$(\blacktriangleright \mathsf{Stream}(\mathsf{Int}))_n \approx \mathsf{Int}^{n-1}$$

Main Claims of This Talk

- By considering a large class of transformations, *time warps*, we obtain a very general parametric modality.
- It is possible to design a type system around this modality to make it both *usable* and *implementable*.

Pulsar is a prototype implementation of these ideas.

Outline

1 Introduction

2 Programming in a Language with Time Warps

3 Metatheoretical Aspects

4 Algorithmic Type Checking

5 Perspectives

Outline

1 Introduction

2 Programming in a Language with Time Warps

3 Metatheoretical Aspects

4 Algorithmic Type Checking

5 Perspectives

Overview

- **Pulsar** is based on the simply-typed λ-calculus extended with a built-in stream type.

$$\begin{aligned} \tau &::= \ \nu \mid \mathsf{Stream}(\tau) \mid \tau \to \tau \mid \tau \times \tau \mid \dots \\ \nu &::= \ \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Char} \end{aligned}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \mathtt{fun}\,(x : \tau_1).t : \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}$$

Overview

- **Pulsar** is based on the simply-typed λ-calculus extended with a built-in stream type.

$$\begin{aligned} \tau &::= \ \nu \mid \mathsf{Stream}(\tau) \mid \tau \to \tau \mid \tau \times \tau \mid \ldots \\ \nu &::= \ \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Char} \end{aligned}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \mathtt{fun}\,(x : \tau_1).t : \tau_2} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\ t_2 : \tau_2}$$

- To the above, it adds the warp modality

$$\tau ::= \cdots \mid {\color{red} *_p\, \tau}$$

plus guarded recursion, subtyping, and a new construct.

Time Warps

- Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \qquad\qquad f(\omega) = \bigsqcup_{i<\omega} f(i)$$

Time Warps

- Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \qquad\qquad f(\omega) = \bigsqcup_{i<\omega} f(i)$$

- We restrict ourselves to warps defined as running sums $\mathcal{O}\, p$ of ultimately periodic number sequences p.

$$(\mathcal{O}\, p)(i) = \sum_{j=0}^{j<i} p[j] \text{ for } 0 < i < \omega$$

For example:

Time Warps

- Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \qquad\qquad f(\omega) = \bigsqcup_{i<\omega} f(i)$$

- We restrict ourselves to warps defined as running sums $\mathcal{O}\,p$ of ultimately periodic number sequences p.

$$(\mathcal{O}\,p)(i) = \sum_{j=0}^{j<i} p[j] \text{ for } 0 < i < \omega$$

For example:

$$(\mathcal{O}\,(0))(i) = 0$$

Time Warps

- Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \qquad\qquad f(\omega) = \bigsqcup_{i<\omega} f(i)$$

- We restrict ourselves to warps defined as running sums $\mathcal{O}\, p$ of ultimately periodic number sequences p.

$$(\mathcal{O}\, p)(i) = \sum_{j=0}^{j<i} p[j] \text{ for } 0 < i < \omega$$

For example:

$$(\mathcal{O}\,(0))(i) = 0 \qquad (\mathcal{O}\,(1))(i) = i$$

Time Warps

- Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \qquad\qquad f(\omega) = \bigsqcup_{i<\omega} f(i)$$

- We restrict ourselves to warps defined as running sums $\mathcal{O}\, p$ of ultimately periodic number sequences p.

$$(\mathcal{O}\, p)(i) = \sum_{j=0}^{j<i} p[j] \text{ for } 0 < i < \omega$$

For example:

$$(\mathcal{O}\,(0))(i) = 0 \qquad (\mathcal{O}\,(1))(i) = i \qquad (\mathcal{O}\,0\,(1))(i) = i - 1$$

Time Warps

- Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \qquad\qquad f(\omega) = \bigsqcup_{i<\omega} f(i)$$

- We restrict ourselves to warps defined as running sums $\mathcal{O}\, p$ of ultimately periodic number sequences p.

$$(\mathcal{O}\, p)(i) = \sum_{j=0}^{j<i} p[j] \text{ for } 0 < i < \omega$$

For example:

$$(\mathcal{O}\,(0))(i) = 0 \qquad (\mathcal{O}\,(1))(i) = i \qquad (\mathcal{O}\,0\,(1))(i) = i - 1$$

$$(\mathcal{O}\,(2))(i) = 2i$$

Time Warps

- Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \qquad\qquad f(\omega) = \bigsqcup_{i<\omega} f(i)$$

- We restrict ourselves to warps defined as running sums $\mathcal{O}\, p$ of ultimately periodic number sequences p.

$$(\mathcal{O}\, p)(i) = \sum_{j=0}^{j<i} p[j] \text{ for } 0 < i < \omega$$

For example:

$$(\mathcal{O}\,(0))(i) = 0 \qquad (\mathcal{O}\,(1))(i) = i \qquad (\mathcal{O}\,0\,(1))(i) = i - 1$$

$$(\mathcal{O}\,(2))(i) = 2i \qquad (\mathcal{O}\,(\omega))(i) = \omega \text{ for } i > 0$$

Guarded Recursion in **Pulsar**

Demonstration 2

Let us try to write `zeroes`.

Guarded Recursion in **Pulsar**

Demonstration 2

Let us try to write `zeroes`.

Guarded recursion is formulated with $\blacktriangleright \tau \triangleq *_{0\,(1)}\,\tau$ as expected.

$$\frac{\Gamma, x : *_{0\,(1)}\,\tau \vdash e : \tau}{\Gamma \vdash \texttt{rec}\ (x : \tau).e : \tau}$$

Similarly, primitives have types

$$(::) : \tau \to *_{0\,(1)}\,\mathsf{Stream}(\tau) \to \mathsf{Stream}(\tau)$$

$$\texttt{head} : \mathsf{Stream}(\tau) \to \tau \qquad \texttt{tail} : \mathsf{Stream}(\tau) \to *_{0\,(1)}\,\mathsf{Stream}(\tau)$$

Warp Composition

Demonstration 3

Let us try to write `weird`.

Warp Composition

Demonstration 3

Let us try to write `weird`.

As expected, there is no τ such that $\vdash \texttt{weird} : \mathrm{Stream}(\tau)$ holds.
However, $\vdash \texttt{weird} : \ast_{(0)} \mathrm{Stream}(\tau)$ holds for any τ. Why?

Warp Composition

Demonstration 3

Let us try to write `weird`.

As expected, there is no τ such that $\vdash \texttt{weird} : \mathsf{Stream}(\tau)$ holds. However, $\vdash \texttt{weird} : \ast_{(0)}\,\mathsf{Stream}(\tau)$ holds for any τ. Why?

$$\frac{}{\ast_{p \ast q}\,\tau \equiv \ast_p \ast_q \tau}$$

The operator $\ast$, called *warp composition*, is characterized by

$$\mathcal{O}\,(p \ast q) = \mathcal{O}\,q \circ \mathcal{O}\,p$$

hence we have

$$\ast_{0\,(1)} \ast_{(0)} \mathsf{Stream}(\tau) \equiv \ast_{0\,(1) \ast (0)} \mathsf{Stream}(\tau) \equiv \ast_{(0)} \mathsf{Stream}(\tau)$$

Warping and Delays

Demonstration 3

Let us try to write `map`.

Warping and Delays

Demonstration 3

Let us try to write `map`.

$$\frac{\Gamma \vdash e : \tau}{*_p \Gamma \vdash e \; \texttt{by} \; p : *_p \tau}$$

Warping and Delays

Demonstration 3

Let us try to write `map`.

$$\frac{\Gamma \vdash e : \tau}{*_p \Gamma \vdash e \ \mathtt{by}\ p : *_p \tau}$$

$$\begin{aligned}
\mathtt{map} &: \ *_{0\,(1)}\,((\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Stream}(\mathsf{Int}) \to \mathsf{Stream}(\mathsf{Int})) \\
\mathtt{f} &: \ \mathsf{Int} \to \mathsf{Int} \\
\mathtt{xs} &: \ *_{0\,(1)}\,\mathsf{Stream}(\mathsf{Int})
\end{aligned}$$

Warping and Delays

Demonstration 3

Let us try to write map.

$$\frac{\Gamma \vdash e : \tau}{*_p \, \Gamma \vdash e \; \texttt{by} \; p : *_p \, \tau} \qquad \frac{}{\tau \equiv *_{(1)} \, \tau}$$

$$\begin{aligned}
\texttt{map} &: \; *_{0\,(1)} \, ((\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Stream}(\mathsf{Int}) \to \mathsf{Stream}(\mathsf{Int})) \\
\texttt{f} &: \; \mathsf{Int} \to \mathsf{Int} \equiv *_{(1)} \, (\mathsf{Int} \to \mathsf{Int}) \\
\texttt{xs} &: \; *_{0\,(1)} \, \mathsf{Stream}(\mathsf{Int})
\end{aligned}$$

Warping and Delays

Demonstration 3

Let us try to write `map`.

$$\frac{\Gamma \vdash e : \tau}{*_p \Gamma \vdash e \texttt{ by } p : *_p \tau} \qquad \frac{}{\tau \equiv *_{(1)} \tau} \qquad \frac{q \leq p}{*_p \tau <: *_q \tau}$$

$$\begin{aligned}
\texttt{map} :&\quad *_{0\,(1)} ((\mathsf{Int} \to \mathsf{Int}) \to \mathsf{Stream}(\mathsf{Int}) \to \mathsf{Stream}(\mathsf{Int})) \\
\texttt{f} :&\quad \mathsf{Int} \to \mathsf{Int} \equiv *_{(1)} (\mathsf{Int} \to \mathsf{Int}) <: *_{0\,(1)} (\mathsf{Int} \to \mathsf{Int}) \\
\texttt{xs} :&\quad *_{0\,(1)} \mathsf{Stream}(\mathsf{Int})
\end{aligned}$$

Putting it all Together: Mutual Recursion

Demonstration 4

Let us write `nat` and `spos`.

Putting it all Together: Fine-Grained Dependencies

Demonstration 5

Let us write `thuemorse`.

Outline

1 Introduction

2 Programming in a Language with Time Warps

3 Metatheoretical Aspects

4 Algorithmic Type Checking

5 Perspectives

A Two-Level Calculus

- Implicit terms correspond to user programs:

$$t ::= x \mid \texttt{fun}\,(x : \tau).t \mid t\ t \mid (t, t) \mid \texttt{pr}_{i\in\{0,1\}} t \mid \texttt{rec}\ (x : \tau).t \mid t\ \texttt{by}\ p$$

A Two-Level Calculus

- Implicit terms correspond to user programs:

$$t ::= x \mid \texttt{fun}\,(x:\tau).t \mid t\ t \mid (t,t) \mid \texttt{pr}_{i\in\{0,1\}}t \mid \texttt{rec}\ (x:\tau).t \mid t\ \texttt{by}\ p$$

- Explicit terms have coercions and syntax-directed typing rules:

$$\begin{aligned} e ::= {} & x \mid \texttt{fun}\,(x:\tau).e \mid e\ e \mid (e,e) \mid \texttt{pr}_{i\in\{0,1\}}e \mid \texttt{rec}\ (x:\tau).e \mid e\ \texttt{by}\ p \\ \mid {} & (t;\alpha) \mid (\gamma;t) \end{aligned}$$

$$\ldots \qquad \frac{\Gamma \vdash e : \tau \qquad \alpha : \tau <: \tau'}{\Gamma' \vdash (\alpha; e) : \tau'} \qquad \frac{\gamma : \Gamma' <: \Gamma \qquad \Gamma \vdash e : \tau}{\Gamma' \vdash (\gamma; e) : \tau}$$

A Two-Level Calculus

- Implicit terms correspond to user programs:

$$t ::= x \mid \texttt{fun}\,(x:\tau).t \mid t\ t \mid (t,t) \mid \texttt{pr}_{i\in\{0,1\}}t \mid \texttt{rec}\ (x:\tau).t \mid t\ \texttt{by}\ p$$

- Explicit terms have coercions and syntax-directed typing rules:

$$\begin{aligned} e ::= {} & x \mid \texttt{fun}\,(x:\tau).e \mid e\ e \mid (e,e) \mid \texttt{pr}_{i\in\{0,1\}}e \mid \texttt{rec}\ (x:\tau).e \mid e\ \texttt{by}\ p \\ \mid {} & (t;\alpha) \mid (\gamma;t) \end{aligned}$$

$$\ldots \qquad \frac{\Gamma \vdash e:\tau \qquad \alpha:\tau <: \tau'}{\Gamma' \vdash (\alpha;e):\tau'} \qquad \frac{\gamma:\Gamma' <: \Gamma \qquad \Gamma \vdash e:\tau}{\Gamma' \vdash (\gamma;e):\tau}$$

- Every explicit term e erases to a unique implicit term $\mathbf{U}(e)$.

The Dynamics of **Pulsar**

Pulsar enjoys two distinct semantics, both defined on explicit terms:

- Operational, as a big-step evaluation relation

$$\boxed{e; \sigma \Downarrow_n v}$$

The Dynamics of **Pulsar**

Pulsar enjoys two distinct semantics, both defined on explicit terms:

- Operational, as a big-step evaluation relation

$$\boxed{e; \sigma \Downarrow_n v}$$

- Denotational, as an interpretation in the *topos of trees*

$$[\![\tau]\!] \in |\widehat{\omega}| \qquad\qquad [\![\Gamma \vdash e : \tau]\!] \in \widehat{\omega}([\![\Gamma]\!], [\![\tau]\!])$$

Operational Semantics

$$v ::= \texttt{nil} \mid s \mid v :: v \mid (v, v) \mid (x.e)\{\sigma\} \mid (p, v)$$

Operational Semantics

$$v \quad ::= \quad \texttt{nil} \mid s \mid v :: v \mid (v, v) \mid (x.e)\{\sigma\} \mid (p, v)$$

$$\boxed{v : \tau \;\texttt{@}\; n}$$

$$\frac{}{\texttt{nil} : \tau \;\texttt{@}\; 0} \qquad \frac{v_1 : \tau \;\texttt{@}\; n+1 \qquad v_2 : \mathsf{Stream}(\tau) \;\texttt{@}\; n}{v_1 :: v_2 : \mathsf{Stream}(\tau) \;\texttt{@}\; n+1} \qquad \cdots \qquad \frac{v : \tau \;\texttt{@}\; p(n)}{(p, v) : *_p \tau \;\texttt{@}\; n}$$

Operational Semantics

$$v \quad ::= \quad \texttt{nil} \mid s \mid v :: v \mid (v, v) \mid (x.e)\{\sigma\} \mid (p, v)$$

$$\boxed{v : \tau \ @\ n}$$

$$\frac{}{\texttt{nil} : \tau \ @\ 0} \qquad \frac{v_1 : \tau \ @\ n+1 \qquad v_2 : \mathsf{Stream}(\tau) \ @\ n}{v_1 :: v_2 : \mathsf{Stream}(\tau) \ @\ n+1} \qquad \cdots \qquad \frac{v : \tau \ @\ p(n)}{(p, v) : *_p \tau \ @\ n}$$

$$\boxed{e; \sigma \Downarrow_n v}$$

$$\frac{}{e; \sigma \Downarrow_0 \texttt{nil}} \qquad \cdots \qquad \frac{e; \pi_2(\sigma) \Downarrow_{p(n)} v}{e \ \texttt{by} \ p; \sigma \Downarrow_n (p, v)} \qquad \frac{x.e; \sigma; \texttt{nil} \Uparrow_0^n v}{\texttt{rec} \ (x : \tau).e; \sigma \Downarrow_n v}$$

Operational Semantics

$$v \quad ::= \quad \texttt{nil} \mid s \mid v :: v \mid (v, v) \mid (x.e)\{\sigma\} \mid (p, v)$$

$$\boxed{v : \tau \mathbin{@} n}$$

$$\frac{}{\texttt{nil} : \tau \mathbin{@} 0} \qquad \frac{v_1 : \tau \mathbin{@} n+1 \qquad v_2 : \mathsf{Stream}(\tau) \mathbin{@} n}{v_1 :: v_2 : \mathsf{Stream}(\tau) \mathbin{@} n+1} \qquad \cdots \qquad \frac{v : \tau \mathbin{@} p(n)}{(p, v) : *_p \tau \mathbin{@} n}$$

$$\boxed{e; \sigma \Downarrow_n v}$$

$$\frac{}{e; \sigma \Downarrow_0 \texttt{nil}} \qquad \cdots \qquad \frac{e; \pi_2(\sigma) \Downarrow_{p(n)} v}{e \;\texttt{by}\; p; \sigma \Downarrow_n (p, v)} \qquad \frac{x.e; \sigma; \texttt{nil} \Uparrow_0^n v}{\texttt{rec}\ (x : \tau).e; \sigma \Downarrow_n v}$$

$$\boxed{x.e; \sigma; v \Uparrow_m^n v'}$$

$$\frac{m < n \qquad e; \lfloor \sigma \rfloor_m [x \mapsto v] \Downarrow_{v'} \qquad x.e; \sigma; v' \Uparrow_{m+1}^n v''}{x.e; \sigma; v \Uparrow_m^n v''} \qquad \frac{m \geq n}{x.e; \sigma; v \Uparrow_m^n v}$$

Denotational Semantics: the Topos of Trees

Definition:

$$\widehat{\omega} \quad \triangleq \quad [\omega^{op}, \mathbf{Set}]$$

Denotational Semantics: the Topos of Trees

Definition:

$$\widehat{\omega} \quad \triangleq \quad [\omega^{op}, \mathbf{Set}]$$

Concretely:

Denotational Semantics: the Topos of Trees

Definition:

$$\widehat{\omega} \quad \triangleq \quad [\omega^{op}, \mathbf{Set}]$$

Concretely:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \quad \dots$$

Denotational Semantics: the Topos of Trees

Definition:

$$\widehat{\omega} \quad \triangleq \quad [\omega^{op}, \mathbf{Set}]$$

Concretely:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \qquad \ldots$$

X

Denotational Semantics: the Topos of Trees

Definition:

$$\widehat{\omega} \quad \triangleq \quad [\omega^{op}, \mathbf{Set}]$$

Concretely:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \qquad \ldots$$

X

$$X(0) \xleftarrow{r_0^X} X(1) \xleftarrow{r_1^X} X(2) \xleftarrow{r_2^X} X(3) \xleftarrow{r_3^X} X(4) \qquad \ldots$$

Denotational Semantics: the Topos of Trees

Definition:

$$\widehat{\omega} \triangleq [\omega^{op}, \mathbf{Set}]$$

Concretely:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \qquad \ldots$$

$$X \qquad X(0) \xleftarrow{r_0^X} X(1) \xleftarrow{r_1^X} X(2) \xleftarrow{r_2^X} X(3) \xleftarrow{r_3^X} X(4) \qquad \ldots$$

$$Y \qquad Y(0) \xleftarrow{r_0^Y} Y(1) \xleftarrow{r_1^Y} Y(2) \xleftarrow{r_2^Y} Y(3) \xleftarrow{r_3^Y} Y(4) \qquad \ldots$$

Denotational Semantics: the Topos of Trees

Definition:

$$\widehat{\omega} \quad \triangleq \quad [\omega^{op}, \textbf{Set}]$$

Concretely:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \qquad \ldots$$

$$X \xrightarrow{f} Y$$

$$X(0) \xleftarrow{r_0^X} X(1) \xleftarrow{r_1^X} X(2) \xleftarrow{r_2^X} X(3) \xleftarrow{r_3^X} X(4) \qquad \ldots$$

$$Y(0) \xleftarrow{r_0^Y} Y(1) \xleftarrow{r_1^Y} Y(2) \xleftarrow{r_2^Y} Y(3) \xleftarrow{r_3^Y} Y(4) \qquad \ldots$$

Denotational Semantics: the Topos of Trees

Definition:

$$\widehat{\omega} \quad \triangleq \quad [\omega^{op}, \textbf{Set}]$$

Concretely:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \qquad \ldots$$

$$\begin{array}{c} X \\ \downarrow f \\ Y \end{array} \qquad \begin{array}{ccccccccccc}
X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \xleftarrow{r_3^X} & X(4) & & \ldots \\
\downarrow f_0 & & \downarrow f_1 & & \downarrow f_2 & & \downarrow f_3 & & \downarrow f_4 & & \\
Y(0) & \xleftarrow{r_0^Y} & Y(1) & \xleftarrow{r_1^Y} & Y(2) & \xleftarrow{r_2^Y} & Y(3) & \xleftarrow{r_3^Y} & Y(4) & & \ldots
\end{array}$$

Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \quad \leq \quad \dots$$

$$\texttt{Stream}\ \mathbb{B} \qquad \mathbb{B}^0 \underset{\texttt{take}_0}{\longleftarrow} \mathbb{B}^1 \underset{\texttt{take}_1}{\longleftarrow} \mathbb{B}^2 \underset{\texttt{take}_2}{\longleftarrow} \mathbb{B}^3 \underset{\texttt{take}_3}{\longleftarrow} \mathbb{B}^4 \underset{\texttt{take}_4}{\longleftarrow} \dots$$

$*_{(0\ 2)}\,\mathsf{Stream}(\mathbb{B})$

Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \quad \leq \quad \dots$$

$$\texttt{Stream}\ \mathbb{B} \qquad \mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \dots$$

$*_{(0\ 2)}\, \mathsf{Stream}(\mathbb{B})$

Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:

$$0 \quad \leq \quad 1 \quad \leq \quad 2 \quad \leq \quad 3 \quad \leq \quad 4 \quad \leq \quad \dots$$

$\texttt{Stream}\ \mathbb{B}$	$\mathbb{B}^0 \underset{\texttt{take}_0}{\longleftarrow} \mathbb{B}^1 \underset{\texttt{take}_1}{\longleftarrow} \mathbb{B}^2 \underset{\texttt{take}_2}{\longleftarrow} \mathbb{B}^3 \underset{\texttt{take}_3}{\longleftarrow} \mathbb{B}^4 \underset{\texttt{take}_4}{\longleftarrow} \dots$
$*_{(0\ 2)}\,\mathsf{Stream}(\mathbb{B})$	$\mathbb{B}^0$

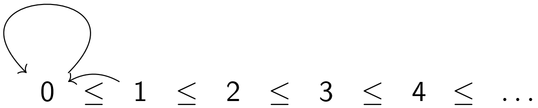
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



`Stream` $\mathbb{B}$ $\qquad \mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \ldots$

$*_{(0\ 2)}\,\mathrm{Stream}(\mathbb{B}) \qquad \mathbb{B}^0$

Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:

$$0 \leq 1 \leq 2 \leq 3 \leq 4 \leq \dots$$

$$\texttt{Stream}\ \mathbb{B} \qquad \mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \dots$$

$$*_{(0\ 2)}\,\mathsf{Stream}(\mathbb{B}) \qquad \mathbb{B}^0 \xleftarrow[id]{} \mathbb{B}^0$$

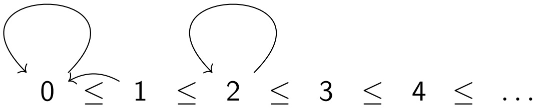
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



`Stream` $\mathbb{B}$	$\mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \ldots$
$*_{(0\ 2)}\,\mathsf{Stream}(\mathbb{B})$	$\mathbb{B}^0 \xleftarrow[id]{} \mathbb{B}^0$

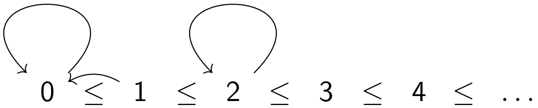
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



`Stream` $\mathbb{B}$	$\mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \ldots$
$*_{(0\ 2)}\,\mathsf{Stream}(\mathbb{B})$	$\mathbb{B}^0 \xleftarrow[id]{} \mathbb{B}^0 \xleftarrow[\texttt{take}_{0,1}]{} \mathbb{B}^2$

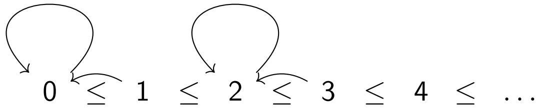
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



$\texttt{Stream}\ \mathbb{B}$ $\qquad \mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} {\color{red}\mathbb{B}^2} \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \dots$

$*_{(0\ 2)}\,\mathsf{Stream}(\mathbb{B})$ $\qquad \mathbb{B}^0 \xleftarrow[id]{} \mathbb{B}^0 \xleftarrow[\texttt{take}_{0,1}]{} \mathbb{B}^2$

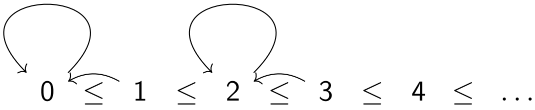
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



$\texttt{Stream}\ \mathbb{B}$ $\qquad \mathbb{B}^0 \underset{\texttt{take}_0}{\longleftarrow} \mathbb{B}^1 \underset{\texttt{take}_1}{\longleftarrow} \mathbb{B}^2 \underset{\texttt{take}_2}{\longleftarrow} \mathbb{B}^3 \underset{\texttt{take}_3}{\longleftarrow} \mathbb{B}^4 \underset{\texttt{take}_4}{\longleftarrow} \dots$

$*_{(0\ 2)}\, \mathsf{Stream}(\mathbb{B})$ $\qquad \mathbb{B}^0 \underset{id}{\longleftarrow} \mathbb{B}^0 \underset{\texttt{take}_{0,1}}{\longleftarrow} \mathbb{B}^2 \underset{id}{\longleftarrow} \mathbb{B}^2$

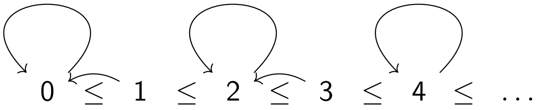
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



$\texttt{Stream}\ \mathbb{B}$ $\qquad \mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \ldots$

$*_{(0\ 2)}\,\mathsf{Stream}(\mathbb{B})$ $\qquad \mathbb{B}^0 \xleftarrow[id]{} \mathbb{B}^0 \xleftarrow[\texttt{take}_{0,1}]{} \mathbb{B}^2 \xleftarrow[id]{} \mathbb{B}^2$

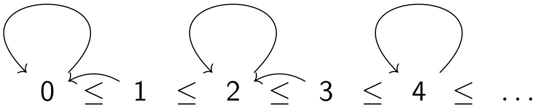
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



`Stream` $\mathbb{B}$ $\qquad \mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \dots$

$*_{(0\ 2)}$ Stream($\mathbb{B}$) $\qquad \mathbb{B}^0 \xleftarrow[id]{} \mathbb{B}^0 \xleftarrow[\texttt{take}_{0,1}]{} \mathbb{B}^2 \xleftarrow[id]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_{2,3}]{} \mathbb{B}^4$

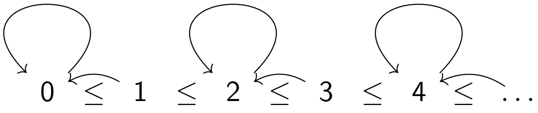
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



$\texttt{Stream}\ \mathbb{B}$ $\qquad \mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \color{red}{\mathbb{B}^4} \xleftarrow[\texttt{take}_4]{} \dots$

$*_{(0\ 2)}\,\mathsf{Stream}(\mathbb{B})$ $\qquad \mathbb{B}^0 \xleftarrow[id]{} \mathbb{B}^0 \xleftarrow[\texttt{take}_{0,1}]{} \mathbb{B}^2 \xleftarrow[id]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_{2,3}]{} \mathbb{B}^4 \qquad \dots$

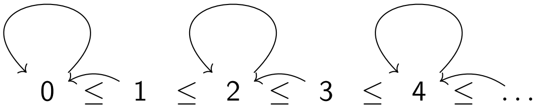
Denotational Semantics: the Interpretation, Abridged

The language is mostly interpreted by exploiting the cartesian-closed structure of toposes, following Birkedal et al. In addition:

- Warps are (isomorphic to) endofunctors of ω, and thus of ω^{op}.
- Thus, if X is a presheaf, so is $X \circ p$. In other words:

$$[\![*_p \, \tau]\!](n) = [\![\tau]\!](p(n))$$

For example:



Stream $\mathbb{B}$ $\qquad \mathbb{B}^0 \xleftarrow[\texttt{take}_0]{} \mathbb{B}^1 \xleftarrow[\texttt{take}_1]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_2]{} \mathbb{B}^3 \xleftarrow[\texttt{take}_3]{} \mathbb{B}^4 \xleftarrow[\texttt{take}_4]{} \dots$

$*_{(0\ 2)}$ Stream($\mathbb{B}$) $\qquad \mathbb{B}^0 \xleftarrow[id]{} \mathbb{B}^0 \xleftarrow[\texttt{take}_{0,1}]{} \mathbb{B}^2 \xleftarrow[id]{} \mathbb{B}^2 \xleftarrow[\texttt{take}_{2,3}]{} \mathbb{B}^4 \xleftarrow[id]{} \dots$

Conjectural Results

Operational Semantics: Soundness and Totality

If $\Gamma \vdash e : \tau$ and $\sigma : \Gamma \mathbin{@} n$, then there is v s.t. $e; \sigma \Downarrow_n v$ and $v : \tau \mathbin{@} n$.

Denotational Semantics: Adequacy

If $[\![\Gamma \vdash e : \tau]\!] = [\![\Gamma \vdash e' : \tau]\!]$ then $\Gamma \vdash e \cong_{\mathrm{ctx}} e' : \tau$.

Outline

1 Introduction

2 Programming in a Language with Time Warps

3 Metatheoretical Aspects

4 Algorithmic Type Checking

5 Perspectives

Subtyping and Coherence

In `map`, we used

$$\mathtt{f} : \mathsf{Int} \to \mathsf{Int} \equiv *_{(1)}(\mathsf{Int} \to \mathsf{Int}) <: *_{0\,(1)}(\mathsf{Int} \to \mathsf{Int})$$

In fact, the compiler did

$$\begin{aligned}\mathtt{f} :\ \mathsf{Int} \to \mathsf{Int} \equiv *_{(1)}(\mathsf{Int} \to \mathsf{Int}) &<: *_{0\,2\,(1)}(\mathsf{Int} \to \mathsf{Int}) \\ &= *_{0\,(1)\,*\,2\,(1)}(\mathsf{Int} \to \mathsf{Int}) \\ &\equiv *_{0\,(1)}\,*_{2\,(1)}(\mathsf{Int} \to \mathsf{Int})\end{aligned}$$

and then

$$*_{2\,(1)}(\mathsf{Int} \to \mathsf{Int}) <: *_{(1)}(\mathsf{Int} \to \mathsf{Int}) <: \mathsf{Int} \to \mathsf{Int}$$

Coherence Issues

- Distinct explicit terms mean different things *a priori*.
- Programmer writes implicit terms, compiler elaborates them into explicit ones. Is this reasonable?

Type-Checking and Elaboration

Goals

- Define an algorithm $\Gamma \vdash t \rightsquigarrow e : \tau$ taking (Γ, t) and returning (e, τ) such that $\mathbf{U}(e) = t$ and $\Gamma \vdash e : \tau$.
- Make this choice canonical in a certain sense.

Type-Checking and Elaboration

Goals

- Define an algorithm $\Gamma \vdash t \rightsquigarrow e : \tau$ taking (Γ, t) and returning (e, τ) such that $\mathbf{U}(e) = t$ and $\Gamma \vdash e : \tau$.
- Make this choice canonical in a certain sense.

In other words, we have to decide where to add $(-; \alpha)$ and $(\gamma; -)$ in t. This involves two main questions:

- Infer coercions $\alpha : \tau_1 <: \tau_2$ given τ_1 and τ_2.
- Infer coercions $\gamma : \Gamma_1 <: *_p \Gamma_2$ given Γ_1 and p.

Deciding Subtyping in Three Steps

1. The algorithmic judgment $\boxed{\tau \gg \tau^s \leadsto \alpha \rightleftarrows \alpha'}$:
   - implies $\alpha : \tau <: \tau^s$ and $\alpha' : \tau_s <: \tau$;
   - implies that τ^s respects the following grammar.

$$\begin{aligned} \tau^s &::= \ *_p\, \tau^r \mid \tau^s \times \tau^s \\ \tau^r &::= \ \nu \mid \mathsf{Stream}(\tau^s) \mid \tau^s \to \tau^s \end{aligned}$$

Deciding Subtyping in Three Steps

1. The algorithmic judgment $\boxed{\tau \gg \tau^s \leadsto \alpha \rightleftarrows \alpha'}$:
   - implies $\alpha : \tau <: \tau^s$ and $\alpha' : \tau_s <: \tau$;
   - implies that τ^s respects the following grammar.

$$\begin{aligned} \tau^s &::= \ *_p \, \tau^r \mid \tau^s \times \tau^s \\ \tau^r &::= \ \nu \mid \mathsf{Stream}(\tau^s) \mid \tau^s \to \tau^s \end{aligned}$$

2. The algorithmic judgment $\boxed{\tau \geq \tau' \leadsto \alpha}$:
   - implies $\alpha : \tau <: \tau'$;
   - holds iff τ is coercible to τ' using only delays.

Deciding Subtyping in Three Steps

1. The algorithmic judgment $\boxed{\tau \gg \tau^s \rightsquigarrow \alpha \rightleftarrows \alpha'}$:
   - implies $\alpha : \tau <: \tau^s$ and $\alpha' : \tau_s <: \tau$;
   - implies that τ^s respects the following grammar.

$$\begin{aligned} \tau^s &::= \ast_p \tau^r \mid \tau^s \times \tau^s \\ \tau^r &::= \nu \mid \mathsf{Stream}(\tau^s) \mid \tau^s \to \tau^s \end{aligned}$$

2. The algorithmic judgment $\boxed{\tau \geq \tau' \rightsquigarrow \alpha}$:
   - implies $\alpha : \tau <: \tau'$;
   - holds iff τ is coercible to τ' using only delays.
3. Algorithmic subtyping $\boxed{\tau_1 <: \tau_2 \rightsquigarrow \alpha}$ can then be defined by

$$\frac{\tau_1 \gg \tau_1^s \rightsquigarrow \alpha_1 \rightleftarrows - \qquad \tau_2 \gg \tau_2^s \rightsquigarrow - \rightleftarrows \alpha_3 \qquad \tau_1^s \geq \tau_2^s \rightsquigarrow \alpha_2}{\tau_1 <: \tau_2 \rightsquigarrow \alpha_1; \alpha_2; \alpha_3}$$

Type-Checking Warping

To type-check e `by` p in Γ, for any τ in Γ we must find τ^p such that

$$\tau <: *_p \tau^p$$

Type-Checking Warping

To type-check e `by` p in Γ, for any τ in Γ we must find τ^p such that

$$\tau <: *_p \tau^p$$

Moreover, to be complete we need the *best* solution, i.e. for any τ'

$$\tau <: *_p \tau' \Leftrightarrow \tau^p <: \tau'$$

Type-Checking Warping

To type-check e by p in Γ, for any τ in Γ we must find τ^p such that

$$\tau <: *_p \tau^p$$

Moreover, to be complete we need the *best* solution, i.e. for any τ'

$$\tau <: *_p \tau' \Leftrightarrow \tau^p <: \tau'$$

Fortunately, there exists an operation $-/p$ on types such that

$$\tau <: *_p \tau' \Leftrightarrow \tau/p <: \tau'$$

Type-Checking Warping

To type-check e by p in Γ, for any τ in Γ we must find τ^p such that

$$\tau <: \ast_p \tau^p$$

Moreover, to be complete we need the *best* solution, i.e. for any τ'

$$\tau <: \ast_p \tau' \Leftrightarrow \tau^p <: \tau'$$

Fortunately, there exists an operation $-/p$ on types such that

$$\tau <: \ast_p \tau' \Leftrightarrow \tau/p <: \tau'$$

It reduces to a similar operation on warps.

$$q \geq p * r \Leftrightarrow q/p \geq r$$

Warp Division

We are looking for a Galois connection $(-/g) \dashv (- \circ g)$.

$$h \circ g \leq f \Leftrightarrow h \leq f/g$$

Warp Division

We are looking for a Galois connection $(-/g) \dashv (- \circ g)$.

$$h \circ g \leq f \Leftrightarrow h \leq f/g$$

Such a thing exists for purely order-theoretic reasons. It can be built from the *right Kan extension* of f along g.

$$(\mathrm{Ran}_g(f))(n) = \bigsqcap f^*(g_*(\uparrow n))$$

Warp Division

We are looking for a Galois connection $(-/g) \dashv (- \circ g)$.

$$h \circ g \leq f \Leftrightarrow h \leq f/g$$

Such a thing exists for purely order-theoretic reasons. It can be built from the *right Kan extension* of f along g.

$$(\mathrm{Ran}_g(f))(n) = \prod f^*(g_*(\uparrow n))$$

The right Kan extension is presentable by a ultimately periodic sequence when both f and g are, and can be computed.

$$(1)/0\,(1) = 2\,(1) \qquad (1)/(2) = (1\,0) \qquad (1\,0)/(1\,0) = (1)$$

$$(1)/(0) = (\omega) \qquad (1)/(\omega) = 1\,(0)$$

Main Results

Coherence of Subtyping

If $\alpha : \tau_1 <: \tau_2$ and $\alpha' : \tau_1 <: \tau_2$ then

$$[\![\alpha : \tau_1 <: \tau_2]\!] = [\![\alpha' : \tau_1 <: \tau_2]\!]$$

Main Results

Coherence of Subtyping

If $\alpha : \tau_1 <: \tau_2$ and $\alpha' : \tau_1 <: \tau_2$ then

$$[\![\alpha : \tau_1 <: \tau_2]\!] = [\![\alpha' : \tau_1 <: \tau_2]\!]$$

Completeness of Type-Checking

For any $\Gamma \vdash e : \tau$, there is e_m, τ_m, α such that

$$\Gamma \vdash \mathbf{U}(e) \rightsquigarrow e_m : \tau_m \qquad \Gamma \vdash e_m : \tau_m \qquad \alpha : \tau_m <: \tau$$

$$[\![\Gamma \vdash (e_m; \alpha) : \tau]\!] = [\![\Gamma \vdash e : \tau]\!]$$

Main Results

Coherence of Subtyping

If $\alpha : \tau_1 <: \tau_2$ and $\alpha' : \tau_1 <: \tau_2$ then

$$[\![\alpha : \tau_1 <: \tau_2]\!] = [\![\alpha' : \tau_1 <: \tau_2]\!]$$

Completeness of Type-Checking

For any $\Gamma \vdash e : \tau$, there is e_m, τ_m, α such that

$$\Gamma \vdash \mathbf{U}(e) \leadsto e_m : \tau_m \qquad \Gamma \vdash e_m : \tau_m \qquad \alpha : \tau_m <: \tau$$

$$[\![\Gamma \vdash (e_m; \alpha) : \tau]\!] = [\![\Gamma \vdash e : \tau]\!]$$

Corollary: Coherence

If $\Gamma \vdash e_1 : \tau$ and $\Gamma \vdash e_2 : \tau$ then

$$[\![\Gamma \vdash e_1 : \tau]\!] = [\![\Gamma \vdash e_2 : \tau]\!]$$

Outline

1 Introduction

2 Programming in a Language with Time Warps

3 Metatheoretical Aspects

4 Algorithmic Type Checking

5 Perspectives

Future Work

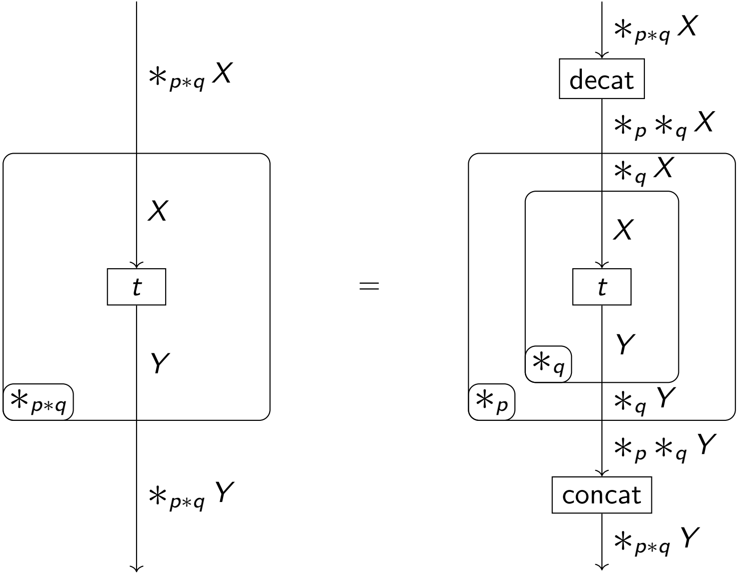
Frontend

- More ambitious subtyping.
- Type inference.

Backend

- Single-loop code generation.
- Typing restrictions to run within finite space.
-

What I Didn't Talk About



Conclusion

- I have presented a higher-order language with a rich notion of time. It handles programs that were previously out of reach of both synchronous languages and guarded type theories.
- Certain aspects of synchronous dataflow languages can be generalized through semantical intuitions in a natural way. I believe that this approach could be pushed much further.

Thank you!