Sketches of a RaTT

Fitch-Style Modal Calculi for Reactive Programming

Christian Graulund

IT University of Copenhagen

Talk given at LogSem, AU 2020

Reactive Programming

Reactive Programming

Reactive Programs

- A reactive program has continual interaction with environment.
- Includes control software, servers, GUI etc.
- Traditionally imperative with shared state and call-backs.
- Hence, error-prone and difficult to reason about.
- Many safety-critical systems are reactive.

Reactive Programming

Reactive Programs

- A reactive program has continual interaction with environment.
- Includes control software, servers, GUI etc.
- Traditionally imperative with shared state and call-backs.
- Hence, error-prone and difficult to reason about.
- Many safety-critical systems are reactive.

Why *Functional* Reactive Programs?

- We want to reason about reactive programs.
- We want high abstraction with efficient implementations.
- We want modular programs.
- We want safety guarantees.

Functional Reactive Programming

- FRP[1] is programming with signals.
- Signals are values that *vary over time*.
- Programs are signal transducers:

$$\mathit{prog} : \mathrm{Signal}\ A \to \mathrm{Signal}\ B$$

One implementation is signals as streams:

$$\mathrm{Stream}\ A \cong A \times \mathrm{Stream}\ A$$

Known problems include *causality*, *productivity* and *space-leaks*.

[1] Elliott and Hudak, 1997.

Causality and Productivity

Causality
A program is causal *(implementable)* if the *nth* output depends only on the first n inputs.

$$\mathrm{noncausal} : \mathrm{Stream}\ A \to \mathrm{Stream}\ A$$
$$\mathrm{noncausal}\ as = \mathrm{head}(\mathrm{tail}\ as) :: \mathrm{noncausal}\ as$$

Productivity
A program is productive *(useful)* if something is output at every n.

$$\mathrm{nonproductive} : \mathrm{Stream}\ A$$
$$\mathrm{nonproductive} = \mathrm{tail}\ \mathrm{nonproductive}$$

Leaking Space

Space Leaks

A program has a space leak if the execution of the program uses more memory than expected and the memory is released later than expected.

bs	F	F	F	T	F	T	$\cdots$
ns	n_1	n_2	n_3	n_4	n_5	n_6	$\cdots$
f ns bs	0	0	0	n_1	0	n_1	$\cdots$

Leaking Space

Space Leaks
A program has a space leak if the execution of the program uses more memory than expected and the memory is released later than expected.

bs	F	F	F	T	F	T	$\cdots$
ns	n_1	n_2	n_3	n_4	n_5	n_6	$\cdots$
f ns bs	0	0	0	n_1	0	n_1	$\cdots$

$$
\begin{aligned}
&\mathit{leakyF} : \text{Stream Bool} \to \text{Stream Nat} \to \text{Stream Nat}\\
&\mathit{leakyF}\ \mathit{bs}\ \mathit{ns} = \text{let } g\ s = \text{if } (\text{head } s) \text{ then } (\text{head } \mathit{ns}) :: g\ (\text{tail } s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else } 0 \qquad\qquad\quad :: g\ (\text{tail } s)\\
&\qquad\qquad\qquad\ \ \text{in } g\ \mathit{bs}
\end{aligned}
$$

Plugging a Leak

The problem is that streams is not stable over time and this can lead to leaks.

$$
\begin{aligned}
&\mathit{safeF} : \text{Stream Bool} \to \text{Stream Nat} \to \text{Stream Nat}\\
&\mathit{safeF}\ \mathit{bs}\ \mathit{ns} = \text{let } n = \text{head } \mathit{ns} \text{ in}\\
&\qquad\qquad\quad \text{let } g\ s = \text{if (head } s\text{) then } n :: g\ (\text{tail } s)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{else } 0\ \ :: g\ (\text{tail } s)\\
&\qquad\qquad\quad \text{in } g\ \mathit{bs}
\end{aligned}
$$

To avoid leaks, we should only store stable values for future retrival, but they can not be differentiated in the naive approach.

Known Solution

Possible Solution: Restricted API

- Restrict direct access to signals.
- Restrict FRP to predefined combinators.
- E.g. arrowized FRP[2] (arrows are signal transformers)

Drawbacks

- Loose simplicity and flexibility of original formulation.
- We want to make signals first class again!

[2] Nilsson et. al. 2002.

Known Solution

Possible Solution: Restricted API

- Restrict direct access to signals.
- Restrict FRP to predefined combinators.
- E.g. arrowized FRP[2] (arrows are signal transformers)

Drawbacks

- Loose simplicity and flexibility of original formulation.
- We want to make signals first class again!

Solution with first class signals: Modal FRP

[2] Nilsson et. al. 2002.

Modal FRP

Modal FRP[3] = FRP + modal types

- Add modality $\triangleright$ pronounced *"Later"*.
- $\triangleright A$ denotes *"A one time step from now"*.
- We now work with guarded streams:

$$\text{Stream A} \cong \text{A} \times \triangleright(\text{Stream A})$$

[3] Jeffrey 2012, 2014; Jeltsch 2013; Krishnaswami 2013.

Modal FRP

Modal FRP[3] = FRP + modal types

- Add modality $\triangleright$ pronounced *"Later"*.
- $\triangleright A$ denotes *"A one time step from now"*.
- We now work with guarded streams:

$$\text{Stream A} \cong \text{A} \times \triangleright(\text{Stream A})$$

Ensures Causality

$$\begin{aligned} &\text{noncausal} : \text{Stream } A \to \text{Stream } A \\ &\text{noncausal } as = \text{head } \underbrace{(\text{tail } as)}_{\text{type error}} :: \text{noncausal } as \end{aligned}$$

- Similarly for productivity

[3] Jeffrey 2012, 2014; Jeltsch 2013; Krishnaswami 2013.

Simply RaTT: The language

Simply RaTT: An overview

Goal:

- Full dependent type theory for reactive programming (RaTT).

Simply RaTT:

- A simply typed calcus for modal FRP.
- Fitch-style approach:
  - Removes need for let-bindings.
  - Allows simple and concise programs.

Contributions:

- Heap-based operational semantics for streams and transducers.
- Disallow (implicit) space leaks *by construction.*[4]
- Type system that ensures safety, causality and productivity.

[4] Following Krishnaswami.

Examples with Simplified Syntax

$$
\begin{aligned}
&\mathit{const} : A \text{ stable} \Rightarrow A \to \text{Stream } A \\
&\mathit{const}\ a\, \sharp\ = a :: \text{delay}\ (\mathit{const}\ a)
\end{aligned}
$$

Examples with Simplified Syntax

$$\mathit{const} : A \text{ stable} \Rightarrow A \to \text{Stream } A$$
$$\mathit{const}\ a\,\sharp\ = a :: \text{delay } (\mathit{const}\ a)$$

$$\mathit{zip} : \text{Stream } A \to \text{Stream } B \to \text{Stream } (A \times B)$$
$$\mathit{zip}\,\sharp\,(a :: \mathit{as})\ (b :: \mathit{bs}) = (a, b) :: \text{delay } (\mathit{zip}\ (\text{adv } \mathit{as})\ (\text{adv } \mathit{bs}))$$

Examples with Simplified Syntax

$$\begin{aligned}&\mathit{const} : A \text{ stable} \Rightarrow A \to \text{Stream } A\\&\mathit{const}\ a\,\sharp\ = a :: \text{delay } (\mathit{const}\ a)\end{aligned}$$

$$\begin{aligned}&\mathit{zip} : \text{Stream } A \to \text{Stream } B \to \text{Stream } (A \times B)\\&\mathit{zip}\,\sharp\,(a :: as)\ (b :: bs) = (a, b) :: \text{delay } (\mathit{zip}\ (\text{adv } as)\ (\text{adv } bs))\end{aligned}$$

$$\begin{aligned}&\mathit{switch} : \text{Stream } A \to \text{Ev } (\text{Stream } A) \to \text{Stream } A\\&\mathit{switch}\,\sharp\,(x :: xs)\ (\text{wait } es) = x :: \text{delay } (\mathit{switch}\ (\text{adv } xs)\ (\text{adv } es))\\&\mathit{switch}\,\sharp\,xs\qquad\quad (\text{val } ys)\quad = ys\end{aligned}$$

Type System

Let considered harmful

Traditional: Dual contexts

$$\frac{\Theta \mid \emptyset \vdash t : A}{\Gamma \mid \Theta \vdash \mathsf{delay}(t) : \triangleright A}$$

$$\frac{\begin{array}{c}\Gamma \mid \Theta \vdash t : \triangleright A \\ \Gamma \mid \Theta, x : A \vdash t' : C\end{array}}{\Gamma \mid \Theta \vdash \mathsf{let}\ x = t\ \mathsf{in}\ t' : C}$$

$$\frac{}{\underbrace{\Gamma, x : A, \Gamma'}_{\textit{now}} \mid \underbrace{\Theta}_{\textit{later}} \vdash x : A}$$

Let considered harmful

Traditional: Dual contexts

$$\frac{\Theta \mid \emptyset \vdash t : A}{\Gamma \mid \Theta \vdash \mathsf{delay}(t) : \triangleright A}$$

$$\frac{\Gamma \mid \Theta \vdash t : \triangleright A \qquad \Gamma \mid \Theta, x : A \vdash t' : C}{\Gamma \mid \Theta \vdash \mathsf{let}\ x = t\ \mathsf{in}\ t' : C}$$

$$\frac{}{\underbrace{\Gamma, x : A, \Gamma'}_{\textit{now}} \mid \underbrace{\Theta}_{\textit{later}} \vdash x : A}$$

Modern: Fitch-style

$$\frac{\overbrace{\Gamma}^{\textit{earlier}}, \checkmark \vdash t : A}{\underbrace{\Gamma}_{\textit{now}} \vdash \mathsf{delay}(t) : \triangleright A}$$

$$\frac{\Gamma \vdash t : \triangleright A}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{adv}(t) : A}$$

$$\frac{\checkmark\text{-free}(\Gamma')}{\Gamma, x : A, \Gamma' \vdash x : A}$$

Stability

To know what values are safe to transport into the future we have two notions:

Stable types

- Types that are inherently stable.
- These are Nat, 1 and products and sums of these.

Stability

To know what values are safe to transport into the future we have two notions:

Stable types

- Types that are inherently stable.
- These are Nat, 1 and products and sums of these.

Box Modality

- Given a type A, we can restrict it to its stable terms.
- Represented by box modality, $\Box A$.
- $\Box A$ is a *stable type.*

$$\frac{\Gamma, \sharp \vdash t : A}{\Gamma \vdash \mathsf{box}(t) : \Box A} \qquad \frac{\Gamma \vdash t : \Box A}{\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}(t) : A}$$

Nakano Style Recursion

To ensure *causality* and *productivity* of recursive definition we use modified Nakano Style fixed point.

$$\frac{\Gamma, \sharp, x : \triangleright A \vdash t : A}{\Gamma \vdash \mathsf{fix}\, x.t : \Box A}$$

Crucially, fixed points are stable to allow the recursive call *in the future*.

In general we do not have $A \to \triangleright A$, but we do have $\Box A \to \triangleright \Box A$.

Examples

$$(\circledast) : \triangleright (A \to B) \to \triangleright A \to \triangleright B$$
$$f \circledast a = \mathsf{delay}\ ((\mathsf{adv}\ f)\ (\mathsf{adv}\ a))$$

$$(\boxast) : \Box (A \to B) \to \Box A \to \Box B$$
$$f \boxast a = \mathsf{box}\ ((\mathsf{unbox}\ f)\ (\mathsf{unbox}\ a))$$

Examples

$$(\circledast) : \rhd (A \to B) \to \rhd A \to \rhd B$$
$$f \circledast a = \mathrm{delay}\ ((\mathrm{adv}\ f)\ (\mathrm{adv}\ a))$$

$$(\boxast) : \Box (A \to B) \to \Box A \to \Box B$$
$$f \boxast a = \mathrm{box}\ ((\mathrm{unbox}\ f)\ (\mathrm{unbox}\ a))$$

$$map : \Box (A \to B) \to \Box (\mathrm{Stream}\ A \to \mathrm{Stream}\ B)$$
$$map = \lambda f\,.\,\mathrm{fix}\ map'\,.\,\lambda a\,.\,(\mathrm{unbox}\ f)\ (\mathrm{head}\ a) :: map' \circledast \mathrm{tail}\ a$$

Examples

$$(\circledast) : \triangleright (A \to B) \to \triangleright A \to \triangleright B$$
$$f \circledast a = \mathsf{delay}\ ((\mathsf{adv}\ f)\ (\mathsf{adv}\ a))$$

$$(\boxast) : \Box (A \to B) \to \Box A \to \Box B$$
$$f \boxast a = \mathsf{box}\ ((\mathsf{unbox}\ f)\ (\mathsf{unbox}\ a))$$

$$map : \Box (A \to B) \to \Box (\mathsf{Stream}\ A \to \mathsf{Stream}\ B)$$
$$map = \lambda f\,.\,\mathsf{fix}\ map'\,.\,\lambda a\,.\,(\mathsf{unbox}\ f)\ (\mathsf{head}\ a) :: map' \circledast \mathsf{tail}\ a$$

$$mapSugar : \Box (A \to B) \to \Box (\mathsf{Stream}\ A \to \mathsf{Stream}\ B)$$
$$mapSugar\ f \sharp (a :: as) = (\mathsf{unbox}\ f)\ a :: mapSugar\ f \circledast as$$

Eliminating Space Leaks

Evaluation and Step Semantics

We define two heap based semantics for Simply RaTT, an evaluation semantics and a step semantics.

Evaluation and Step Semantics

We define two heap based semantics for Simply RaTT, an evaluation semantics and a step semantics.

The evaluation semantics describes how a term evaluates to a value with a given store:

$$\langle t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle$$

Evaluation and Step Semantics

We define two heap based semantics for Simply RaTT, an evaluation semantics and a step semantics.

The evaluation semantics describes how a term evaluates to a value with a given store:

$$\langle t; \sigma \rangle \Downarrow \langle v; \sigma' \rangle$$

The step semantics describes how an *internal* stream evaluates over time. Given $\vdash t : \mathsf{Str}\ A$, we have

$$\langle t; \eta \rangle \xrightarrow{v_1} \langle t_1; \eta_1 \rangle \xrightarrow{v_2} \langle t_2; \eta_2 \rangle \ldots$$

Evaluation and Step Semantics

We define two heap based semantics for Simply RaTT, an evaluation semantics and a step semantics.

The evaluation semantics describes how a term evaluates to a value with a given store:

$$\langle t; \sigma \rangle \Downarrow \left\langle v; \sigma' \right\rangle$$

The step semantics describes how an *internal* stream evaluates over time. Given $\vdash t : \mathsf{Str}\, A$, we have

$$\langle t; \eta \rangle \xRightarrow{v_1} \langle t_1; \eta_1 \rangle \xRightarrow{v_2} \langle t_2; \eta_2 \rangle \ldots$$

In both cases, the store is used for delayed computations, recursive calls and for transducers, inputs from the environment.

Shape of the store

The store σ is of the form:

$$\sigma ::= \bullet \mid \eta_L \mid \eta_N \checkmark \eta_L$$

where η_N, η_L are heaps, i.e., finite maps from locations to terms.

Shape of the store

The store σ is of the form:

$$\sigma ::= \bullet \mid \eta_L \mid \eta_N \checkmark \eta_L$$

where η_N, η_L are heaps, i.e., finite maps from locations to terms.

The shape of the store corresponds to capabilities:

- $\bullet$ allows neither reading or writing.
- η_L allows writing but not reading.
- $\eta_N \checkmark \eta_L$ allows both reading and writing.

Evaluation Rules

$$\frac{l = \mathsf{alloc}\,(\sigma) \qquad \sigma \neq \bullet}{\langle \mathsf{delay}\; t; \sigma \rangle \Downarrow \langle l; (\sigma, l \mapsto t) \rangle}$$

$$\frac{\langle t; \eta_N \rangle \Downarrow \langle l; \eta'_N \rangle \qquad \langle \eta'_N(l); (\eta'_N \checkmark \eta_L) \rangle \Downarrow \langle v; \sigma' \rangle}{\langle \mathsf{adv}\; t; (\eta_N \checkmark \eta_L) \rangle \Downarrow \langle v; \sigma' \rangle}$$

Evaluation Rules

$$\frac{l = \mathrm{alloc}\,(\sigma) \qquad \sigma \neq \bullet}{\langle \mathrm{delay}\; t; \sigma\rangle \Downarrow \langle l; (\sigma, l \mapsto t)\rangle}$$

$$\frac{\langle t; \eta_N\rangle \Downarrow \left\langle l; \eta'_N\right\rangle \qquad \left\langle \eta'_N(l); (\eta'_N \checkmark \eta_L)\right\rangle \Downarrow \left\langle v; \sigma'\right\rangle}{\langle \mathrm{adv}\; t; (\eta_N \checkmark \eta_L)\rangle \Downarrow \left\langle v; \sigma'\right\rangle}$$

$$\frac{\langle t; \bullet\rangle \Downarrow \left\langle \mathrm{box}\; t'; \bullet\right\rangle \qquad \left\langle t'; \sigma\right\rangle \Downarrow \left\langle v; \sigma'\right\rangle \qquad \sigma \neq \bullet}{\langle \mathrm{unbox}\; t; \sigma\rangle \Downarrow \left\langle v; \sigma'\right\rangle}$$

$$\frac{\begin{array}{c}\langle t; \bullet\rangle \Downarrow \left\langle \mathrm{fix}\, x.t'; \bullet\right\rangle \\ \left\langle t'[\mathrm{box}(\mathrm{delay}(\mathrm{unbox}(\mathrm{fix}\; x.t')))/x]; \sigma\right\rangle \Downarrow \left\langle v; \sigma'\right\rangle \qquad \sigma \neq \bullet\end{array}}{\langle \mathrm{unbox}\; t; \sigma\rangle \Downarrow \left\langle v; \sigma'\right\rangle}$$

Step Semantics

Given t : Str A we define

$$\frac{\langle t; (\eta\checkmark)\rangle \Downarrow \langle v :: l; (\eta_N \checkmark \eta_L)\rangle}{\langle t; \eta\rangle \overset{v}{\Longrightarrow} \langle \mathsf{adv}(l); \eta_L\rangle}$$

- After evaluation we delete the entire “now” heap η_N.
- All data must be *explicitely* moved forward to remain available.

Step Semantics

Given t : Str A we define

$$\frac{\langle t; (\eta\checkmark)\rangle \Downarrow \langle v :: l; (\eta_N \checkmark \eta_L)\rangle}{\langle t; \eta\rangle \overset{v}{\Longrightarrow} \langle \mathrm{adv}(l); \eta_L\rangle}$$

- After evaluation we delete the entire "now" heap η_N.
- All data must be *explicitely* moved forward to remain available.

We additionally define a reactive step semantics for transducers, i.e, given $\vdash t$: Str $A \to$ Str B we define

$$\frac{\begin{gathered}\left\langle t; (\eta, l \mapsto v :: l' \checkmark l' \mapsto \langle\rangle)\right\rangle \Downarrow \left\langle v' :: w; (\eta_N \checkmark \eta_L, l' \mapsto \langle\rangle)\right\rangle \\ l' = \mathrm{alloc}\,(\eta\checkmark)\end{gathered}}{\langle t; \eta; l\rangle \overset{v/v'}{\Longrightarrow} \langle \mathrm{adv}\, w; \eta_L; l'\rangle}$$

- The inputs for the transducer are deleted after each step.

Productivity Theorems

Our main results are productivity and causality theorems:

Theorem (Productivity)
Given $\vdash t : \Box(\mathsf{Str}\ A)$ *and any* $n \in$ *nats, there exists a reduction sequences*

$$\langle \mathsf{unbox}\ t; \emptyset \rangle \xRightarrow{v_1} \langle t_1; \eta_1 \rangle \xRightarrow{v_2} \ldots \xRightarrow{v_n} \langle t_n; \eta_n \rangle$$

s.t. $\forall 1 \leqslant i \leqslant n. \vdash v_i : A.$

Productivity Theorems

Our main results are productivity and causality theorems:

Theorem (Productivity)
Given $\vdash t : \Box(\mathsf{Str}\ A)$ *and any* $n \in$ *nats, there exists a reduction sequences*

$$\langle \mathsf{unbox}\ t; \emptyset \rangle \overset{v_1}{\Longrightarrow} \langle t_1; \eta_1 \rangle \overset{v_2}{\Longrightarrow} \ldots \overset{v_n}{\Longrightarrow} \langle t_n; \eta_n \rangle$$

s.t. $\forall 1 \leqslant i \leqslant n. \vdash v_i : A.$

Theorem (Causality, simplified)
Given $\vdash t : \Box(\mathsf{Str}\ A \to \mathsf{Str}\ B)$ *and inputs* $\vdash v_i : A$ *there is a reduction*

$$\langle \mathsf{unbox}\ t; \emptyset \rangle \overset{v_1/v_1'}{\Longrightarrow} \langle t_1; \eta_1 \rangle \overset{v_2/v_2'}{\Longrightarrow} \ldots$$

s.t. $\forall i. \vdash v_i' : B.$

Step-Indexed Kripke Logical Relations

Worlds

The worlds are triples $(\sigma, \overline{\eta}, \beta)$ where, σ is a store, $\overline{\eta}$ is an infinite sequence of heaps and $\beta < \omega$ is an ordinal.

The stores describes the state of the store, the sequence describes possible future inputs and the ordinal is the step-index

Delay Semantics

$$\mathcal{V}[\![\rhd A]\!](\sigma, (\eta; \overline{\eta}), \beta) = \begin{cases} \mathsf{dom}\,(\mathsf{gc}\,(\sigma)) & \beta = 0 \\ \{\, l \mid \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \beta') \,\} & \beta = \beta' + 1 \end{cases}$$

where

$$\mathsf{gc}\,(\sigma) = \begin{cases} \eta_L & \sigma = \eta_N \checkmark \eta_L \\ \sigma & \text{otherwise} \end{cases}$$

describes the semantics of garbage collection.

Box Semantics

The semantics of $\Box$ describes stability:

$$\mathcal{V}[\![\Box A]\!](\sigma, \overline{\eta}, \beta) = \left\{ t \,\middle|\, \forall \overline{\eta}'.\mathsf{unbox}\, t \in \mathcal{T}[\![A]\!](\emptyset, \overline{\eta}', \beta) \right\}$$

Note that $\emptyset \neq \bullet$ and $\overline{\eta}'$ is freely chosen.

Contexts

The tick desribes the passage of time in the context:

$$\mathcal{C}[\![\Gamma, \checkmark]\!]((\eta_N \checkmark \eta_L), \overline{\eta}, \beta) = \mathcal{C}[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta + 1)$$

Contexts

The tick desribes the passage of time in the context:

$$\mathcal{C}[\![\Gamma, \checkmark]\!]((\eta_N \checkmark \eta_L), \overline{\eta}, \beta) = \mathcal{C}[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta + 1)$$

The lock describes when the store, and hence writing and possibly reading, is available:

$$\mathcal{C}[\![\Gamma, \sharp]\!](\sigma, \overline{\eta}, \beta) = \bigcup_{\overline{\eta}'} \mathcal{C}[\![\Gamma]\!](\bullet, \overline{\eta}', \beta) \qquad\qquad \sigma \neq \bullet$$

Lively RaTT: Adding temporal inductive types

Lively RaTT

- Lively RaTT[5] is an extension of Simply RaTT.
- The goal was to add temporal inductive types to Simply RaTT.
- We want to reason about *liveness* for reactive systems.
- We add the *until* type, $A \,\mathcal{U}\, B$, from linear temporal logic.
- Problem: In systems with guarded recursion, least and greatest fixpoints coincide.

[5] Bahr. et. al: Diamonds Are Not Forever: Liveness in Guarded Reactive Programming.

A new sub-modality

- Solution: We consider a sub-modality of $\triangleright$, denoted $\bigcirc$.

$$\frac{\Gamma, \checkmark_m \vdash t : A \qquad m \in \{\bigcirc, \triangleright\}}{\Gamma \vdash \mathsf{delay}\, t : m\, A} \qquad\qquad \frac{\Gamma \vdash t : m\, A \qquad m \leqslant m'}{\Gamma, \checkmark_{m'}, \Gamma' \vdash \mathsf{adv}\, t : A}$$

A new sub-modality

- Solution: We consider a sub-modality of $\triangleright$, denoted $\bigcirc$.

$$\frac{\Gamma, \checkmark_m \vdash t : A \qquad m \in \{\bigcirc, \triangleright\}}{\Gamma \vdash \mathsf{delay}\, t : m\, A} \qquad\qquad \frac{\Gamma \vdash t : m\, A \qquad m \leqslant m'}{\Gamma, \checkmark_{m'}, \Gamma' \vdash \mathsf{adv}\, t : A}$$

- We always have an inclusion $\bigcirc A \to \triangleright A$, but no general inclusion $\triangleright A \to \bigcirc A$.
- This will induce an inclusion from inductive types into co-inductive types, but not the other way.
- Operationally, $\bigcirc$ and $\triangleright$ have the same behaviour.

Until types restricted to $\bigcirc$:

- Until types, in particular until recursion, is restricted to $\bigcirc$:

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{now}\, t : A\,\mathcal{U}\, B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : \bigcirc(A\,\mathcal{U}\, B)}{\Gamma \vdash \mathsf{wait}\, s\, t : A\,\mathcal{U}\, B}$$

$$\frac{\begin{array}{c}\Gamma, \sharp, x : B \vdash s : C \\ \Gamma, \sharp, x : A, y : \bigcirc(A\,\mathcal{U}\, B), z : \bigcirc C \vdash t : C \qquad \Gamma, \sharp, \Gamma' \vdash u : A\,\mathcal{U}\, B\end{array}}{\Gamma, \sharp, \Gamma' \vdash \mathsf{rec}_{\mathcal{U}}(x.s, x\, y\, z.t, u) : C}$$

Until types restricted to $\bigcirc$:

- Until types, in particular until recursion, is restricted to $\bigcirc$:

$$\frac{\Gamma \vdash t : B}{\Gamma \vdash \mathsf{now}\, t : A\,\mathcal{U}\,B} \qquad \frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : \bigcirc(A\,\mathcal{U}\,B)}{\Gamma \vdash \mathsf{wait}\, s\, t : A\,\mathcal{U}\,B}$$

$$\frac{\begin{array}{c}\Gamma, \sharp, x : B \vdash s : C \\ \Gamma, \sharp, x : A, y : \bigcirc(A\,\mathcal{U}\,B), z : \bigcirc C \vdash t : C \qquad \Gamma, \sharp, \Gamma' \vdash u : A\,\mathcal{U}\,B\end{array}}{\Gamma, \sharp, \Gamma' \vdash \mathsf{rec}_{\mathcal{U}}(x.s, x\,y\,z.t, u) : C}$$

The restriction ensures that the following is not well-typed:

$$\boxed{\begin{array}{l} \mathit{waitForever} : \Box\, A \to \Box\, (A\ \mathcal{U}\ B) \\ \mathit{waitForever}\ a = \mathsf{fix}\ w\,.\,\mathsf{wait}\ (\mathsf{unbox}\ a)\ (\mathsf{delay}\ (\mathsf{adv}\ w)) \end{array}}$$

Examples

We can encode terminating events of A, denoted $\Diamond A$, as $1\,\mathcal{U}\ A$.

$\Diamond A$ is *almost* a monad:

$$\boxed{\begin{array}{l} \mathit{returnDia} : \Box\ (A \to \Diamond\ A) \\ \mathit{returnDia} = \text{box}\ (\lambda a\,.\,\text{now}\ a) \\ \mathit{bindDia} : \Box\ (A \to \Diamond\ B) \to \Box\ (\Diamond\ A \to \Diamond\ B) \\ \mathit{bindDia} = \lambda f\,.\,\text{box}\ (\lambda \mathit{dia}\,.\,\text{rec}_{\mathcal{U}}\ (a\,.\,(\text{unbox}\ f)\ a, u\ w\ d\,.\,\text{wait}\ \langle\rangle\ d, \mathit{dia})) \end{array}}$$

Examples

We can encode terminating events of A, denoted $\Diamond A$, as $1\,\mathcal{U}\ A$.

$\Diamond A$ is *almost* a monad:

$$
\begin{array}{|l|}
\hline
\textit{returnDia} : \Box\ (A \to \Diamond\ A) \\
\textit{returnDia} = \mathsf{box}\ (\lambda a\,.\,\mathsf{now}\ a) \\
\textit{bindDia} : \Box\ (A \to \Diamond\ B) \to \Box\ (\Diamond\ A \to \Diamond\ B) \\
\textit{bindDia} = \lambda f\,.\,\mathsf{box}\ (\lambda \textit{dia}\,.\,\mathsf{rec}_{\mathcal{U}}\ (a\,.\,(\mathsf{unbox}\ f)\ a, u\ w\ d\,.\,\mathsf{wait}\ \langle\rangle\ d, \textit{dia})) \\
\hline
\end{array}
$$

We can encode fair streams by *interleaving* the guarded fixpoint and until types:

$$\mathsf{Fair}\ A\ B := \mathsf{Fix}\,\alpha.A\,\mathcal{U}\ (B \times \triangleright(B\,\mathcal{U}\ (A \times \alpha)))$$

Evaluation semantics

We extend the evaluation semantics with cases for $A \,\mathcal{U}\, B$:

$$\frac{\langle t; \sigma\rangle \Downarrow \langle v; \sigma'\rangle}{\langle \mathsf{now}\ t; \sigma\rangle \Downarrow \langle \mathsf{now}\ v; \sigma'\rangle}$$

$$\frac{\langle t_1; \sigma\rangle \Downarrow \langle v_1; \sigma'\rangle \qquad \langle t_2; \sigma'\rangle \Downarrow \langle v_2; \sigma''\rangle}{\langle \mathsf{wait}\ t_1\ t_2; \sigma\rangle \Downarrow \langle \mathsf{wait}\ v_1\ v_2; \sigma''\rangle}$$

$$\frac{\langle u; \sigma\rangle \Downarrow \langle \mathsf{now}\ v; \sigma'\rangle \qquad \langle s[v/x]; \sigma'\rangle \Downarrow \langle w; \sigma''\rangle}{\langle \mathsf{rec}_{\mathcal{U}}(x.s, x\,y\,z.t, u); \sigma\rangle \Downarrow \langle w; \sigma''\rangle}$$

$$\frac{\begin{array}{c}\langle u; \sigma\rangle \Downarrow \langle \mathsf{wait}\ v_1\ v_2; \sigma'\rangle \\ \langle t[v_1/x, v_2/y, l/z]; (\sigma', l \mapsto \mathsf{rec}_{\mathcal{U}}(x.s, x\,y\,z.t, \mathsf{adv}(v_2)))\rangle \Downarrow \langle v'; \sigma''\rangle\end{array}}{\langle \mathsf{rec}_{\mathcal{U}}(x.s, x\,y\,z.t, u); \sigma\rangle \Downarrow \langle v'; \sigma''\rangle}$$

Extension to logical Relation

We extend the worlds with a new ordinal. The worlds are now $(\sigma, \overline{\eta}, \alpha, \beta)$ with $\alpha \leqslant \omega$ and $\beta < \omega \cdot 2$. α is for inductive unfolding, and β is for guarded recursive unfolding.

Extension to logical Relation

We extend the worlds with a new ordinal. The worlds are now $(\sigma, \overline{\eta}, \alpha, \beta)$ with $\alpha \leqslant \omega$ and $\beta < \omega \cdot 2$. α is for inductive unfolding, and β is for guarded recursive unfolding.

The step-index for guarded recursive unfolding is extendend beyond ω, as different from Simply RaTT, to allow *"enough time for unfolding"*.

Take it to the limit

The interpretation of $\bigcirc$ and $\triangleright$ differs at limit ordinals:

$$\mathcal{V}[\![\bigcirc A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta) = \begin{cases} \mathsf{dom}\,(\mathsf{gc}\,(\sigma)) \\ \{ l \mid \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta') \} \\ \{ l \mid \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta) \} \end{cases}$$

$$\mathcal{V}[\![\triangleright A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta) = \begin{cases} \mathsf{dom}\,(\mathsf{gc}\,(\sigma)) \\ \{ l \mid \mathsf{adv}\, l \in \mathcal{T}[\![A]\!](\mathsf{gc}\,(\sigma) \checkmark \eta, \overline{\eta}, \alpha, \beta') \} \\ \bigcap_{\beta' < \beta} \mathcal{V}[\![\triangleright A]\!](\sigma, (\eta; \overline{\eta}), \alpha, \beta') \end{cases}$$

where in both cases, the cases are for $\beta = 0, \beta = \beta' + 1$ and β being a limit ordinal.

Semantic sub-modality

That $\bigcirc$ is a sub-modality of $\triangleright$ has a clear semantic menaing.

Lemma (Sub-modality)

Given A and world w then

$$\mathcal{V}[\![\bigcirc A]\!](w) \subseteq \mathcal{V}[\![\triangleright A]\!](w)$$

Until semantics

The semantics of $A \,\mathcal{U}\, B$ captures termination. Let $w = (\sigma, \overline{\eta}, \alpha, \beta)$.

$$\mathcal{V}[\![A \,\mathcal{U}\, B]\!](w) =$$
$$\{\text{now}\ v \mid v \in \mathcal{V}[\![B]\!](\sigma, \overline{\eta}, \omega, \beta)\} \cup$$
$$\left\{\text{wait}\ v\ u \,\middle|\, v \in \mathcal{V}[\![A]\!](w) \wedge \exists \alpha' < \alpha. u \in \mathcal{V}[\![\bigcirc(A \,\mathcal{U}\, B)]\!]((\sigma, \overline{\eta}, \alpha', \beta)\right\}$$

At some point, the interpretation will only contain values of the form now v.

Contex interpretation

The two ticks are also different at the limit:

$$\mathcal{C}[\![\Gamma, \checkmark_{\triangleright}]\!]((\eta_N \checkmark \eta_L), \overline{\eta}, \beta) = \mathcal{C}[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta + 1)$$

$$\mathcal{C}[\![\Gamma, \checkmark_{\bigcirc}]\!]((\eta_N \checkmark \eta_L), \overline{\eta}, \beta) = \begin{cases} \mathcal{C}[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta) & \beta \text{ limit ordinal} \\ \mathcal{C}[\![\Gamma]\!](\eta_N, (\eta_L; \overline{\eta}), \beta + 1) & \text{otherwise} \end{cases}$$

Ongoing Work: Denotational Semantics

Denotational Semantics

Current work focuses on give a presheaf model of Simply RaTT.
We use the same worlds as for the logical relation.

Denotational Semantics

Current work focuses on give a presheaf model of Simply RaTT.
We use the same worlds as for the logical relation.

Modalities and corresponding tokens are interpreted using adjunctions:

$$\mathrm{Hom}(\Gamma\checkmark, A) \cong \mathrm{Hom}(\Gamma, \triangleright A)$$

$$\mathrm{Hom}(\Gamma\sharp, A) \cong \mathrm{Hom}(\Gamma, \Box A)$$

Denotational Semantics

Current work focuses on give a presheaf model of Simply RaTT. We use the same worlds as for the logical relation.

Modalities and corresponding tokens are interpreted using adjunctions:

$$\mathrm{Hom}(\Gamma\checkmark, A) \cong \mathrm{Hom}(\Gamma, \triangleright A)$$
$$\mathrm{Hom}(\Gamma\sharp, A) \cong \mathrm{Hom}(\Gamma, \Box A)$$

The safety guarentees are encoded using a garbage collection modality:

$$(\mathrm{GC}(A))(\sigma, \overline{\eta}, \beta) = A(\mathrm{gc}\left(\sigma\right), \overline{\eta}, \beta)$$

Types are interpreted as GC -algebras, i.e., we have

$$\mathcal{V}[\![A]\!] \cong \mathsf{GC}(\mathcal{V}[\![A]\!])$$

Denotational Semantics

Types are interpreted as GC -algebras, i.e., we have

$$\mathcal{V}[\![A]\!] \cong \mathsf{GC}(\mathcal{V}[\![A]\!])$$

Terms are interpretated with respect to an appropiate store object.
In particular, we interpreted a term of type A as a map

$$\mathcal{S} \to \mathcal{V}[\![A]\!]$$

The term interpretation is itself a monad.

Summary

Summary and Future Work

Summary

- Fitch-style approach to Modal FRP.
- New sub-modality approach.
- Heap-based operational semantics that rules out space leaks.
- Type system that ensures safety, causality and productivity.
- Extension ensures termination of until-types.

Summary and Future Work

Summary

- Fitch-style approach to Modal FRP.
- New sub-modality approach.
- Heap-based operational semantics that rules out space leaks.
- Type system that ensures safety, causality and productivity.
- Extension ensures termination of until-types.

Future Work

- Many ticks and many heaps.
- Denotational semantics.
- Logic on top of language.
- Extension to dependent types (`RaTT`).

Thank you

Thank you for your attention!

Questions?

Time Leaks

- Computation on tail of stream will never evaluate fully.

$$\begin{aligned}&\mathsf{leakyNats} : \mathsf{Str\ Nat}\\&\mathsf{leakyNats} = \mathsf{fix}\ ns.\ 0 :: \underbrace{\mathsf{delay\ unbox}(\mathsf{map}\ {+1})\ ns}_{\bigcirc \mathsf{Str\ Nat}}\end{aligned}$$

- We (roughly) have the unfolding:

$$\begin{aligned}&\quad 0 :: (\mathsf{map}\ {+1})\ ns\\&\rightsquigarrow 0 :: (\mathsf{map}\ {+1})\ (0 :: (\mathsf{map}\ {+1})\ ns)\\&\rightsquigarrow 0 :: 1 :: (\mathsf{map}\ {+1})\ (\mathsf{map}\ {+1})\ (0 :: (\mathsf{map}\ {+1})\ ns)\\&\rightsquigarrow 0 :: \cdots :: n-1 :: (\mathsf{map}\ {+1})^n\ (0 :: (\mathsf{map}\ {+1})\ ns)\end{aligned}$$

- To compute n we need to recompute $(n-1)$ elements!

Time Leaks II

- Solution: Disallow fixed points under delay.

$$\frac{\Gamma, \sharp, x : \bigcirc A \vdash t : A}{\Gamma \vdash \mathsf{fix}\ x.t : \Box A} \qquad \frac{\Gamma \vdash t : \Box A \qquad \checkmark\text{-free}(\Gamma')}{\Gamma, \sharp, \Gamma' \vdash \mathsf{unbox}\ t : A}$$

$$\begin{aligned} &\mathsf{leakyNats} : \mathsf{Str\ Nat} \\ &\mathsf{leakyNats} = \mathsf{fix}\ ns.\ 0 :: \mathsf{delay}\ \overbrace{\mathsf{unbox}(\mathsf{map}\ {+1})}^{\text{type error}}\ ns \end{aligned}$$

- Solution is *unique to Fitch-style approach*.
- We can write nats with explicit buffering:

$$\begin{aligned} &\mathsf{nats} : \Box(\mathsf{Str\ Nat}) && \mathsf{from} : \Box(\mathsf{Nat} \to \mathsf{Str}\ Nat) \\ &\mathsf{nats} = \mathsf{from} \boxdot 0 && \mathsf{from} n = n :: \mathsf{f} \odot (n+1) \end{aligned}$$

Stream Transducer:

- For $t : \mathsf{Str}\ A \to \mathsf{Str}\ B$:

$$\frac{\left\langle t; \sharp\eta, l^* \mapsto v :: l^* \checkmark l^* \mapsto \langle\rangle \right\rangle \Downarrow \left\langle v' :: l; \sharp\eta_N \checkmark \eta_L, l^* \mapsto \langle\rangle \right\rangle}{\langle t; \eta \rangle \xRightarrow{v/v'} \langle \mathsf{adv}\ l; \eta_L \rangle}$$

- Head of input stream is stored on heap and deleted after each iteration

Typing Rules:

$$\frac{\Gamma, x : A, \Gamma' \vdash \qquad \textsf{token-free}(\Gamma')}{\Gamma, x : A, \Gamma' \vdash x : A} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \langle\rangle : \mathbf{1}} \qquad \frac{n \in \mathbb{N}}{\Gamma \vdash \bar{n} : \textsf{Nat}}$$

$$\frac{\Gamma \vdash s : \textsf{Nat} \qquad \Gamma \vdash t : \textsf{Nat}}{\Gamma \vdash s + t : \textsf{Nat}} \qquad \frac{\Gamma, x : A \vdash t : B \qquad \textsf{tick-free}(\Gamma)}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\,t' : B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash t' : B}{\Gamma \vdash \left\langle t, t' \right\rangle : A \times B} \qquad \frac{\Gamma \vdash t : A_1 \times A_2 \qquad i \in \{1,2\}}{\Gamma \vdash \pi_i\, t : A_i}$$

$$\frac{\Gamma \vdash t : A_i \qquad i \in \{1,2\}}{\Gamma \vdash \textsf{in}_i\, t : A_1 + A_2} \qquad \frac{\Gamma, x : A_i \vdash t_i : B \qquad \Gamma \vdash t : A_1 + A_2 \qquad i \in \{1,2\}}{\Gamma \vdash \textsf{case}\, t\, \textsf{of}\, \textsf{in}_1\, x.t_1; \textsf{in}_2\, x.t_2 : B}$$

$$\frac{\Gamma, \checkmark \vdash t : A}{\Gamma \vdash \textsf{delay}\, t : \bigcirc A} \qquad \frac{\Gamma \vdash t : \bigcirc A \qquad \Gamma, \checkmark, \Gamma' \vdash}{\Gamma, \checkmark, \Gamma' \vdash \textsf{adv}\, t : A} \qquad \frac{\Gamma \vdash t : \Box A \qquad \textsf{token-free}(\Gamma')}{\Gamma, \sharp, \Gamma' \vdash \textsf{unbox}\, t : A}$$

$$\frac{\Gamma, \sharp \vdash t : A}{\Gamma \vdash \textsf{box}\, t : \Box A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma, \checkmark, \Gamma' \vdash \qquad A\ \textsf{stable}}{\Gamma, \checkmark, \Gamma' \vdash \textsf{progress}\, t : A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma, \sharp, \Gamma' \vdash \qquad A\ \textsf{stable}}{\Gamma, \sharp, \Gamma' \vdash \textsf{promote}\, t : A}$$

$$\frac{\Gamma \vdash t : A[\bigcirc(\mu\alpha.A)/\alpha]}{\Gamma \vdash \textsf{into}\, t : \mu\alpha.A} \qquad \frac{\Gamma \vdash t : \mu\alpha.A}{\Gamma \vdash \textsf{out}\, t : A[\bigcirc(\mu\alpha.A)/\alpha]} \qquad \frac{\Gamma, \sharp, x : \bigcirc A \vdash t : A}{\Gamma \vdash \textsf{fix}\, x.t : \Box A}$$

Operational Semantics (selected rules):

$$\frac{}{\langle v;\sigma\rangle \Downarrow \langle v;\sigma\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \left\langle \lambda x.s;\sigma'\right\rangle \qquad \left\langle t';\sigma'\right\rangle \Downarrow \left\langle v;\sigma''\right\rangle \qquad \left\langle s[v/x];\sigma''\right\rangle \Downarrow \left\langle v';\sigma'''\right\rangle}{\left\langle t\,t';\sigma\right\rangle \Downarrow \left\langle v';\sigma'''\right\rangle}$$

$$\frac{\sigma \neq \bot \qquad l = \mathbf{alloc}\,(\sigma)}{\langle \mathbf{delay}\ t;\sigma\rangle \Downarrow \langle l;\sigma, l \mapsto t\rangle} \qquad \frac{\langle t;\sharp\eta_N\rangle \Downarrow \left\langle l;\sharp\eta'_N\right\rangle \qquad \left\langle \eta'_N(l);\sharp\eta'_N\surd\eta_L\right\rangle \Downarrow \left\langle v;\sigma'\right\rangle}{\langle \mathbf{adv}\ t;\sharp\eta_N\surd\eta_L\rangle \Downarrow \left\langle v;\sigma'\right\rangle}$$

$$\frac{\langle t;\bot\rangle \Downarrow \langle v;\bot\rangle \qquad \sigma \neq \bot}{\langle \mathbf{promote}\ t;\sigma\rangle \Downarrow \langle v;\sigma\rangle} \qquad \frac{\langle t;\sharp\eta_N\rangle \Downarrow \left\langle v;\sharp\eta'_N\right\rangle}{\langle \mathbf{progress}\ t;\sharp\eta_N\surd\eta_L\rangle \Downarrow \left\langle v;\sharp\eta'_N\surd\eta_L\right\rangle}$$

$$\frac{\langle t;\bot\rangle \Downarrow \left\langle \mathbf{box}\,t';\bot\right\rangle \qquad \left\langle t';\sigma\right\rangle \Downarrow \left\langle v;\sigma'\right\rangle \qquad \sigma \neq \bot}{\langle \mathbf{unbox}\ t;\sigma\rangle \Downarrow \left\langle v;\sigma'\right\rangle} \qquad \frac{\langle t;\sigma\rangle \Downarrow \left\langle v;\sigma'\right\rangle}{\langle \mathbf{into}\ t;\sigma\rangle \Downarrow \left\langle \mathbf{into}\ v;\sigma'\right\rangle}$$

$$\frac{\langle t;\sigma\rangle \Downarrow \left\langle \mathbf{into}\ v;\sigma'\right\rangle}{\langle \mathbf{out}\ t;\sigma\rangle \Downarrow \left\langle v;\sigma'\right\rangle}$$

$$\frac{\langle t;\bot\rangle \Downarrow \left\langle \mathbf{fix}\ x.t';\bot\right\rangle \qquad \left\langle t'[l/x];\sigma, l \mapsto \mathbf{unbox}(\mathbf{fix}\ x.t')\right\rangle \Downarrow \left\langle v;\sigma'\right\rangle \qquad \sigma \neq \bot \qquad l = \mathbf{alloc}\,(\sigma)}{\langle \mathbf{unbox}\ t;\sigma\rangle \Downarrow \left\langle v;\sigma'\right\rangle}$$

SafeNats

$$\mathit{from} : \Box(\mathsf{Nat} \to \mathsf{Str\ Nat})$$

$$\mathit{from} = \mathsf{fix}\ f.\lambda(n : \mathsf{Nat}).n :: \mathsf{delay}((\mathsf{adv}\ f)\ (\mathsf{progress}\ n))$$

$$\mathit{nats} : \Box(\mathsf{Str\ Nat})$$

$$\mathit{nats} = \mathsf{box}(\mathsf{unbox}(\mathit{from})\ \mathsf{promote}(0))$$

LeakyNats

$$
\begin{aligned}
& \langle \mathsf{unbox}\ \mathsf{leakyNats}; \emptyset \rangle \\
\xRightarrow{\bar{0}}\ & \left\langle \mathsf{adv}\ l_1';\ l_1 \mapsto \mathsf{unbox}\ \mathsf{leakyNats},\ l_1' \mapsto \mathsf{unbox}\ \mathsf{map}\,(\mathsf{box}\,\lambda x.x+\bar{1})\,(\mathsf{adv}\ l_1) \right\rangle \\
\xRightarrow{\bar{1}}\ & \left\langle \mathsf{adv}\ l_2^3;\ \begin{array}{ll} l_2^0 \mapsto \mathsf{unbox}\ \mathsf{leakyNats}, & l_2^1 \mapsto \mathsf{unbox}\ \mathsf{map}\,(\mathsf{box}\,\lambda x.x+\bar{1})\,(\mathsf{adv}\ l_2^0), \\ l_2^2 \mapsto \mathsf{unbox}\ \mathsf{step}, & l_2^3 \mapsto \mathsf{adv}\ l_2^2\,(\mathsf{adv}\,(\mathsf{tail}\,(\bar{0} :: l_2^1))) \end{array} \right\rangle \\
\xRightarrow{\bar{2}}\ & \left\langle \mathsf{adv}\ l_3^5;\ \begin{array}{ll} l_3^0 \mapsto \mathsf{unbox}\ \mathsf{leakyNats}, & l_3^1 \mapsto \mathsf{unbox}\ \mathsf{map}\,(\mathsf{box}\,\lambda x.x+\bar{1})\,(\mathsf{adv}\ l_3^0), \\ l_3^2 \mapsto \mathsf{unbox}\ \mathsf{step}, & l_3^3 \mapsto \mathsf{adv}\ l_3^2\,(\mathsf{adv}\,(\mathsf{tail}\,(\bar{0} :: l_3^1))) \\ l_3^4 \mapsto \mathsf{unbox}\ \mathsf{step}, & l_3^5 \mapsto \mathsf{adv}\ l_3^4\,(\mathsf{adv}\,(\mathsf{tail}\,(\bar{1} :: l_3^3))) \end{array} \right\rangle \\
& \vdots
\end{aligned}
$$

where $\mathsf{step} = \mathsf{fix}\, f.\lambda\, s.\mathsf{unbox}\,(\mathsf{box}\,\lambda\, n.n+\bar{1})\,(\mathsf{head}\ s) :: (f \circledast \mathsf{tail}\ s)$.