How to be a Productive Programmer

by putting things off until tomorrow

Robert Atkey
University of Strathclyde, Glasgow, UK

11th November 2011

Playing in Streams

```
data Stream = StreamCons Integer Stream

ones :: Stream
ones = StreamCons 1 ones

map :: (Integer → Integer) → Stream → Stream
map f (StreamCons x xs) = StreamCons (f x) (map f xs)

merge :: Stream → Stream → Stream
merge (StreamCons x xs) (StreamCons y ys) =
    StreamCons x (StreamCons y (merge xs ys))
```

Playing in Streams

```
co data Stream = StreamCons Integer Stream

ones :: Stream
ones = StreamCons 1 ones

map :: (Integer → Integer) → Stream → Stream
map f (StreamCons x xs) = StreamCons (f x) (map f xs)

merge :: Stream → Stream → Stream
merge (StreamCons x xs) (StreamCons y ys) =
      StreamCons x (StreamCons y (merge xs ys))
```

Higher-order Functions on Streams

```
mergef ::
    (Integer → Integer → Stream → Stream) →
    Stream →
    Stream →
    Stream
mergef f (StreamCons x xs) (StreamCons y ys) =
    f x y (mergef f xs ys)
```

Higher-order Functions on Streams

```
mergef ::
    (Integer → Integer → Stream → Stream) →
    Stream →
    Stream →
    Stream
mergef f (StreamCons x xs) (StreamCons y ys) =
    f x y (mergef f xs ys)
```

A bad choice of argument yields non-productive definitions:

```
badf :: Integer → Integer → Stream → Stream
badf x y s = s
```

Higher-order Functions on Streams

```
mergef ::
    (Integer → Integer → Stream → Stream) →
    Stream →
    Stream →
    Stream
mergef f (StreamCons x xs) (StreamCons y ys) =
    f x y (mergef f xs ys)
```

Coq reports:

Sub-expression “f x y (mergef f xs ys)” *not in guarded form*

Higher-order Functions on Streams

```
mergef ::
    (Integer → Integer → Stream → Stream) →
    Stream →
    Stream →
    Stream
mergef f (StreamCons x xs) (StreamCons y ys) =
    f x y (mergef f xs ys)
```

Get around guardedness check by changing the type of f?

```
f :: Integer → Integer → Integer
f :: Integer → Integer → (Integer, [Integer])
```

but what about:

```
f x y s = StreamCons x (map (+1) s)
```

How can we put guardedness constraints into the types?

Putting things off until tomorrow

Putting things off until tomorrow

(Nakano, 2000)

(McBride, 2009)

Type-based Guardedness

$$\mathrm{f} :: \mathit{Integer} \to \mathit{Integer} \to \quad \mathit{Stream} \to \mathit{Stream}$$

Type-based Guardedness

$$\mathsf{f} :: \mathit{Integer} \to \mathit{Integer} \to \triangleright\, \mathit{Stream} \to \mathit{Stream}$$

Type-based Guardedness

$\triangleright \mathit{Stream}$ — "a stream tomorrow"

$$\mathsf{StreamCons} :: \mathit{Integer} \to \triangleright \mathit{Stream} \to \mathit{Stream}$$
$$\mathsf{deStreamCons} :: \mathit{Stream} \to (\mathit{Integer}, \triangleright \mathit{Stream})$$

$$\mathsf{fix} :: (\triangleright A \to A) \to A$$

Type-based Guardedness

$\triangleright \mathit{Stream}$ — "a stream tomorrow"

$$\mathsf{StreamCons} :: \mathit{Integer} \to \triangleright \mathit{Stream} \to \mathit{Stream}$$
$$\mathsf{deStreamCons} :: \mathit{Stream} \to (\mathit{Integer}, \triangleright \mathit{Stream})$$

$$\mathsf{fix} :: (\triangleright A \to A) \to A$$

$$\mathsf{ones} = \mathrm{fix}\ (\lambda s.\ \mathsf{StreamCons}\ 1\ \mathrm{s})$$

$$\mathrm{fix}\ (\lambda \mathrm{x}.\ \mathrm{x})$$

Type-based Guardedness

$\triangleright \mathit{Stream}$ — "a stream tomorrow"

$$\mathsf{StreamCons} :: \mathit{Integer} \to \triangleright \mathit{Stream} \to \mathit{Stream}$$
$$\mathsf{deStreamCons} :: \mathit{Stream} \to (\mathit{Integer}, \triangleright \mathit{Stream})$$

$$\mathsf{fix} :: (\triangleright A \to A) \to A$$

$$\mathsf{ones} = \mathrm{fix}\ (\lambda s.\ \mathsf{StreamCons}\ 1\ \mathrm{s})$$

~~$\mathrm{fix}\ (\lambda \mathrm{x}.\ \mathrm{x})$~~

Applicative Functor

$$\text{pure} :: A \to \triangleright A$$
$$(\circledast) :: \triangleright(A \to B) \to \triangleright A \to \triangleright B$$

Applicative Functor

$$\text{pure} :: A \to \triangleright A$$
$$(\circledast) :: \triangleright(A \to B) \to \triangleright A \to \triangleright B$$

```
mergef ::
      (Integer → Integer → ▷Stream → Stream) →
      Stream →
      Stream →
      Stream
mergef f = fix (λg xs ys.
                     let (x, xs') = deStreamCons xs
                         (y, ys') = deStreamCons ys
                     in f x y (g ⊛ xs' ⊛ ys'))
```

$$g :: \triangleright(\mathit{Stream} \to \mathit{Stream} \to \mathit{Stream})$$

From Infinite to Finite

Can we write

$$\text{take} :: \mathit{Natural} \to \mathit{Stream} \to [\mathit{Integer}]$$

by structural recursion on the first argument?

From Infinite to Finite

Can we write

$$\mathsf{take} :: \mathit{Natural} \to \mathit{Stream} \to [\mathit{Integer}]$$

by structural recursion on the first argument?

```
take 0 s = []
take (n + 1) s = x : take n s'
  where (x, s') = deStreamCons s
```

From Infinite to Finite

Can we write

$$\mathsf{take} :: \mathit{Natural} \to \mathit{Stream} \to [\mathit{Integer}]$$

by structural recursion on the first argument?

$$\begin{aligned}
&\mathsf{take}\ 0\ \mathrm{s} = [] \\
&\mathsf{take}\ (\mathrm{n}+1)\ \mathrm{s} = \mathrm{x} : \mathsf{take}\ \mathrm{n}\ \mathrm{s'} \\
&\quad \textbf{where}\ (\mathrm{x}, \mathrm{s'}) = \mathsf{deStreamCons}\ \mathrm{s}
\end{aligned}$$

This type prevents us:

$$\mathsf{deStreamCons} :: \mathit{Stream} \to (\mathit{Integer}, \triangleright \mathit{Stream})$$

We cannot leave the “time stream” of the stream.

Clock Variables

Clock Variables

Annotate with “clock variables”, κ:

- A clock κ represents a fixed amount of time remaining
- $Stream^{\kappa}$ is a stream for at least the time remaining in κ
- $\triangleright^{\kappa}$ removes one unit of time remaining in κ

Clock Variables

Annotate with "clock variables", κ:

- A clock κ represents a fixed amount of time remaining
- Stream^{κ} is a stream for at least the time remaining in κ
- $\triangleright^{\kappa}$ removes one unit of time remaining in κ

$$\mathsf{StreamCons}^{\kappa} :: \mathit{Integer} \to \triangleright^{\kappa}\mathit{Stream}^{\kappa} \to \mathit{Stream}^{\kappa}$$
$$\mathsf{deStreamCons}^{\kappa} :: \mathit{Stream}^{\kappa} \to (\mathit{Integer}, \triangleright^{\kappa}\mathit{Stream}^{\kappa})$$

Clock Variables

Annotate with “clock variables”, κ:

- A clock κ represents a fixed amount of time remaining
- Stream^{κ} is a stream for at least the time remaining in κ
- $\triangleright^{\kappa}$ removes one unit of time remaining in κ

$$\mathsf{StreamCons}^{\kappa} :: \mathit{Integer} \to \triangleright^{\kappa}\mathit{Stream}^{\kappa} \to \mathit{Stream}^{\kappa}$$
$$\mathsf{deStreamCons}^{\kappa} :: \mathit{Stream}^{\kappa} \to (\mathit{Integer}, \triangleright^{\kappa}\mathit{Stream}^{\kappa})$$

$$\mathsf{delay}^{\kappa} :: A \to \triangleright^{\kappa} A$$
$$(\circledast^{\kappa}) :: \triangleright^{\kappa}(A \to B) \to \triangleright^{\kappa} A \to \triangleright^{\kappa} B$$

Clock Variables

Annotate with "clock variables", κ:

- A clock κ represents a fixed amount of time remaining
- Stream^{κ} is a stream for at least the time remaining in κ
- $\triangleright^{\kappa}$ removes one unit of time remaining in κ

$$\mathsf{StreamCons}^{\kappa} :: \mathit{Integer} \to \triangleright^{\kappa}\mathit{Stream}^{\kappa} \to \mathit{Stream}^{\kappa}$$
$$\mathsf{deStreamCons}^{\kappa} :: \mathit{Stream}^{\kappa} \to (\mathit{Integer}, \triangleright^{\kappa}\mathit{Stream}^{\kappa})$$

$$\mathsf{delay}^{\kappa} :: A \to \triangleright^{\kappa} A$$
$$(\circledast^{\kappa}) :: \triangleright^{\kappa}(A \to B) \to \triangleright^{\kappa} A \to \triangleright^{\kappa} B$$

$$\mathsf{fix}^{\kappa} :: (\triangleright^{\kappa} A \to A) \to A$$

Eternity in an Hour

For type A with a free clock variable κ:

- Form the type $\forall\kappa.\ A$
- A with all possible amounts of “time remaining”

Eternity in an Hour

For type A with a free clock variable κ:

- Form the type $\forall\kappa.\ A$
- A with all possible amounts of "time remaining"

$(\forall\kappa.\ \mathit{Stream}^{\kappa})$ $\approx$ streams that go on forever

Eternity in an Hour

For type A with a free clock variable κ:

- Form the type $\forall\kappa.\ A$
- A with all possible amounts of "time remaining"

$(\forall\kappa.\ \mathit{Stream}^{\kappa})$ $\approx$ streams that go on forever

Quantification allows removal of delays:

$$\mathsf{force} :: (\forall\kappa.\ \triangleright^{\kappa} A) \to (\forall\kappa.\ A)$$

From Infinite to Finite, again

Can we write

$$\mathsf{take} :: \mathit{Natural} \to (\forall \kappa.\ \mathit{Stream}^{\kappa}) \to [\mathit{Integer}]$$

by structural recursion on the first argument?

From Infinite to Finite, again

Can we write

$$\mathsf{take} :: \mathit{Natural} \to (\forall \kappa.\ \mathit{Stream}^{\kappa}) \to [\mathit{Integer}]$$

by structural recursion on the first argument?

```
take 0 s = []
take (n + 1) s = x : take n (force s')
  where (x, s') = deStreamCons s
```

Derivation

Let $\Gamma = \mathsf{s} : \forall \kappa.\ \mathit{Stream}^{\kappa}$

$$\dfrac{\dfrac{\dfrac{\dfrac{\kappa; \Gamma \vdash \mathsf{s} : \forall \kappa.\ \mathit{Stream}^{\kappa}}{\kappa; \Gamma \vdash \mathsf{s} : \mathit{Stream}^{\kappa}}}{\kappa; \Gamma \vdash \mathsf{deStreamCons}\ \mathsf{s} : (\mathit{Integer}, \triangleright^{\kappa} \mathit{Stream}^{\kappa})}}{; \Gamma \vdash \mathsf{deStreamCons}\ \mathsf{s} : \forall \kappa.\ (\mathit{Integer}, \triangleright^{\kappa} \mathit{Stream}^{\kappa})}}{; \Gamma \vdash \mathsf{deStreamCons}\ \mathsf{s} : (\mathit{Integer}, \forall \kappa.\ \triangleright^{\kappa} \mathit{Stream}^{\kappa})}$$

Replace-Min

A classic example (Bird, 1984)

```
replaceMin :: Tree → Tree
replaceMin t =
      let (t', m) = replaceMinBody (t, m) in t'
    where
      replaceMinBody (Leaf x, m) = (Leaf m, x)
      replaceMinBody (Br l r, m) =
                  let (l', m_l) = replaceMinBody l m
                      (r', m_r) = replaceMinBody r m
                  in (Br l' r', min m_l m_r)
```

With Clocks

```
replaceMinBody :: (Tree, Integer) → (Tree, Integer)
replaceMinBody (Leaf x, m) = (Leaf m, x)
replaceMinBody (Br l r, m) =
        let (l', m_l) = replaceMinBody l m
            (r', m_r) = replaceMinBody r m
        in (Br l' r', min m_l m_r)
```

With Clocks

```
replaceMinBody :: ∀κ. (Tree,      Integer) → (     Tree, Integer)
replaceMinBody (Leaf x, m) = (         Leaf     m, x)
replaceMinBody (Br l r, m) =
                  let (l', m_l) = replaceMinBody l m
                      (r', m_r) = replaceMinBody r m
                  in (         Br     l'     r', min m_l m_r)
```

With Clocks

$$
\begin{aligned}
&\mathsf{replaceMinBody} :: \forall\kappa.\ (\mathit{Tree},\ \triangleright^{\kappa}\ \mathit{Integer}) \to (\quad \mathit{Tree},\ \mathit{Integer})\\
&\mathsf{replaceMinBody}\ (\mathsf{Leaf}\ \mathrm{x},\ \mathrm{m}) = (\quad\ \mathsf{Leaf}\quad\ \mathrm{m}, \mathrm{x})\\
&\mathsf{replaceMinBody}\ (\mathsf{Br}\ \mathrm{l}\ \mathrm{r},\ \mathrm{m}) =\\
&\qquad\qquad \mathbf{let}\ (\mathrm{l'}, \mathrm{m}_l) = \mathsf{replaceMinBody}\ \mathrm{l}\ \mathrm{m}\\
&\qquad\qquad\quad\ (\mathrm{r'}, \mathrm{m}_r) = \mathsf{replaceMinBody}\ \mathrm{r}\ \mathrm{m}\\
&\qquad\qquad \mathbf{in}\ (\quad\ \mathsf{Br}\quad\ \mathrm{l'}\quad\ \mathrm{r'},\ \mathrm{min}\ \mathrm{m}_l\ \mathrm{m}_r)
\end{aligned}
$$

With Clocks

$$
\begin{aligned}
&\mathsf{replaceMinBody} :: \forall\kappa.\ (\mathit{Tree},\ \triangleright^{\kappa}\ \mathit{Integer}) \to (\ \triangleright^{\kappa}\ \mathit{Tree},\ \mathit{Integer})\\
&\mathsf{replaceMinBody}\ (\mathsf{Leaf}\ \mathrm{x},\ \mathrm{m}) = (\quad \mathsf{Leaf}\quad \mathrm{m}, \mathrm{x})\\
&\mathsf{replaceMinBody}\ (\mathsf{Br}\ \mathrm{l}\ \mathrm{r},\ \mathrm{m}) =\\
&\qquad \textbf{let}\ (\mathrm{l'}, \mathrm{m}_l) = \mathsf{replaceMinBody}\ \mathrm{l}\ \mathrm{m}\\
&\qquad\quad (\mathrm{r'}, \mathrm{m}_r) = \mathsf{replaceMinBody}\ \mathrm{r}\ \mathrm{m}\\
&\qquad \textbf{in}\ (\quad \mathsf{Br}\quad \mathrm{l'}\quad \mathrm{r'},\ \mathsf{min}\ \mathrm{m}_l\ \mathrm{m}_r)
\end{aligned}
$$

With Clocks

```
replaceMinBody :: ∀κ. (Tree, ▷κ Integer) → (▷κ Tree, Integer)
replaceMinBody (Leaf x, m) = (delay Leaf ⊛ m, x)
replaceMinBody (Br l r, m) =
                  let (l', m_l) = replaceMinBody l m
                      (r', m_r) = replaceMinBody r m
                  in (        Br     l'     r', min m_l m_r)
```

With Clocks

```
replaceMinBody :: ∀κ. (Tree, ▷κ Integer) → (▷κ Tree, Integer)
replaceMinBody (Leaf x, m) = (delay Leaf ⊛ m, x)
replaceMinBody (Br l r, m) =
                  let (l', m_l) = replaceMinBody l m
                      (r', m_r) = replaceMinBody r m
                  in (delay Br ⊛ l' ⊛ r', min m_l m_r)
```

With Clocks

$$
\begin{aligned}
&\mathsf{replaceMinBody} :: \forall \kappa.\ (\mathit{Tree},\ \triangleright^{\kappa}\ \mathit{Integer}) \to (\ \triangleright^{\kappa}\ \mathit{Tree},\ \mathit{Integer})\\
&\mathsf{replaceMinBody}\ (\mathsf{Leaf}\ \mathrm{x},\ \mathrm{m}) = (\ \mathsf{delay}\ \mathsf{Leaf}\ \circledast\ \mathrm{m}, \mathrm{x})\\
&\mathsf{replaceMinBody}\ (\mathsf{Br}\ \mathrm{l}\ \mathrm{r},\ \mathrm{m}) =\\
&\qquad\qquad \textbf{let}\ (\mathrm{l'}, \mathrm{m}_l) = \mathrm{replaceMinBody\ l\ m}\\
&\qquad\qquad\quad\ (\mathrm{r'}, \mathrm{m}_r) = \mathrm{replaceMinBody\ r\ m}\\
&\qquad\qquad \textbf{in}\ (\ \mathsf{delay}\ \mathsf{Br}\ \circledast\ \mathrm{l'}\ \circledast\ \mathrm{r'},\ \mathrm{min}\ \mathrm{m}_l\ \mathrm{m}_r)
\end{aligned}
$$

$$
\mathsf{trace} :: \forall \kappa.\ ((A[\kappa], \triangleright^{\kappa} U) \to (B[\kappa], U)) \to A[\kappa] \to B[\kappa]
$$

With Clocks

$$
\begin{aligned}
&\mathsf{replaceMinBody} :: \forall\kappa.\ (\mathit{Tree},\ \triangleright^{\kappa}\ \mathit{Integer}) \to (\ \triangleright^{\kappa}\ \mathit{Tree},\ \mathit{Integer})\\
&\mathsf{replaceMinBody}\ (\mathsf{Leaf}\ \mathrm{x},\ \mathrm{m}) = (\ \mathsf{delay}\ \mathsf{Leaf}\ \circledast\ \mathrm{m}, \mathrm{x})\\
&\mathsf{replaceMinBody}\ (\mathsf{Br}\ \mathrm{l}\ \mathrm{r},\ \mathrm{m}) =\\
&\qquad\qquad \mathbf{let}\ (\mathrm{l'}, \mathrm{m}_l) = \mathsf{replaceMinBody}\ \mathrm{l}\ \mathrm{m}\\
&\qquad\qquad\quad\ (\mathrm{r'}, \mathrm{m}_r) = \mathsf{replaceMinBody}\ \mathrm{r}\ \mathrm{m}\\
&\qquad\qquad \mathbf{in}\ (\ \mathsf{delay}\ \mathsf{Br}\ \circledast\ \mathrm{l'}\ \circledast\ \mathrm{r'},\ \mathsf{min}\ \mathrm{m}_l\ \mathrm{m}_r)
\end{aligned}
$$

$$
\mathsf{trace} :: \forall\kappa.\ ((A[\kappa], \triangleright^{\kappa} U) \to (B[\kappa], U)) \to A[\kappa] \to B[\kappa]
$$

$$
\begin{aligned}
&\mathsf{replaceMin} :: \mathit{Tree} \to \mathit{Tree}\\
&\mathsf{replaceMin}\ \mathrm{t} = \mathsf{force}\ (\mathsf{trace}\ \mathsf{replaceMinBody}\ \mathrm{t})
\end{aligned}
$$

Related Work

Related Work

A Modality for recursion
(Nakano, 2000)

Step-indexed Logical Relations
(Appel, McAllester, 2001)
(Dreyer, Ahmed, Birkedal, 2011)

Time Flies like an Applicative Functor
(McBride, 2009)

First steps in synthetic guarded domain theory: step-indexing in the topos of trees
(Birkedal, Møgelberg, Støvring, Schwinghammer, 2011)

Ultrametric Semantics of Reactive Programs
(Krishnaswami, Benton, 2011)

The Logic of Provability
(Boolos, 1993)

▸ Short of time?

Formal Calculus

Types

Well formed types: $\Delta \vdash A$ where $\Delta = \kappa_1, \ldots, \kappa_n$

$$\frac{}{\Delta \vdash 1} \qquad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \to B} \qquad \frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A \times B}$$

$$\frac{\Delta \vdash A \qquad \Delta \vdash B}{\Delta \vdash A + B} \qquad \frac{\kappa \in \Delta \qquad \Delta \vdash A}{\Delta \vdash \triangleright^{\kappa} A} \qquad \frac{\Delta, \kappa \vdash A}{\Delta \vdash \forall \kappa . A}$$

Type Equalities

$A \equiv B$ is

- reflexive, symmetric, transitive
- a congruence

and satisfies:

$$
\begin{array}{rcll}
\forall \kappa.A & \equiv & A & (\kappa \notin \mathit{fv}(A)) \\
\forall \kappa.A + B & \equiv & (\forall \kappa.A) + (\forall \kappa.B) & \\
\forall \kappa.A \times B & \equiv & (\forall \kappa.A) \times (\forall \kappa.B) & \\
\forall \kappa.A \to B & \equiv & A \to \forall \kappa.B & (\kappa \notin \mathit{fv}(A)) \\
\forall \kappa.\forall \kappa'.A & \equiv & \forall \kappa'.\forall \kappa.A &
\end{array}
$$

Typing Rules

Well-typed terms: $\Delta; \Gamma \vdash e : A$ where $\Gamma = x_1 : A_1, \ldots, x_n : A_n$

- The simply-typed λ-calculus with products and sums

$$\frac{\Delta; \Gamma \vdash e : A \qquad \kappa \in \Delta}{\Delta; \Gamma \vdash \mathsf{delay}\ e : \triangleright^{\kappa} A} \qquad \frac{\Delta; \Gamma \vdash f : \triangleright^{\kappa}(A \to B) \qquad \Delta; \Gamma \vdash e : \triangleright^{\kappa} A}{\Delta; \Gamma \vdash f \circledast e : \triangleright^{\kappa} B}$$

Typing Rules

$$\frac{\Delta, \kappa; \Gamma \vdash e : A \qquad \kappa \notin fv(\Gamma)}{\Delta; \Gamma \vdash e : \forall \kappa . A} \qquad \frac{\Delta; \Gamma \vdash e : \forall \kappa . A \qquad \kappa' \in \Delta}{\Delta; \Gamma \vdash e : A[\kappa \mapsto \kappa']}$$

$$\frac{\Delta; \Gamma \vdash e : A \qquad \Delta \vdash A \equiv B}{\Delta; \Gamma \vdash e : B}$$

$$\frac{\Delta; \Gamma \vdash f : \triangleright^{\kappa} A \to A}{\Delta; \Gamma \vdash \mathsf{fix}\, f : A} \qquad \frac{\Delta; \Gamma \vdash e : \forall \kappa . \triangleright^{\kappa} A}{\Delta; \Gamma \vdash \mathsf{force}\, e : \forall \kappa . A}$$

Data types

Descriptions of data types: $F, G ::= A \mid F \times G \mid F + G \mid -$

$$\overline{A}X = A \qquad \overline{F \times G}X = \overline{F}X \times \overline{G}X \qquad \overline{F + G}X = \overline{F}X + \overline{G}X \qquad \overline{-}X = X$$

Data types

Descriptions of data types: $F, G ::= A \mid F \times G \mid F + G \mid -$

$$\overline{A}X = A \qquad \overline{F \times G}X = \overline{F}X \times \overline{G}X \qquad \overline{F + G}X = \overline{F}X + \overline{G}X \qquad \overline{-}X = X$$

$$\frac{}{\Delta \vdash \mu F} \qquad \frac{\kappa \in \Delta}{\Delta \vdash \nu^{\kappa} F} \qquad \nu F \stackrel{def}{=} \forall \kappa . \nu^{\kappa} F$$

Data types

Descriptions of data types: $F, G ::= A \mid F \times G \mid F + G \mid -$

$$\overline{A}X = A \qquad \overline{F \times G}X = \overline{F}X \times \overline{G}X \qquad \overline{F + G}X = \overline{F}X + \overline{G}X \qquad \overline{-}X = X$$

$$\frac{}{\Delta \vdash \mu F} \qquad \frac{\kappa \in \Delta}{\Delta \vdash \nu^{\kappa} F} \qquad \nu F \stackrel{def}{=} \forall \kappa . \nu^{\kappa} F$$

$\mathsf{cons}_{\mu} : \overline{F}(\mu F) \to \mu F$
$\mathsf{fold}_{\mu} : (\overline{F}A \to A) \to \mu F \to A$

$\mathsf{cons}_{\nu} : \overline{F}(\triangleright^{\kappa} \nu^{\kappa} F) \to \nu^{\kappa} F$
$\mathsf{deCons}_{\nu} : \nu^{\kappa} F \to \overline{F}(\triangleright^{\kappa} \nu^{\kappa} F)$

Semantics

Untyped Semantics

$$\textbf{data}\ D = \mathsf{Lam}\ (D \to D) \mid \mathsf{Inl}\ D \mid \mathsf{Inr}\ D \mid \mathsf{Pair}\ D\ D \mid \mathsf{Unit}$$

$$\begin{aligned}
[\![-]\!] &:: \mathit{Term} \to (\mathit{Var} \to D) \to D \\
[\![\mathrm{x}]\!]\ \eta &= \eta(\mathrm{x}) \\
[\![\lambda \mathrm{x}.\ e]\!]\ \eta &= \mathsf{Lam}\ (\lambda v.\ [\![e]\!](\eta[\mathrm{x} \mapsto v])) \\
[\![f\ e]\!]\ \eta &= d_f\ ([\![e]\!]\eta)\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta
\end{aligned}$$

Untyped Semantics

$$\textbf{data}\ D = \mathsf{Lam}\ (D \to D) \mid \mathsf{Inl}\ D \mid \mathsf{Inr}\ D \mid \mathsf{Pair}\ D\ D \mid \mathsf{Unit}$$

$$\begin{aligned}
[\![-]\!] &:: \mathit{Term} \to (\mathit{Var} \to D) \to D \\
[\![\mathrm{x}]\!]\ \eta &= \eta(\mathrm{x}) \\
[\![\lambda \mathrm{x}.\ e]\!]\ \eta &= \mathsf{Lam}\ (\lambda v.\ [\![e]\!](\eta[\mathrm{x} \mapsto v])) \\
[\![f\ e]\!]\ \eta &= d_f\ ([\![e]\!]\eta)\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta
\end{aligned}$$

$$\begin{aligned}
[\![\mathrm{delay}\ e]\!]\ \eta &= [\![e]\!]\eta \\
[\![f \circledast e]\!]\ \eta &= d_f\ ([\![e]\!]\eta)\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta \\
[\![\mathrm{force}\ e]\!]\ \eta &= [\![e]\!]\eta
\end{aligned}$$

Untyped Semantics

$$\textbf{data}\ D = \mathsf{Lam}\ (D \to D) \mid \mathsf{Inl}\ D \mid \mathsf{Inr}\ D \mid \mathsf{Pair}\ D\ D \mid \mathsf{Unit}$$

$$\begin{aligned}
[\![-]\!] &:: \mathit{Term} \to (\mathit{Var} \to D) \to D \\
[\![\mathrm{x}]\!]\ \eta &= \eta(\mathrm{x}) \\
[\![\lambda \mathrm{x}.\ e]\!]\ \eta &= \mathsf{Lam}\ (\lambda v.\ [\![e]\!](\eta[\mathrm{x} \mapsto v])) \\
[\![f\ e]\!]\ \eta &= d_f\ ([\![e]\!]\eta)\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta
\end{aligned}$$

$$\begin{aligned}
[\![\mathrm{delay}\ e]\!]\ \eta &= [\![e]\!]\eta \\
[\![f \circledast e]\!]\ \eta &= d_f\ ([\![e]\!]\eta)\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta \\
[\![\mathrm{force}\ e]\!]\ \eta &= [\![e]\!]\eta
\end{aligned}$$

$$[\![\mathrm{fix}\ f]\!]\ \eta = \mathit{fix}\ d_f\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta$$

Untyped Semantics

$$\textbf{data}\ D = \mathsf{Lam}\ (D \to D) \mid \mathsf{Inl}\ D \mid \mathsf{Inr}\ D \mid \mathsf{Pair}\ D\ D \mid \mathsf{Unit}$$

$$\begin{aligned}
[\![-]\!] &:: \mathit{Term} \to (\mathit{Var} \to D) \to D \\
[\![\mathrm{x}]\!]\ \eta &= \eta(\mathrm{x}) \\
[\![\lambda \mathrm{x}.\ e]\!]\ \eta &= \mathsf{Lam}\ (\lambda v.\ [\![e]\!](\eta[\mathrm{x} \mapsto v])) \\
[\![f\ e]\!]\ \eta &= d_f\ ([\![e]\!]\eta)\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta
\end{aligned}$$

$$\begin{aligned}
[\![\mathrm{delay}\ e]\!]\ \eta &= [\![e]\!]\eta \\
[\![f \circledast e]\!]\ \eta &= d_f\ ([\![e]\!]\eta)\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta \\
[\![\mathrm{force}\ e]\!]\ \eta &= [\![e]\!]\eta
\end{aligned}$$

$$[\![\mathrm{fix}\ f]\!]\ \eta = \mathit{fix}\ d_f\ \textbf{where}\ \mathsf{Lam}\ d_f = [\![f]\!]\eta$$

(really in $\mathrm{DCPO}_\perp$)

Semantics of Types

A type $\Delta \vdash A$ is interpreted as a collection of subsets of D

$$[\![\Delta \vdash A]\!] : \mathbb{N}^{|\Delta|} \to \mathcal{P}(D)$$

The interpretation of $\Delta \vdash A$ is parameterised by Δ-many clocks

Satisfying:

- $(n_1, ..., n_{|\Delta|}) \leq (n'_1, ..., n'_{|\Delta|})$ implies
  $$[\![\Delta \vdash A]\!](n'_1, ..., n'_\Delta) \subseteq [\![\Delta \vdash A]\!](n_1, ..., n_\Delta)$$
- If $x \in [\![\Delta \vdash A]\!]\delta$ and $x \sqsubseteq x'$ then $x' \in [\![\Delta \vdash A]\!]\delta$.

Semantics of Types

$\delta = (n_1, ..., n_{|\Delta|})$ and also: $[\![F]\!] : \mathcal{P}(D) \to \mathcal{P}(D)$

$$
\begin{aligned}
[\![1]\!]\delta &= \{\mathsf{Unit}\} \\
[\![A \to B]\!]\delta &= \{\mathsf{Lam}\, f \mid \forall \delta' \sqsubseteq \delta, x \in [\![A]\!]\delta'.\, fx \in [\![B]\!]\delta'\} \\
[\![A \times B]\!]\delta &= \{\mathsf{Pair}(x,y) \mid x \in [\![A]\!]\delta \wedge y \in [\![B]\!]\delta\} \\
[\![A + B]\!]\delta &= \{\mathsf{Inl}(x) \mid x \in [\![A]\!]\delta\} \cup \{\mathsf{Inr}(y) \mid y \in [\![B]\!]\delta\} \\
[\![\triangleright^\kappa A]\!]\delta &= \begin{cases} D & \text{if } \delta(\kappa) = 0 \\ [\![A]\!](\delta[\kappa \mapsto n]) & \text{if } \delta(\kappa) = n+1 \end{cases} \\
[\![\forall \kappa.A]\!]\delta &= \bigcap_{n \in \mathbb{N}} [\![A]\!](\delta[\kappa \mapsto n]) \\
[\![\nu^\kappa F]\!]\delta &= [\![F]\!]^{\delta(\kappa)}(D) \\
[\![\mu F]\!]\delta &= \bigcup_{n \in \mathbb{N}} [\![F]\!]^n(\emptyset)
\end{aligned}
$$

Type Safety

Theorem
If $- ; - \vdash e : A$, *then for all* $\eta \in Var \to D$,

$$[\![e]\!]\eta \in [\![\Delta \vdash A]\!]()$$

This means:

- $[\![e]\!]\eta \neq \bot$
- If $A = \forall\kappa.\nu^{\kappa}(X \times -)$, then e generates infinitely many Xs

Slogan

Well typed programs ramble on forever
and never get to the $\perp$ of things

Alternative Ending

Type Safety

Define a semantics where:

- delay, $\circledast$, force are no-ops, and
  fix is interpreted by domain-theoretic fixpoint

If $-;- \vdash e : A$, then

- $[\![e]\!]\eta \neq \bot$
- If $A = \forall\kappa.\nu^{\kappa}(X \times -)$, then e generates infinitely many Xs

Slogan

Well typed programs ramble on forever
and never get to the $\perp$ of things