Simply `RaTT`

A Fitch-Style Modal Calculus for Reactive Programming Without Space Leaks.

Patrick Bahr, **Christian Graulund**, Rasmus Møgelberg

IT University of Copenhagen

ICFP 2019

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 *(e.g. Ethereum)*.

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 *(e.g. Ethereum)*.

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] uses datatypes that represents values *"over time"*.
- Values that vary over time are called signals.
- Programs are signal transducers:

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

One implementation is signals as streams:

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

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	$\cdots$

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

Plugging a Leak

The problem is a stream is not stable over time and can lead to leaks.

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

To avoid leaks, we should only push stable values into the future but they can't 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]

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]

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 $\bigcirc$ pronounced *"Later"* or *"Delay"*.
- $\bigcirc A$ denotes *"A one time step from now"*.
- We now work with guarded streams:

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

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

Modal FRP

Modal FRP[3] = FRP + modal types

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

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

Ensures Causality

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

- 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

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

Examples with Simplified Syntax

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

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

Examples with Simplified Syntax

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

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

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

Type System

Let considered harmful

Traditional: Dual contexts

$$\frac{\overbrace{\Theta}^{\textit{now}} \mid \overbrace{\emptyset}^{\textit{later}} \vdash t : A}{\underbrace{\Gamma}_{\textit{now}} \mid \underbrace{\Theta}_{\textit{later}} \vdash \mathsf{delay}(t) : \bigcirc A}$$

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

$$\frac{}{\Gamma, x : A, \Gamma' \mid \Theta \vdash x : A}$$

Let considered harmful

Traditional: Dual contexts

$$\frac{\overbrace{\Theta}^{\textit{now}} \mid \overbrace{\emptyset}^{\textit{later}} \vdash t : A}{\underbrace{\Gamma}_{\textit{now}} \mid \underbrace{\Theta}_{\textit{later}} \vdash \mathsf{delay}(t) : \bigcirc A}$$

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

$$\frac{}{\Gamma, x : A, \Gamma' \mid \Theta \vdash x : A}$$

Modern: Fitch-style

$$\frac{\Gamma, \checkmark \vdash t : A}{\Gamma \vdash \mathsf{delay}(t) : \bigcirc A}$$

$$\frac{\overbrace{\Gamma}^{\textit{now}} \vdash t : \bigcirc A}{\underbrace{\Gamma}_{\textit{earlier}}, \checkmark, \underbrace{\Gamma'}_{\textit{now}} \vdash \mathsf{adv}(t) : A}$$

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

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, $\square A$.
- $\square A$ is a *stable type.*

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

Nakano Style Recursion

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

$$\frac{\Gamma, \sharp, x : \bigcirc A \vdash t : A}{\Gamma \vdash \mathrm{fix}\; x.t : \Box A}$$

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

In general we don't have $A \to \bigcirc A$, but we do have $\Box A \to \bigcirc \Box A$.

Examples

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

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

Examples

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

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

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

Examples

$$(\circledast) : \bigcirc (A \to B) \to \bigcirc A \to \bigcirc B$$
$$f \circledast a = \text{delay}\,((\text{adv}\ f)\ (\text{adv}\ a))$$

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

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

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

Eliminating Space Leaks

Big Step Operational Semantics

We define a heap based big step operational semantics:

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

The store σ is used for

- Allocation of delayed computation.
- Recursive calls.
- Input data for tranducers.

We define a small step semantics for streams:

$$\frac{\langle t : \mathsf{Str}\ A; \sharp\eta\checkmark\rangle \Downarrow \langle v :: l; \sharp\eta_N\checkmark\,\eta_L\rangle}{\langle t; \sharp\eta\rangle \xRightarrow{v} \langle \mathsf{adv}(l); \sharp\eta_L\rangle}$$

- After evaluation we delete the entire “now” heap η_N.
- Similar semantics for stream transducers.
- For transducers, *the input stream is allocated on the heap.*
- Theorem: Evaluation of streams and transducers is safe.

Example Reduction

Given the stream of natural numbers, we have the general big step reduction:

$$\langle nats_n; \sigma \rangle \Downarrow \langle n :: l_n; \sigma, l_n \mapsto nats_{n+1} \rangle$$

where $nats_n$ is the nth unfolding of nat.

Example Reduction

Given the stream of natural numbers, we have the general big step reduction:

$$\langle nats_n; \sigma\rangle \Downarrow \langle n :: l_n; \sigma, l_n \mapsto nats_{n+1}\rangle$$

where $nats_n$ is the nth unfolding of nat.

This give the reduction:

$$\begin{aligned}
\langle nats_0; \sharp\rangle &\overset{0}{\Longrightarrow} \langle \mathsf{adv}(l_0); \sharp l_0 \mapsto nats_1\rangle \\
&\overset{1}{\Longrightarrow} \langle \mathsf{adv}(l_1); \sharp l_1 \mapsto nats_2\rangle \\
&\overset{2}{\Longrightarrow} \langle \mathsf{adv}(l_2); \sharp l_2 \mapsto nats_3\rangle \\
&\;\vdots \\
&\overset{n}{\Longrightarrow} \langle \mathsf{adv}(l_n); \sharp l_n \mapsto nats_{n+1}\rangle
\end{aligned}$$

Summary and Future Work

Summary and Future Work

Summary

- Fitch-style approach to Modal FRP.
- Heap-based operational semantics that rules out space leaks.
- Type system that ensures safety, causality and productivity.
- Rule out (some) time leaks.
- Formalized meta-theory in Coq.

Summary and Future Work

Summary

- Fitch-style approach to Modal FRP.
- Heap-based operational semantics that rules out space leaks.
- Type system that ensures safety, causality and productivity.
- Rule out (some) time leaks.
- Formalized meta-theory in Coq.

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.

$$\mathsf{leakyNats} : \mathsf{Str\ Nat}$$

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

- We (roughly) have the unfolding:

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

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

Time Leaks II

- Solution: Disallow fixed points under delay.

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

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

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

$$\mathrm{nats} : \Box(\mathrm{Str\ Nat}) \qquad \mathrm{from} : \Box(\mathrm{Nat} \to \mathrm{Str}\ Nat)$$
$$\mathrm{nats} = \mathrm{from} \boxdot 0 \qquad \mathrm{from}\, n = n :: \mathrm{f} \odot (n+1)$$

Stream Transducer:

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

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

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

Typing Rules:

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

$$\frac{\Gamma \vdash s : \mathsf{Nat} \qquad \Gamma \vdash t : \mathsf{Nat}}{\Gamma \vdash s + t : \mathsf{Nat}} \qquad \frac{\Gamma, x : A \vdash t : B \qquad \text{tick-free}(\Gamma)}{\Gamma \vdash \lambda x.t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash t' : A}{\Gamma \vdash t\, t' : B}$$

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

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

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

$$\frac{\Gamma \vdash t : A \qquad \Gamma, \checkmark, \Gamma' \vdash \qquad A\ \text{stable}}{\Gamma, \checkmark, \Gamma' \vdash \mathsf{progress}\ t : A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma, \sharp, \Gamma' \vdash \qquad A\ \text{stable}}{\Gamma, \sharp, \Gamma' \vdash \mathsf{promote}\ t : A}$$

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

Operational Semantics (selected rules):

$$\frac{}{\langle v; \sigma \rangle \Downarrow \langle v; \sigma \rangle}$$

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

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

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

$$\frac{\langle t; \bot \rangle \Downarrow \langle v; \bot \rangle \qquad \sigma \neq \bot}{\langle \mathbf{promote}\; t; \sigma \rangle \Downarrow \langle v; \sigma \rangle}$$

$$\frac{\langle t; \sharp\eta_N \rangle \Downarrow \left\langle v; \sharp\eta'_N \right\rangle}{\langle \mathbf{progress}\; t; \sharp\eta_N \checkmark \eta_L \rangle \Downarrow \left\langle v; \sharp\eta'_N \checkmark \eta_L \right\rangle}$$

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

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

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

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

SafeNats

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

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

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

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

LeakyNats

$$
\begin{array}{ll}
 & \langle \mathsf{unbox\ leakyNats}; \emptyset \rangle \\[1em]
\overset{\overline{0}}{\Longrightarrow} & \left\langle \mathsf{adv}\ l_1';\ l_1 \mapsto \mathsf{unbox\ leakyNats},\ l_1' \mapsto \mathsf{unbox\ map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\ l_1) \right\rangle \\[1.5em]
\overset{\overline{1}}{\Longrightarrow} & \left\langle \mathsf{adv}\ l_2^3;\ \begin{array}{ll} l_2^0 \mapsto \mathsf{unbox\ leakyNats}, & l_2^1 \mapsto \mathsf{unbox\ map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\ l_2^0), \\ l_2^2 \mapsto \mathsf{unbox\ step}, & l_2^3 \mapsto \mathsf{adv}\ l_2^2\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{0} :: l_2^1))) \end{array} \right\rangle \\[2em]
\overset{\overline{2}}{\Longrightarrow} & \left\langle \mathsf{adv}\ l_3^5;\ \begin{array}{ll} l_3^0 \mapsto \mathsf{unbox\ leakyNats}, & l_3^1 \mapsto \mathsf{unbox\ map}\,(\mathsf{box}\,\lambda x.x + \overline{1})\,(\mathsf{adv}\ l_3^0), \\ l_3^2 \mapsto \mathsf{unbox\ step}, & l_3^3 \mapsto \mathsf{adv}\ l_3^2\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{0} :: l_3^1))) \\ l_3^4 \mapsto \mathsf{unbox\ step}, & l_3^5 \mapsto \mathsf{adv}\ l_3^4\,(\mathsf{adv}\,(\mathsf{tail}\,(\overline{1} :: l_3^3))) \end{array} \right\rangle \\
\vdots &
\end{array}
$$

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