A Functional Synchronous Language with Time Warps

Adrien Guatto

Inria Paris

Gallium - 06/11/2017
Streams in Programs and Proofs

- Infinite sequences of values
  \[
  \text{Stream}(X) \simeq \mathbb{N} \rightarrow X
  \]

- Kahn’s insight: a deterministic reactive system can be described as a mathematical function
  \[
  \text{Stream}(X) \rightarrow \text{Stream}(Y)
  \]

- 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.
- Streams, as infinite objects, have to be introduced via self-referential definitions.

- For example, zeroes can be characterized as the solution of

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

  and defined as such in Haskell, Lustre, and Coq.

- What about equations with several or no solutions?

  $$\text{weird} = \text{weird}$$

  Different languages follow different approaches.
Streams, as infinite objects, have to be introduced via self-referential definitions.

For example, zeroes can be characterized as the solution of

\[ \text{zeroes} = 0 :: \text{zeroes} \]

and defined as such in Haskell, Lustre, and Coq.

What about equations with several or no solutions?

\[ \text{weird} = \text{weird} \]

Different languages follow different approaches.

Demonstration 1

Try the above in our three prototypical languages.
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

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
Nakano’s Key Idea

In a recursive definition, self-references are only available later.
Nakano’s Key Idea

In a recursive definition, self-references are only available later.
Nakano’s Key Idea

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

Formally:
Nakano’s Key Idea

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

Formally:

- Enrich the type language with a modality

\[
\tau ::= \cdots \mid ▶\tau
\]

and related operations.
Nakano’s Key Idea

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

Formally:

- Enrich the type language with a modality

\[
\tau ::= \cdots | \triangleright\tau
\]

and related operations.

- Give appropriate types to the stream constructor/destructors;

\[
(\::) : X \rightarrow \triangleright\text{Stream}(X) \rightarrow \text{Stream}(X)
\]

\[
\text{head} : \text{Stream}(X) \rightarrow X \quad \text{tail} : \text{Stream}(X) \rightarrow \triangleright\text{Stream}(X)
\]
Nakano’s Key Idea

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

Formally:

- Enrich the type language with a modality

\[ \tau ::= \cdots | \mathbf{\triangleright} \tau \]

and related operations.
- Give appropriate types to the stream constructor/destructors;

\[
(\mathbf{::}) : X \to \mathbf{\triangleright} \text{Stream}(X) \to \text{Stream}(X)
\]

\[
\text{head} : \text{Stream}(X) \to X \quad \text{tail} : \text{Stream}(X) \to \mathbf{\triangleright} \text{Stream}(X)
\]

- Have a special typing rule for recursive definitions.
Guarded Recursive Definitions

\[ \Gamma, x : \tau \vdash t : \tau \]
\[ \frac{}{\Gamma \vdash \text{rec} (x : \tau).t : \tau} \]
Guarded Recursive Definitions

\[ \Gamma, x : \tau \vdash t : \tau \]
\[ \Gamma \vdash \text{rec } (x : \tau).t : \tau \]

- The following definition is well-typed.

\[ \text{zeroes} = 0 :: \text{zeroes} \]
The following definition is well-typed.

\[ \text{rec } (\text{zeroes} : \text{Stream}(	ext{Int})).(0 :: \text{zeroes}) \]
Guarded Recursive Definitions

\[ \Gamma, x : \tau \vdash t : \tau \]
\[ \Gamma \vdash \text{rec (} x : \tau \text{)}.t : \tau \]

- The following definition is well-typed.
  \[ \text{rec (zeroes : Stream(Int))}.(0 :: zeroes) \]

- This one is not:
  \[ \text{rec (weird : Stream(Int))}.\text{weird} \]
Guarded Recursive Definitions

\[ \Gamma, x : \tau \vdash t : \tau \]
\[ \Gamma \vdash \text{rec}(x : \tau).t : \tau \]

- The following definition is well-typed.
  \[ \text{rec}(\text{zeroes} : \text{Stream}(\text{Int})).(0 :: \text{zeroes}) \]

- This one is not:
  \[ \text{rec}(\text{weird} : \text{Stream}(\text{Int})).\text{weird} \]

“This expression has type \(\text{Stream}(\text{Int})\) but was expected to have type \(\text{Stream}(\text{Int})\).”
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

\[
\text{nat} = 0 :: \text{map} (\lambda x. x + 1) \text{nat}
\]

to step-indexed models of programming languages.
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

\[
\text{nat} = 0 :: \text{map} (\lambda x. x + 1) \text{nat}
\]

to step-indexed models of programming languages.

However, current guarded type theories struggle with...

- mutual recursion:

\[
\text{nat} = 0 :: \text{spos} \quad \text{spos} = \text{map} (\lambda x. x + 1) \text{nat}
\]
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

\[
\text{nat} = 0 :: \text{map} (\lambda x. x + 1) \text{nat}
\]

to step-indexed models of programming languages.

- However, current guarded type theories struggle with...
  - mutual recursion:

\[
\text{nat} = 0 :: \text{spos} \quad \text{spos} = \text{map} (\lambda x. x + 1) \text{nat}
\]

- fine-grained dependencies:

\[
\text{thuemorse} = \text{false} :: \text{tail} (h \text{thuemorse})
\]

where \( h (x :: xs) = x :: (\text{not } x) :: h \ xs \)
Models of guarded recursion interpret types by $\omega$-indexed families of sets of observations.

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

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

$$(\text{\uparrow Stream}(\text{Int}))_n \approx \text{Int}^{n-1}$$
Models of guarded recursion interpret types by $\omega$-indexed families of sets of observations.

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

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

$$(\triangleright \text{Stream}(\text{Int}))_n \approx \text{Int}^{n-1}$$
Models of guarded recursion interpret types by $\omega$-indexed families of sets of observations.

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

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

$$(\upRightarrow \text{Stream}(\text{Int}))_n \approx \text{Int}^{n-1}$$

Main Claims of This Talk

By considering a large class of transformations, time warps, we obtain a very general parametric modality.
Models of guarded recursion interpret types by $\omega$-indexed families of sets of observations.

\[(\text{Stream}(\text{Int}))_n \approx \text{Int}^n\]

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

\[(\triangleright \text{Stream}(\text{Int}))_n \approx \text{Int}^{n-1}\]

Main Claims of This Talk

- By considering a large class of transformations, \textit{time warps}, we obtain a very general parametric modality.
- It is possible to design a type system around this modality to make it both \textit{usable} and \textit{implementable}.
Models of guarded recursion interpret types by $\omega$-indexed families of sets of observations.

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

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

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

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.
1 Introduction

2 Programming in a Language with Time Warps

3 Metatheoretical Aspects

4 Algorithmic Type Checking

5 Perspectives
1 Introduction

2 Programming in a Language with Time Warps

3 Metatheoretical Aspects

4 Algorithmic Type Checking

5 Perspectives
• **Pulsar** is based on the simply-typed $\lambda$-calculus extended with a built-in stream type.

\[
\tau ::= \nu \mid \text{Stream}(\tau) \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \ldots
\]

\[
\nu ::= \text{Int} \mid \text{Bool} \mid \text{Char}
\]

\[
\Gamma(x) = \tau \quad \frac{\Gamma \vdash x : \tau_{1} \quad t : \tau_{2}}{\Gamma \vdash \text{fun}(x : \tau_{1}).t : \tau_{2}} \quad \frac{\Gamma \vdash t_{1} : \tau_{1} \rightarrow \tau_{2} \quad \Gamma \vdash t_{2} : \tau_{1}}{\Gamma \vdash t_{1} \ t_{2} : \tau_{2}}
\]

To the above, it adds the warp modality $\tau ::= \ldots \mid \ast p \tau$ plus guarded recursion, subtyping, and a new construct.
■ **Pulsar** is based on the simply-typed $\lambda$-calculus extended with a built-in stream type.

\[
\tau := \nu \mid \text{Stream}(\tau) \mid \tau \to \tau \mid \tau \times \tau \mid \ldots
\]

\[
\nu := \text{Int} \mid \text{Bool} \mid \text{Char}
\]

- $\Gamma(x) = \tau$  
  $\Gamma, x : \tau_1 \vdash t : \tau_2$  
  $\Gamma \vdash t_1 : \tau_1 \to \tau_2$  
  $\Gamma \vdash t_2 : \tau_1$

- $\Gamma \vdash x : \tau$  
  $\Gamma \vdash \text{fun}(x : \tau_1). t : \tau_2$  
  $\Gamma \vdash t_1 \ t_2 : \tau_2$

■ To the above, it adds the warp modality

\[
\tau ::= \ldots \mid \ast_p \tau
\]

plus guarded recursion, subtyping, and a new construct.
Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \quad f(\omega) = \bigsqcup_{i<\omega} f(i)$$
Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

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

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

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

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:

$$\mathcal{O} (0)(i) = 0$$
Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

\[ f(0) = 0 \quad f(\omega) = \bigsqcup_{i < \omega} f(i) \]

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:

\[(\mathcal{O} (0))(i) = 0 \quad (\mathcal{O} (1))(i) = i\]
Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that
\[
f(0) = 0 \quad f(\omega) = \bigsqcup_{i<\omega} f(i)
\]

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

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

For example:
\[
(O \, (0))(i) = 0 \quad (O \, (1))(i) = i \quad (O \, 0 \, (1))(i) = i - 1
\]
Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

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

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:

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

$$\mathcal{(O \ (2))}(i) = 2i$$
Formally, warps are sup-preserving functions from $\omega + 1$ to itself, i.e. monotonic functions such that

$$f(0) = 0 \quad f(\omega) = \bigcup_{i < \omega} f(i)$$

We restrict ourselves to warps defined as running sums $O\ p$ of ultimately periodic number sequences $p$.

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

For example:

$$(O\ (0))(i) = 0 \quad (O\ (1))(i) = i \quad (O\ 0\ (1))(i) = i - 1$$

$$(O\ (2))(i) = 2i \quad (O\ (\omega))(i) = \omega \text{ for } i > 0$$
Demonstration 2
Let us try to write zeroes.
Demonstration 2

Let us try to write zeroes.

Guarded recursion is formulated with $\triangleright \tau \triangleq \ast_0(1) \tau$ as expected.

$$\Gamma, x : \ast_0(1) \tau \vdash e : \tau$$  
$$\Gamma \vdash \text{rec } (x : \tau). e : \tau$$

Similarly, primitives have types

$$(::) : \tau \rightarrow \ast_0(1) \text{Stream}(\tau) \rightarrow \text{Stream}(\tau)$$

$\text{head} : \text{Stream}(\tau) \rightarrow \tau$  
$\text{tail} : \text{Stream}(\tau) \rightarrow \ast_0(1) \text{Stream}(\tau)$
Demonstration 3
Let us try to write weird.
Demonstration 3

Let us try to write \textit{weird}.

As expected, there is no \( \tau \) such that \( \vdash \text{weird} : \text{Stream}(\tau) \) holds. However, \( \vdash \text{weird} : \star_{(0)} \text{Stream}(\tau) \) holds for any \( \tau \). Why?
Demonstration 3

Let us try to write weird.

As expected, there is no $\tau$ such that $\vdash \text{weird} : \text{Stream}(\tau)$ holds. However, $\vdash \text{weird} : *_{(0)} \text{Stream}(\tau)$ holds for any $\tau$. Why?

\[
*_{p*q} \tau \equiv *_{p} *_{q} \tau
\]

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

\[
\mathcal{O}(p * q) = \mathcal{O} q \circ \mathcal{O} p
\]

hence we have

\[
*_{0(1)} *_{(0)} \text{Stream}(\tau) \equiv *_{0(1) *} *_{(0)} \text{Stream}(\tau) \equiv *_{(0)} \text{Stream}(\tau)
\]
Demonstration 3
Let us try to write $\text{map}$. 
Demonstration 3

Let us try to write `map`.

$$\Gamma \vdash e : \tau$$

$$\ast_p \Gamma \vdash e \text{ by } p : \ast_p \tau$$
Demonstration 3

Let us try to write \( \text{map} \).

\[
\Gamma \vdash e : \tau \\
\ast_{\rho} \Gamma \vdash e \text{ by } \rho : \ast_{\rho} \tau
\]

\[
\text{map} : \ast_{0(1)} ((\text{Int} \rightarrow \text{Int}) \rightarrow \text{Stream}(\text{Int}) \rightarrow \text{Stream}(\text{Int})) \\
\text{f} : \text{Int} \rightarrow \text{Int} \\
\text{xs} : \ast_{0(1)} \text{Stream}(\text{Int})
\]
Demonstration 3

Let us try to write \texttt{map}.

\[
\Gamma \vdash e : \tau \\
\star_p \Gamma \vdash e \text{ by } p : \star_p \tau \\
\tau \equiv \star(1) \tau
\]

\[
\text{map} : \star(1)_0 ((\text{Int} \to \text{Int}) \to \text{Stream}((\text{Int} \to \text{Int}) \to \text{Stream}((\text{Int} \to \text{Int})))
\]

\[
f : \text{Int} \to \text{Int} \equiv \star(1) (\text{Int} \to \text{Int})
\]

\[
\text{xs} : \star(1)_0 \text{Stream}((\text{Int})
\]
Demonstration 3
Let us try to write \( \text{map} \).

\[
\begin{align*}
\Gamma &\vdash e : \tau \\
\ast_p \Gamma &\vdash e \text{ by } p : \ast_p \tau \\
\tau &\equiv \ast_1 \tau \\
q &\leq p \\
\ast_p \tau &<: \ast_q \tau
\end{align*}
\]

\[
\text{map} : \ast_0 (\text{Int} \rightarrow \text{Int}) \rightarrow \text{Stream} (\text{Int} \rightarrow \text{Stream} (\text{Int}))
\]

\[
f : \text{Int} \rightarrow \text{Int} \equiv \ast_1 (\text{Int} \rightarrow \text{Int}) <: \ast_0 (\text{Int} \rightarrow \text{Int})
\]

\[
\text{x} : \ast_0 (\text{Stream} (\text{Int}))
\]
Demonstration 4

Let us write nat and spos.
Demonstration 5

Let us write thuemorse.
1 Introduction
2 Programming in a Language with Time Warps
3 Metatheoretical Aspects
4 Algorithmic Type Checking
5 Perspectives
Implicit terms correspond to user programs:

\[
\begin{align*}
t & ::= x \mid \text{fun}(x : \tau).t \mid t t \mid (t, t) \mid \text{pr}_{i \in \{0, 1\}} t \mid \text{rec}(x : \tau).t \mid t \text{ by } p \\
\end{align*}
\]
Implicit terms correspond to user programs:

\[ t ::= x \mid \text{fun}(x : \tau).t \mid t \; t \mid (t, t) \mid \text{pr}_{i \in \{0, 1\}} t \mid \text{rec} (x : \tau).t \mid t \; \text{by} \; p \]

Explicit terms have coercions and syntax-directed typing rules:

\[ e ::= x \mid \text{fun}(x : \tau).e \mid e \; e \mid (e, e) \mid \text{pr}_{i \in \{0, 1\}} e \mid \text{rec} (x : \tau).e \mid e \; \text{by} \; p \mid (t; \alpha) \mid (\gamma; t) \]

\[
\begin{align*}
\Gamma \vdash e : \tau & \quad \alpha : \tau \; <: \; \tau' \\
\Gamma' \vdash (\alpha; e) : \tau' & \quad \gamma : \Gamma' \; <: \; \Gamma \\
\Gamma' \vdash \gamma; e : \tau & \quad \Gamma' \vdash (\gamma; e) : \tau
\end{align*}
\]
Implicit terms correspond to user programs:

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

Explicit terms have coercions and syntax-directed typing rules:

\[ e ::= x \mid \text{fun}(x : \tau).e \mid e \, e \mid (e, e) \mid \text{pr}_{i \in \{0,1\}}e \mid \text{rec}(x : \tau).e \mid e \, \text{by} \, p \mid (t; \alpha) \mid (\gamma; t) \]

\[ \begin{align*}
\Gamma & \vdash \alpha : \tau \\
\alpha & \vdash \tau' \\
\gamma & \vdash \Gamma' <: \Gamma \\
\Gamma & \vdash \Gamma' <: \Gamma \\
\Gamma & \vdash e : \tau
\end{align*} \]

\[ \begin{align*}
\Gamma' & \vdash (\alpha; e) : \tau' \\
\Gamma' & \vdash (\gamma; e) : \tau
\end{align*} \]

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
  \[ e;\sigma \Downarrow^*_n v \]
The Dynamics of **Pulsar**

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

- Operational, as a big-step evaluation relation
  
  \[ e; \sigma \Downarrow^n v \]

- Denotational, as an interpretation in the *topos of trees*
  
  \[ [\tau] \in |\hat{\omega}| \quad \quad [\Gamma \vdash e : \tau] \in \hat{\omega}([\Gamma], [\tau]) \]
\( v ::= \) \( \) nil \( \) \( | \) \( s \) \( \) \( | \) \( v :: v \) \( \) \( | \) \( (v, v) \) \( \) \( | \) \( (x. e)\{\sigma}\) \( \) \( | \) \( (p, v) \) \( \)
v ::= nil | s | v :: v | (v, v) | (x.e){σ} | (p, v)

\[
\begin{array}{c}
\text{nil : } \tau @ 0 \\
\text{v : } \tau @ n
\end{array}
\]

\[
\begin{array}{cccc}
v_1 : \tau @ n + 1 & v_2 : \text{Stream}(\tau) @ n \\
\hline
v_1 :: v_2 : \text{Stream}(\tau) @ n + 1 & \cdots & v : \tau @ p(n) \\
\hline
(p, v) : \ast_p \tau @ n
\end{array}
\]
\[ \nu \ ::= \ \text{nil} \mid s \mid \nu :: \nu \mid (\nu, \nu) \mid (x.\nu)\{\sigma\} \mid (p, \nu) \]

\[
\begin{array}{c}
\text{nil} : \tau \oplus 0 \\
v_1 : \tau \oplus n + 1 \\
v_2 : \text{Stream}(\tau) \oplus n
\end{array} \quad
\begin{array}{c}
\vdash v_1 :: v_2 : \text{Stream}(\tau) \oplus n + 1
\end{array} \quad
\begin{array}{c}
\vdash (p, \nu) : \ast_p \tau \oplus n
\end{array}
\]

\[
\begin{array}{c}
e; \sigma \downarrow_n v
\end{array} \quad
\begin{array}{c}
e; \pi_2(\sigma) \downarrow_{p(n)} v
\end{array} \quad
\begin{array}{c}
x.\nu; \sigma; \text{nil} \uparrow^n_{0} v
\end{array}
\]

\[
\begin{array}{c}
e; \sigma \downarrow_0 \text{nil}
\end{array} \quad
\begin{array}{c}
e \text{ by } p; \sigma \downarrow_n (p, \nu)
\end{array} \quad
\begin{array}{c}
\text{rec } (x : \tau).e; \sigma \downarrow_n v
\end{array}
\]
Operational Semantics

\[
v ::= \text{nil} \mid s \mid v :: v \mid (v, v) \mid (x.e)\{\sigma\} \mid (p, v)
\]

\[
\begin{array}{c}
v : \tau \@ n \\
\hline
nil : \tau \@ 0
\end{array}
\]

\[
\begin{array}{c}
v_1 : \tau \@ n + 1 \quad v_2 : \text{Stream}(\tau) \@ n \\
\hline
v_1 :: v_2 : \text{Stream}(\tau) \@ n + 1
\end{array}
\]

\[
\begin{array}{c}
\vdots
\end{array}
\]

\[
(v : \tau \@ p(n))
\]

\[
\begin{array}{c}
e; \sigma \Downarrow_n v \\
\hline
e; \sigma \Downarrow_0 \text{nil}
\end{array}
\]

\[
\begin{array}{c}
e; \pi_2(\sigma) \Downarrow_{p(n)} v \\
\hline
e \text{ by } p; \sigma \Downarrow_n (p, v)
\end{array}
\]

\[
\begin{array}{c}
x.e; \sigma; \text{nil} \uparrow^n v \\
\hline
\text{rec } (x : \tau).e; \sigma \Downarrow_n v
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
m < n \\
\hline
x.e; [\sigma]_m[x \mapsto v] \Downarrow_{v'}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
m \geq n \\
\hline
x.e; \sigma; v \uparrow^n v
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
m < n \\
\hline
x.e; \sigma; v' \uparrow_{m+1} v''
\end{array}
\end{array}
\end{array}
\]

\[
\begin{array}{c}
\begin{array}{c}
\begin{array}{c}
m \geq n \\
\hline
x.e; \sigma; v \uparrow^n v
\end{array}
\end{array}
\end{array}
\]
Definition:

\[ \hat{\omega} \triangleq [\omega^{op}, \textbf{Set}] \]
Definition:

\[ \hat{\omega} \triangleq [\omega^{op}, \text{Set}] \]

Concretely:
Denotational Semantics: the Topos of Trees

Definition:
\[ \hat{\omega} \triangleq [\omega^{op}, \text{Set}] \]

Concretely:

\[
\begin{align*}
0 & \leq 1 & \leq 2 & \leq 3 & \leq 4 & \ldots
\end{align*}
\]
Definition:

\[ \hat{\omega} \triangleq [\omega^{\text{op}}, \text{Set}] \]

Concretely:

\[
\begin{array}{cccccc}
0 & \leq & 1 & \leq & 2 & \leq & 3 & \leq & 4 & \ldots \\
\end{array}
\]
Denotational Semantics: the Topos of Trees

Definition:

\[ \hat{\omega} \triangleq [\omega^{op}, \text{Set}] \]

Concretely:

\[
\begin{array}{ccccccc}
0 & \leq & 1 & \leq & 2 & \leq & 3 & \leq & 4 & \ldots \\
\end{array}
\]

\[
\begin{array}{ccccccc}
X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \xleftarrow{r_3^X} & X(4) & \ldots \\
\end{array}
\]
Definition:
\[ \hat{\omega} \triangleq \left[ \omega^{op}, \text{Set} \right] \]

Concretely:
\[
\begin{array}{cccccc}
0 & \leq & 1 & \leq & 2 & \leq & 3 & \leq & 4 & \ldots \\
\hline
X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \xleftarrow{r_3^X} & X(4) & \ldots \\
Y(0) & \xleftarrow{r_0^Y} & Y(1) & \xleftarrow{r_1^Y} & Y(2) & \xleftarrow{r_2^Y} & Y(3) & \xleftarrow{r_3^Y} & Y(4) & \ldots \\
\end{array}
\]
Definition:

\[ \hat{\omega} \triangleq [\omega^{op}, \text{Set}] \]

Concretely:

\[
\begin{align*}
0 & \leq 1 & \leq 2 & \leq 3 & \leq 4 & \ldots \\
\end{align*}
\]

\[
\begin{array}{cccc}
X(0) & \xleftarrow{r_0^X} & X(1) & \xleftarrow{r_1^X} & X(2) & \xleftarrow{r_2^X} & X(3) & \xleftarrow{r_3^X} & X(4) & \ldots \\
Y(0) & \xleftarrow{r_0^Y} & Y(1) & \xleftarrow{r_1^Y} & Y(2) & \xleftarrow{r_2^Y} & Y(3) & \xleftarrow{r_3^Y} & Y(4) & \ldots \\
\end{array}
\]
Definition:

\[ \hat{\omega} \triangleq [\omega^{op}, \text{Set}] \]

Concretely:

\[
\begin{align*}
0 & \leq 1 & \leq 2 & \leq 3 & \leq 4 & \ldots \\
X(0) & \underset{r_0^X}{\leftarrow} X(1) & \underset{r_1^X}{\leftarrow} X(2) & \underset{r_2^X}{\leftarrow} X(3) & \underset{r_3^X}{\leftarrow} X(4) & \ldots \\
Y(0) & \underset{r_0^Y}{\leftarrow} Y(1) & \underset{r_1^Y}{\leftarrow} Y(2) & \underset{r_2^Y}{\leftarrow} Y(3) & \underset{r_3^Y}{\leftarrow} Y(4) & \ldots
\end{align*}
\]
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$\llbracket \star_p \tau \rrbracket(n) = \llbracket \tau \rrbracket(p(n))$$

For example:

\[
\begin{array}{cccccc}
0 & \leq & 1 & \leq & 2 & \leq & 3 & \leq & 4 & \leq & \ldots
\end{array}
\]

\[
\begin{array}{cccccccc}
\text{Stream } & B & & B^0 & \xleftarrow{\text{take}_0} & B^1 & \xleftarrow{\text{take}_1} & B^2 & \xleftarrow{\text{take}_2} & B^3 & \xleftarrow{\text{take}_3} & B^4 & \xleftarrow{\text{take}_4} & \ldots
\end{array}
\]

\[\star_{(0 \ 2)} \text{ Stream}(B)\]
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$\llbracket \ast_p \tau \rrbracket(n) = \llbracket \tau \rrbracket(p(n))$$

For example:

\[
\begin{array}{cccccc}
0 & \leq & 1 & \leq & 2 & \leq & 3 & \leq & 4 & \leq & \ldots
\end{array}
\]

Stream $B$

\[
\begin{array}{cccccc}
B_0 & \leftarrow & B_1 & \leftarrow & B_2 & \leftarrow & B_3 & \leftarrow & B_4 & \leftarrow & \ldots
\end{array}
\]

$\ast(0, 2)\text{ Stream}(B)$
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$[\ast_p \tau](n) = [\tau](p(n))$$

For example: 

$$\ast(0 \ 2) \ \text{Stream}(\mathbb{B}) \quad \mathbb{B}^0$$
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 \( \omega \), and thus of \( \omega^{op} \).
- Thus, if \( X \) is a presheaf, so is \( X \circ p \). In other words:

\[
\llbracket *_p \tau \rrbracket (n) = \llbracket \tau \rrbracket (p(n))
\]

For example:

\[
\begin{align*}
0 & \leq 1 \leq 2 \leq 3 \leq 4 \leq \ldots \\
\text{Stream } B & \\
& \quad \text{take}_0 \quad \text{take}_1 \quad \text{take}_2 \quad \text{take}_3 \quad \text{take}_4 \quad \ldots \\
\ast_{(0 \ 2)} \text{Stream}(B) & \\
& \quad B^0
\end{align*}
\]
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$\[\ast_p \tau\](n) = \[\tau\](p(n))$$

For example:

Stream $B$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
</table>

\[\ast(0 \ 2)\ \text{Stream}(B)\]

$$B^0 \xleftarrow{id} B^0$$
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$[[\ast_p \tau]](n) = [[\tau]](p(n))$$

For example:

$$\begin{array}{c}
0 \leq 1 \leq 2 \leq 3 \leq 4 \leq \ldots \\
\text{Stream } \mathbb{B} \\
\ast_{(0,2)} \text{Stream(}\mathbb{B}) \\
\end{array}$$
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$[[\ast_p \tau]](n) = [[\tau]](p(n))$$

For example:

$$\ast(0, 2) \text{ Stream } B$$

$$\begin{align*}
\text{Stream } B & \quad B^0 \leftarrow B^1 \leftarrow B^2 \leftarrow B^3 \leftarrow B^4 \leftarrow \ldots \\
& \quad \text{take}_0 \quad \text{take}_1 \quad \text{take}_2 \quad \text{take}_3 \quad \text{take}_4 \ldots \\
\ast(0, 2) \text{ Stream } B & \quad B^0 \leftarrow id \quad B^0 \leftarrow B^2 \\
& \quad \text{take}_{0,1} \end{align*}$$
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$\lbrack p \tau \rbrack(n) = \lbrack \tau \rbrack(p(n))$$

For example:

Stream $\mathbb{B}$

$\mathbb{B}^0 \leftarrow \mathbb{B}^1 \leftarrow \mathbb{B}^2 \leftarrow \mathbb{B}^3 \leftarrow \mathbb{B}^4 \leftarrow \ldots$

$\star (0, 2)$ Stream($\mathbb{B}$)

$\mathbb{B}^0 \leftarrow \mathbb{B}^0 \leftarrow \mathbb{B}^2$

$\mathbb{B}^0 \leftarrow \mathbb{B}^1 \leftarrow \mathbb{B}^2 \leftarrow \mathbb{B}^3 \leftarrow \mathbb{B}^4 \leftarrow \ldots$

$\mathbb{B}^0 \leftarrow \mathbb{B}^0 \leftarrow \mathbb{B}^2$
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$\llbracket p \tau \rrbracket(n) = \llbracket \tau \rrbracket(p(n))$$

For example:

$$\llbracket \ast p \tau \rrbracket(n) = \llbracket \tau \rrbracket(p(n))$$
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 \( \omega \), and thus of \( \omega^{op} \).
- Thus, if \( X \) is a presheaf, so is \( X \circ p \). In other words:

\[
[\star_p \tau](n) = [\tau](p(n))
\]

For example:

\[
\begin{align*}
\mathbb{B}^0 & \xleftarrow{id} \mathbb{B}^0 \\
\mathbb{B}^{0} & \xleftarrow{\text{take}_0} \mathbb{B}^1 \\
\mathbb{B}^{1} & \xleftarrow{\text{take}_1} \mathbb{B}^2 \\
\mathbb{B}^{2} & \xleftarrow{\text{take}_2} \mathbb{B}^3 \\
\mathbb{B}^{3} & \xleftarrow{\text{take}_3} \mathbb{B}^4 \\
\mathbb{B}^{4} & \xleftarrow{\text{take}_4} \ldots
\end{align*}
\]
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$\llbracket *_p \tau \rrbracket(n) = \llbracket \tau \rrbracket(p(n))$$

For example:

\[
\begin{array}{cccccc}
0 & \leq & 1 & \leq & 2 & \leq & 3 & \leq & 4 & \leq & \ldots \\
\end{array}
\]

**Stream $B$**

\[
\begin{array}{cccccc}
B^0 & \xleftarrow{\text{take}_0} & B^1 & \xleftarrow{\text{take}_1} & B^2 & \xleftarrow{\text{take}_2} & B^3 & \xleftarrow{\text{take}_3} & B^4 & \xleftarrow{\text{take}_4} & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccc}
\ast_{(0\ 2)} \text{Stream}(B) & B^0 & \xleftarrow{id} & B^0 & \xleftarrow{\text{take}_{0,1}} & B^2 & \xleftarrow{id} & B^2 & \xleftarrow{\text{take}_{2,3}} & B^4 \\
\end{array}
\]
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$[\star_p \tau](n) = [\tau](p(n))$$

For example:

![Diagram showing the structure of streams and warped streams](image)
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 $\omega$, and thus of $\omega^{op}$.
- Thus, if $X$ is a presheaf, so is $X \circ p$. In other words:

$$[\ast_p \tau](n) = [\tau](p(n))$$

For example:

$$\begin{array}{cccccccc}
0 & \leq & 1 & \leq & 2 & \leq & 3 & \leq & 4 & \leq & \ldots
\end{array}$$

Stream $B$

$$\begin{array}{cccccccc}
& B^0 & \leftarrow & B^1 & \leftarrow & B^2 & \leftarrow & B^3 & \leftarrow & B^4 & \leftarrow & \ldots
& \text{take}_0 & \text{take}_1 & \text{take}_2 & \text{take}_3 & \text{take}_4 & \ldots
\end{array}$$

$\ast(0, 2)$ Stream($B$)

$$\begin{array}{cccccccc}
& B^0 & \leftarrow & B^0 & \leftarrow & B^2 & \leftarrow & B^2 & \leftarrow & B^4 & \leftarrow & \ldots
& \text{id} & \text{take}_{0,1} & \text{id} & \text{take}_{2,3} & \text{id} & \ldots
\end{array}$$
Conjectural Results

Operational Semantics: Soundness and Totality
If $\Gamma \vdash e : \tau$ and $\sigma : \Gamma \odot n$, then there is $v$ s.t. $e; \sigma \Downarrow_n v$ and $v : \tau \odot n$.

Denotational Semantics: Adequacy
If $\llbracket \Gamma \vdash e : \tau \rrbracket = \llbracket \Gamma \vdash e' : \tau \rrbracket$ then $\Gamma \vdash e \cong_{ctx} e' : \tau$. 
Subtyping and Coherence

In `map`, we used

\[ f : \text{Int} \rightarrow \text{Int} \equiv \ast_{(1)} (\text{Int} \rightarrow \text{Int}) \ll : \ast_{0(1)} (\text{Int} \rightarrow \text{Int}) \]

In fact, the compiler did

\[ f : \text{Int} \rightarrow \text{Int} \equiv \ast_{(1)} (\text{Int} \rightarrow \text{Int}) \ll : \ast_{0 \cdot 2(1)} (\text{Int} \rightarrow \text{Int}) \]

\[ \equiv \ast_{0(1) \cdot 2(1)} (\text{Int} \rightarrow \text{Int}) \]

\[ \equiv \ast_{0(1)} \ast_{2(1)} (\text{Int} \rightarrow \text{Int}) \]

and then

\[ \ast_{2(1)} (\text{Int} \rightarrow \text{Int}) \ll : \ast_{(1)} (\text{Int} \rightarrow \text{Int}) \ll : \text{Int} \rightarrow \text{Int} \]

Coherence Issues

- Distinct explicit terms mean different things \textit{a priori}.
- Programmer writes implicit terms, compiler elaborates them into explicit ones. Is this reasonable?
Goals

- Define an algorithm $\Gamma \vdash t \leadsto e : \tau$ taking $(\Gamma, t)$ and returning $(e, \tau)$ such that $U(e) = t$ and $\Gamma \vdash e : \tau$.
- Make this choice canonical in a certain sense.
Goals

- Define an algorithm $\Gamma \vdash t \leadsto e : \tau$ taking $(\Gamma, t)$ and returning $(e, \tau)$ such that $U(e) = t$ 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 $\tau_1$ and $\tau_2$.
- Infer coercions $\gamma : \Gamma_1 <: \ast_p \Gamma_2$ given $\Gamma_1$ and $p$. 
Deciding Subtyping in Three Steps

1. The algorithmic judgment $\tau \gg \tau^s \sim \alpha \iff \alpha'$:

- implies $\alpha : \tau <: \tau^s$ and $\alpha' : \tau^s <: \tau$;
- implies that $\tau^s$ respects the following grammar.

$$
\begin{align*}
\tau^s &::= \ast_p \tau^r \mid \tau^s \times \tau^s \\
\tau^r &::= \nu \mid \text{Stream}(\tau^s) \mid \tau^s \rightarrow \tau^s
\end{align*}
$$
Deciding Subtyping in Three Steps

1. The algorithmic judgment \( \tau \gg \tau^s \sim \alpha \overset{\leftrightarrow}{\approx} \alpha' \):
   - implies \( \alpha : \tau <: \tau^s \) and \( \alpha' : \tau^s <: \tau \);
   - implies that \( \tau^s \) respects the following grammar.

   \[
   \tau^s ::= \ast p \tau^r \mid \tau^s \times \tau^s
   \]
   \[
   \tau^r ::= \nu \mid \text{Stream}(\tau^s) \mid \tau^s \rightarrow \tau^s
   \]

2. The algorithmic judgment \( \tau \geq \tau' \sim \alpha \):
   - implies \( \alpha : \tau <: \tau' \);
   - holds iff \( \tau \) is coercible to \( \tau' \) using only delays.
Deciding Subtyping in Three Steps

1. The algorithmic judgment $\tau \gg \tau^s \sim \alpha \leftrightarrow \alpha'$:
   - implies $\alpha : \tau <: \tau^s$ and $\alpha' : \tau_s <: \tau$;
   - implies that $\tau^s$ respects the following grammar.

   $\tau^s ::= \ast_p \tau^r \mid \tau^s \times \tau^s$

   $\tau^r ::= \nu \mid \text{Stream}(\tau^s) \mid \tau^s \rightarrow \tau^s$

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

3. Algorithmic subtyping $\tau_1 <: \tau_2 \sim \alpha$ can then be defined by

   $\begin{align*}
   \tau_1 \gg \tau_1^s \sim \alpha_1 &\leftrightarrow - \\ 
   \tau_2 \gg \tau_2^s \sim - &\leftrightarrow \alpha_3 \\ 
   \tau_1^s \geq \tau_2^s \sim \alpha_2 &
   \end{align*}$

   $\tau_1 <: \tau_2 \sim \alpha_1; \alpha_2; \alpha_3$
To type-check $e$ by $p$ in $\Gamma$, for any $\tau$ in $\Gamma$ we must find $\tau^p$ such that

$$\tau \triangleq \star_p \tau^p$$
To type-check $e$ by $p$ in $\Gamma$, for any $\tau$ in $\Gamma$ we must find $\tau^p$ such that

$$\tau \leq_* p \tau^p$$

Moreover, to be complete we need the best solution, i.e. for any $\tau'$

$$\tau \leq_* p \tau' \iff \tau^p \leq \tau'$$
To type-check $e$ by $p$ in $\Gamma$, for any $\tau$ in $\Gamma$ we must find $\tau^p$ such that

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

Moreover, to be complete we need the best solution, i.e. for any $\tau'$

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

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

$$\tau <: \ast_p \tau' \iff \tau/p <: \tau'$$
To type-check $e$ by $p$ in $\Gamma$, for any $\tau$ in $\Gamma$ we must find $\tau^p$ such that

$$\tau <: \star_p \tau^p$$

Moreover, to be complete we need the best solution, i.e. for any $\tau'$

$$\tau <: \star_p \tau' \iff \tau^p <: \tau'$$

Fortunately, there exists an operation $-\big{/}p$ on types such that

$$\tau <: \star_p \tau' \iff \tau/p <: \tau'$$

It reduces to a similar operation on warps.

$$q \geq p * r \iff q/p \geq r$$
We are looking for a Galois connection \((-/g) \sqsubset (- \circ g)\).

\[ h \circ g \leq f \iff h \leq f / g \]
We are looking for a Galois connection \((-/g) \dashv (- \circ g)\).

\[ h \circ g \leq f \iff h \leq f/g \]

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

\[(\text{Ran}_g(f))(n) = \bigsqcap f^*(g^*(\uparrow n))\]
We are looking for a Galois connection \((-/g) \dashv (- \circ g)\).

\[ h \circ g \leq f \iff h \leq f/g \]

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

\[(\text{Ran}_g(f))(n) = \bigsqcap f^*(g^*(\uparrow n))\]

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

\[
\begin{align*}
(1)/0(1) &= 2(1) & (1)/(2) &= (10) & (10)/(10) &= (1) \\
(1)/(0) &= (\omega) & (1)/(\omega) &= 1(0)
\end{align*}
\]
Coherence of Subtyping

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

$$\left[ \alpha : \tau_1 <: \tau_2 \right] = \left[ \alpha' : \tau_1 <: \tau_2 \right]$$
Main Results

Coherence of Subtyping
If $\alpha : \tau_1 <: \tau_2$ and $\alpha' : \tau_1 <: \tau_2$ then

$$\llbracket \alpha : \tau_1 <: \tau_2 \rrbracket = \llbracket \alpha' : \tau_1 <: \tau_2 \rrbracket$$

Completeness of Type-Checking
For any $\Gamma \vdash e : \tau$, there is $e_m, \tau_m, \alpha$ such that

$$\Gamma \vdash U(e) \sim e_m : \tau_m \quad \Gamma \vdash e_m : \tau_m \quad \alpha : \tau_m <: \tau$$

$$\llbracket \Gamma \vdash (e_m; \alpha) : \tau \rrbracket = \llbracket \Gamma \vdash e : \tau \rrbracket$$
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 $e_m, \tau_m, \alpha$ such that

$$\Gamma \vdash U(e) \sim e_m : \tau_m \quad \Gamma \vdash e_m : \tau_m \quad \alpha : \tau_m <: \tau$$

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

Corollary: Coherence

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

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

\[ t \mapsto p \rightarrow q \mapsto X \mapsto Y \mapsto \ast_{p \ast q} X \mapsto \ast_{p \ast q} Y \]

\[ \text{decat} \]

\[ \text{concat} \]

\[ t \mapsto p \rightarrow q \mapsto X \mapsto Y \mapsto \ast_{p \ast q} X \mapsto \ast_{p \ast q} Y \]
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!