Ultrametric Semantics of Reactive Programs

Neelakantan R. Krishnaswami <neelk@microsoft.com>
Nick Benton <nick@microsoft.com>

LICS 2011

Functional Reactive Programming in a Nutshell

- Goal: Write interactive programs in a pure style
- Idea: Mutable state of type X becomes *stream* of values $S(X)$ [Eliot & Hudak 1997]

Functional Reactive Programming in a Nutshell

- Goal: Write interactive programs in a pure style
- Idea: Mutable state of type X becomes *stream* of values $S(X)$ [Eliot & Hudak 1997]
- Interactive program has type $S(\mathrm{In}) \to S(\mathrm{Out})$

Trouble in Paradise

```
profit :: S(stockprice) → S(order)
profit prices =
  if today < tomorrow
  then cons(Buy,  profit (tl prices))
  else cons(Sell, profit (tl prices))
where
  today = hd prices
  tomorrow = hd (tl prices)
```

Trouble in Paradise

```
profit :: S(stockprice) → S(order)
profit prices =
  if today < tomorrow
  then cons(Buy,  profit (tl prices))
  else cons(Sell, profit (tl prices))
where
  today = hd prices
  tomorrow = hd (tl prices)
```

Causal Stream Functions

A function $f : S(A) \to S(B)$ is *causal*, when for all n, as, as':

$$\lfloor as \rfloor_n = \lfloor as' \rfloor_n \implies \lfloor f\ as \rfloor_n = \lfloor f as' \rfloor_n$$

Causal Stream Functions

A function $f : S(A) \to S(B)$ is *causal*, when for all n, as, as':

$$\lfloor as \rfloor_n = \lfloor as' \rfloor_n \implies \lfloor f\ as \rfloor_n = \lfloor f as' \rfloor_n$$

- First n outputs of f depend only on first n inputs

Causal Stream Functions

A function $f : S(A) \to S(B)$ is *causal*, when for all n, as, as':

$$\lfloor as \rfloor_n = \lfloor as' \rfloor_n \implies \lfloor f\ as \rfloor_n = \lfloor fas' \rfloor_n$$

- First n outputs of f depend only on first n inputs
- `tail` not causal: element n of `tail` xs = element $n + 1$ of xs

Causal Stream Functions

A function $f : S(A) \to S(B)$ is *causal*, when for all n, as, as':

$$\lfloor as \rfloor_n = \lfloor as' \rfloor_n \implies \lfloor f\ as \rfloor_n = \lfloor f as' \rfloor_n$$

- First n outputs of f depend only on first n inputs
- `tail` not causal: element n of `tail` xs = element $n + 1$ of xs
- But what about higher-order?

The Category of Ultrametric Spaces

A pair $(X, d : X \times X \to [0,1])$ is a complete 1-bounded ultrametric space:

- $d(x,y) = 0$ iff $x = y$
- $d(x,y) = d(y,x)$
- $d(x,z) \leq \max(d(x,y), d(y,z))$
- All Cauchy sequences converge

The Category of Ultrametric Spaces

A pair $(X, d : X \times X \to [0, 1])$ is a complete 1-bounded ultrametric space:

- $d(x, y) = 0$ iff $x = y$
- $d(x, y) = d(y, x)$
- $d(x, z) \leq \max(d(x, y), d(y, z))$
- All Cauchy sequences converge

A function $f : A \to B$ is *nonexpansive*, when for all a and a'

$$d_B(f\ a, f\ a') \leq d_A(a, a')$$

So f is *non-distance-increasing*

Streams as Ultrametric Spaces

Streams $S(X)$ can be equipped with an ultrametric

$$d(xs, xs') = 2^{-\min\{n \in \mathbb{N} \mid xs_n \neq xs'_n\}}$$

Streams as Ultrametric Spaces

Streams $S(X)$ can be equipped with an ultrametric

$$d(xs, xs') = 2^{-\min\{n \in \mathbb{N} \mid xs_n \neq xs'_n\}}$$

Distance increases, as xs and xs' differ sooner:

- Differ at time 0 — distance 1
- Differ at time 1 — distance $\frac{1}{2}$
- Differ at time 2 — distance $\frac{1}{4}$
- Never differ — distance 0

Nonexpansive Functions and Causality

Theorem

The nonexpansive functions $S(X) \to S(Y)$ are exactly the causal functions

Nonexpansive Functions and Causality

Theorem

The nonexpansive functions $S(X) \to S(Y)$ are exactly the causal functions

Idea as follows:

- Suppose xs and xs' first differ at position n

Nonexpansive Functions and Causality

Theorem

The nonexpansive functions $S(X) \to S(Y)$ are exactly the causal functions

Idea as follows:

- Suppose xs and xs' first differ at position n
- Then $d(xs, xs') = 2^{-n}$

Nonexpansive Functions and Causality

Theorem

The nonexpansive functions $S(X) \to S(Y)$ are exactly the causal functions

Idea as follows:

- Suppose xs and xs' first differ at position n
- Then $d(xs, xs') = 2^{-n}$
- So $d(f\ xs, f\ xs') \leq 2^{-n}$

Nonexpansive Functions and Causality

Theorem

The nonexpansive functions $S(X) \to S(Y)$ are exactly the causal functions

Idea as follows:

- Suppose xs and xs' first differ at position n
- Then $d(xs, xs') = 2^{-n}$
- So $d(f\ xs, f\ xs') \leq 2^{-n}$
- So at least the first n positions of $f\ xs$ and $f\ xs'$ agree

The Payoff, Part 1

What are the consequences of this abstract view?

The Payoff, Part 1

What are the consequences of this abstract view?

- The category of complete 1-bounded ultrmetric spaces is *Cartesian closed*

The Payoff, Part 1

What are the consequences of this abstract view?

- The category of complete 1-bounded ultrmetric spaces is *Cartesian closed*
- The lambda calculus can be interpreted in any CCC...

The Payoff, Part 1

What are the consequences of this abstract view?

- The category of complete 1-bounded ultrmetric spaces is *Cartesian closed*
- The lambda calculus can be interpreted in any CCC...
- ...so a good DSL for reactive programming is functional programming!

The Payoff, Part 2

Banach's Contraction Map Theorem

Every strictly contractive function $f : A \to A$ on a nonempty metric space A has a unique fixed point.

The Payoff, Part 2

Banach's Contraction Map Theorem

Every strictly contractive function $f : A \to A$ on a nonempty metric space A has a unique fixed point.

- "Strictly contractive" $=$ "well-founded feedback"

The Payoff, Part 2

Banach's Contraction Map Theorem

Every strictly contractive function $f : A \to A$ on a nonempty metric space A has a unique fixed point.

- "Strictly contractive" = "well-founded feedback"
- So $\mu(\lambda xs.\, 0 :: \mathsf{map}\ \mathsf{succ}\ xs) = 0, 1, 2, 3, \ldots$

The Payoff, Part 2

Banach's Contraction Map Theorem

Every strictly contractive function $f : A \to A$ on a nonempty metric space A has a unique fixed point.

- "Strictly contractive" = "well-founded feedback"
- So $\mu(\lambda xs.\, 0 :: \text{map succ } xs) = 0, 1, 2, 3, \ldots$
- Semantic intepretation of feedback...
- ...which ensures it is well-founded and deterministic

From Semantics to Type Theory

$$
\begin{array}{lcll}
A & ::= & P \mid A \to B \mid S(A) \mid \bullet A & \text{Types} \\
e & ::= & x \mid \lambda x.\, e \mid e\, e' & \\
 & \mid & \mathsf{cons}(e, e') \mid \mathsf{hd}(e) \mid \mathsf{tl}(e) & \\
 & \mid & \bullet e \mid \mathsf{await}(e) \mid \mathsf{fix}\, x : A.\, e & \text{Terms}
\end{array}
$$

From Semantics to Type Theory

$$
\begin{array}{rcll}
A & ::= & P \mid A \to B \mid S(A) \mid \bullet A & \text{Types} \\
e & ::= & x \mid \lambda x.\ e \mid e\ e' & \\
 & \mid & \mathsf{cons}(e, e') \mid \mathsf{hd}(e) \mid \mathsf{tl}(e) & \\
 & \mid & \bullet e \mid \mathsf{await}(e) \mid \mathsf{fix}\ x : A.\ e & \text{Terms}
\end{array}
$$

- $\bullet A$ is the *delay* modality

From Semantics to Type Theory

$$
\begin{array}{lcll}
A & ::= & P \mid A \to B \mid S(A) \mid \bullet A & \text{Types} \\
e & ::= & x \mid \lambda x.\ e \mid e\ e' & \\
 & \mid & \mathrm{cons}(e, e') \mid \mathrm{hd}(e) \mid \mathrm{tl}(e) & \\
 & \mid & \bullet e \mid \mathrm{await}(e) \mid \mathrm{fix}\ x : A.\ e & \text{Terms}
\end{array}
$$

- $\bullet A$ is the *delay* modality
  - $\bullet(A, d) = (A, d')$ where
  - $d'(a, a') = \frac{1}{2} \cdot d(a, a')$

From Semantics to Type Theory

$$
\begin{array}{lcll}
A & ::= & P \mid A \to B \mid S(A) \mid \bullet A & \text{Types} \\
e & ::= & x \mid \lambda x.\, e \mid e\ e' & \\
& \mid & \mathsf{cons}(e, e') \mid \mathsf{hd}(e) \mid \mathsf{tl}(e) & \\
& \mid & \bullet e \mid \mathsf{await}(e) \mid \mathsf{fix}\ x : A.\ e & \text{Terms}
\end{array}
$$

- $\bullet A$ is the *delay* modality
  - $\bullet(A, d) = (A, d')$ where
  - $d'(a, a') = \frac{1}{2} \cdot d(a, a')$
- $\bullet S(A)$ are "streams starting on the next time step"

From Semantics to Type Theory

$$\begin{array}{lcll}
A & ::= & P \mid A \to B \mid S(A) \mid \bullet A & \text{Types} \\
e & ::= & x \mid \lambda x.\ e \mid e\ e' & \\
 & \mid & \mathsf{cons}(e, e') \mid \mathsf{hd}(e) \mid \mathsf{tl}(e) & \\
 & \mid & \bullet e \mid \mathsf{await}(e) \mid \mathsf{fix}\ x : A.\ e & \text{Terms}
\end{array}$$

- $\bullet A$ is the *delay* modality
  - $\bullet(A, d) = (A, d')$ where
  - $d'(a, a') = \frac{1}{2} \cdot d(a, a')$
- $\bullet S(A)$ are "streams starting on the next time step"
- $\epsilon : \bullet A \to B \simeq \bullet A \to \bullet B$ and $\zeta : \bullet A \times B \simeq \bullet A \times \bullet B$

From Semantics to Type Theory

$$
\begin{array}{rcll}
A & ::= & P \mid A \to B \mid S(A) \mid \bullet A & \text{Types} \\
e & ::= & x \mid \lambda x.\ e \mid e\ e' & \\
 & \mid & \mathsf{cons}(e, e') \mid \mathsf{hd}(e) \mid \mathsf{tl}(e) & \\
 & \mid & \bullet e \mid \mathsf{await}(e) \mid \mathsf{fix}\ x : A.\ e & \text{Terms}
\end{array}
$$

- $\bullet A$ is the *delay* modality
  - $\bullet(A, d) = (A, d')$ where
  - $d'(a, a') = \frac{1}{2} \cdot d(a, a')$
- $\bullet S(A)$ are "streams starting on the next time step"
- $\epsilon : \bullet A \to B \simeq \bullet A \to \bullet B$ and $\zeta : \bullet A \times B \simeq \bullet A \times \bullet B$
- Banach's theorem has type $(\bullet A \to A) \to A$

The Typing Judgement

- Key judgement $\Gamma \vdash e :_j A$

The Typing Judgement

- Key judgement $\Gamma \vdash e :_j A$
- Read e has type A at *time* i

The Typing Judgement

- Key judgement $\Gamma \vdash e :_j A$
- Read e has type A at *time* i
- Context annotated with times $\Gamma ::= \cdot \mid \Gamma, x :_i A$

Typing Rules

$$\frac{i \leq j}{\Gamma, x :_i A \vdash x :_j A} \qquad \frac{\Gamma, x :_{i+1} A \vdash e :_i A}{\Gamma \vdash \mathsf{fix}\; x : A.\; e :_i A}$$

$$\frac{\Gamma, x :_i A \vdash e :_i B}{\Gamma \vdash \lambda x.\; e :_i A \to B} \qquad \frac{\Gamma \vdash e :_i A \to B \qquad \Gamma \vdash e' :_i A}{\Gamma \vdash e\; e' :_i B}$$

$$\frac{\Gamma \vdash e :_i A \qquad \Gamma \vdash e' :_{i+1} S(A)}{\Gamma \vdash \mathsf{cons}(e, e') :_i S(A)}$$

$$\frac{\Gamma \vdash e :_i S(A)}{\Gamma \vdash \mathsf{hd}(e) :_i A} \qquad \frac{\Gamma \vdash e :_i S(A)}{\Gamma \vdash \mathsf{tl}(e) :_{i+1} S(A)}$$

$$\frac{\Gamma \vdash e :_{i+1} A}{\Gamma \vdash \bullet e :_i \bullet A} \qquad \frac{\Gamma \vdash e :_i \bullet A}{\Gamma \vdash \mathsf{await}(e) :_{i+1} A}$$

Interpreting the Syntax

$$
\begin{array}{lcl}
[\![\Gamma \vdash e :_i A]\!] & \in & [\![\Gamma]\!] \to \bullet^i([\![A]\!]) \\
[\![\Gamma \vdash x :_i A]\!] & = & \delta^{i-j} \circ \pi_x \text{ when } x :_j A \in \Gamma \\
[\![\Gamma \vdash \lambda x.\ e :_i A \to B]\!] & = & \epsilon^i \circ \lambda([\![\Gamma, x :_i A \vdash e :_i B]\!]) \\
[\![\Gamma \vdash \mathrm{cons}(e, e') :_i S(A)]\!] & = & \bullet^i(\mathit{cons}) \circ \zeta^i \circ \langle [\![e]\!], [\![e']\!] \rangle \\
[\![\Gamma \vdash \mathrm{hd}(e) :_i A]\!] & = & \bullet^i(\mathit{head}) \circ [\![\Gamma \vdash e :_i S(A)]\!] \\
[\![\Gamma \vdash \mathrm{tl}(e) :_{i+1} S(A)]\!] & = & \bullet^i(\mathit{tail}) \circ [\![\Gamma \vdash e :_i S(A)]\!] \\
[\![\Gamma \vdash \bullet e :_i \bullet A]\!] & = & [\![\Gamma \vdash e :_{i+1} A]\!] \\
[\![\Gamma \vdash \mathrm{await}(e) :_{i+1} A]\!] & = & [\![\Gamma \vdash e :_i \bullet A]\!] \\
[\![\Gamma \vdash \mathrm{fix}\ x : A.\ e :_{i+1} A]\!] & = & \lambda \gamma.\ \mu(\lambda v.\ [\![\Gamma, x :_{i+1} A \vdash e :_i A]\!]\ (\gamma, v))
\end{array}
$$

Example 1: `fibs`

```
fibs : S(ℕ)                                 // t = 0
fibs = fix xs:S(ℕ).                         // t = 1
           let ys = tl(xs) in               // t = 2
           cons(1, cons(1, map (+) (zip xs ys)))
```

Example 1: `fibs`

```
fibs : S(ℕ)                                  // t = 0
fibs = fix xs:S(ℕ).                          // t = 1
           let ys = tl(xs) in                // t = 2
           cons(1, cons(1, map (+) (zip xs ys)))
```

- Looks like a lazy functional program

Example 1: `fibs`

```
fibs : S(ℕ)                                    // t = 0
fibs = fix xs:S(ℕ).                            // t = 1
         let ys = tl(xs) in                    // t = 2
         cons(1, cons(1, map (+) (zip xs ys)))
```

- Looks like a lazy functional program
- Note *xs* and *ys* are at different times

Example 2: `map`

$$
\begin{array}{l}
\texttt{map : } (A \;\rightarrow\; B) \;\rightarrow\; S(A) \;\rightarrow\; S(B) \\
\texttt{map = } \lambda\texttt{f : } A \;\rightarrow\; B\texttt{.} \\
\qquad\qquad \texttt{fix r : } S(A) \;\rightarrow\; S(B)\texttt{.} \\
\qquad\qquad\qquad \lambda\texttt{xs:}S(A)\texttt{. cons(f (hd xs), r (tl xs))}
\end{array}
$$

Example 2: `map`

$$
\begin{array}{l}
\texttt{map} : (A \to B) \to S(A) \to S(B) \\
\texttt{map} = \lambda\texttt{f} : A \to B. \\
\qquad\qquad \texttt{fix r} : S(A) \to S(B). \\
\qquad\qquad\qquad \lambda\texttt{xs}:S(A).\ \texttt{cons(f (hd xs), r (tl xs))}
\end{array}
$$

- Looks like a functional program

Example 2: map

$$\begin{array}{l}
\texttt{map : } (A \;\to\; B) \;\to\; S(A) \;\to\; S(B) \\
\texttt{map = } \lambda\texttt{f : } A \;\to\; B\texttt{.} \\
\qquad\quad \texttt{fix r : } S(A) \;\to\; S(B)\texttt{.} \\
\qquad\qquad\quad \lambda\texttt{xs:}S(A)\texttt{. cons(f (hd xs), r (tl xs))}
\end{array}$$

- Looks like a functional program
- Note higher-type recursion

Non-Example: `illfounded`

illfounded $\not:$ $S(\mathbb{N})$
illfounded = fix xs:$S(\mathbb{N})$. xs

Non-Example: `illfounded`

$\texttt{illfounded} \not: S(\mathbb{N})$
$\texttt{illfounded} = \texttt{fix xs}:S(\mathbb{N}).\ \texttt{xs}$

- Ill-founded feedback

Non-Example: `illfounded`

$$\texttt{illfounded} \not: S(\mathbb{N})$$
$$\texttt{illfounded = fix xs:}S(\mathbb{N})\texttt{. xs}$$

- Ill-founded feedback
- “Time-checking” error: `xs` is at time 1, not 0

Non-Example: `illfounded`

$\texttt{illfounded} \not: S(\mathbb{N})$
$\texttt{illfounded = fix xs:}S(\mathbb{N})\texttt{. xs}$

- Ill-founded feedback
- "Time-checking" error: `xs` is at time 1, not 0
- Typing rules block unguarded recursion

How Can We Implement This

- Imperative implementation based on dataflow propagation [TLDI 2010]

How Can We Implement This

- Imperative implementation based on dataflow propagation [TLDI 2010]
- Idea similar to spreadsheets, self-adjusting computation [Acar *et al.*], subject-observer

How Can We Implement This

- Imperative implementation based on dataflow propagation [TLDI 2010]
- Idea similar to spreadsheets, self-adjusting computation [Acar *et al.*], subject-observer
- Correctness proof via logical relation between imperative code and ultrametric semantics

How Can We Implement This

- Imperative implementation based on dataflow propagation [TLDI 2010]
- Idea similar to spreadsheets, self-adjusting computation [Acar *et al.*], subject-observer
- Correctness proof via logical relation between imperative code and ultrametric semantics
- Proof uses ideas similar to Dreyer and Hur [POPL 2011]

Demo

Demo!

Conclusions

- Ultrametric semantics give a simple denotational semantics to reactive programs
- The lambda calculus is the correct DSL for this domain
- We can implement this!
- Look at our upcoming LICS 2011 paper
  `<http://research.microsoft.com/~nick/frp-lics11.pdf>`
- Look at our upcoming ICFP 2011 paper
  `<http://research.microsoft.com/~nick/guisemantics.pdf>`