Reversibilization in Functional and Concurrent Programming

Germán Vidal

VRAIN

Valencian Research Institute for Artificial Intelligence

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

PPDP-LOPSTR 2019

A collaborative work...

Naoki Nishida (Nagoya University)

Ivan Lanese (University of Bologna)

Adrián Palacios (Universitat Politecnica de Valencia)

COST action IC1405 on Reversible Computation

Reversible programming languages

Each execution step is **reversible**

Backward steps must be **deterministic**

E.g., **Janus**: `if` c_1 `then` s_1 `else` s_2 `fi` c_2

Reversible languages are not universal
(e.g., cannot compute non-injective functions)

Reversible programming languages

Each execution step is **reversible**

Backward steps must be **deterministic**

E.g., **Janus**: `if` c_1 `then` s_1 `else` s_2 `fi` c_2

Reversible languages are not universal
(e.g., cannot compute non-injective functions)

Reversibilization

Given an irreversible programming language **L** with semantics **Sem** over states $s_0, s_1, \ldots, s_n \in \textbf{State}$:

$$s_0 \to s_1 \to \ldots \to s_n$$

we can extend the states with enough information so that $\textbf{Sem}^R$ over $\langle s_0, h_0 \rangle, \langle s_1, h_1 \rangle, \ldots, \langle s_n, h_n \rangle \in \textbf{State}'$:

$$\langle s_0, [\,] \rangle \to \langle s_1, [s_0] \rangle \to \ldots \to \langle s_n, [s_{n-1}, \ldots, s_0] \rangle$$

becomes reversible

This is known as a Landauer embedding and is the main technique for **reversibilization**

Reversibilization

Given an irreversible programming language **L** with semantics **Sem** over states $s_0, s_1, \ldots, s_n \in$ **State**:

$$s_0 \rightarrow s_1 \rightarrow \ldots \rightarrow s_n$$

we can extend the states with enough information so that $\mathbf{Sem}^R$ over $\langle s_0, h_0\rangle, \langle s_1, h_1\rangle, \ldots, \langle s_n, h_n\rangle \in \mathbf{State}'$:

$$\langle s_0, [\,]\rangle \rightarrow \langle s_1, [s_0]\rangle \rightarrow \ldots \rightarrow \langle s_n, [s_{n-1}, \ldots, s_0]\rangle$$

becomes reversible

This is known as a Landauer embedding and is the main technique for **reversibilization**

It may seem impractical at first. . .

However,

- in some cases, performance is not critical (e.g., debugging)
- in some other cases, the history can be optimized (e.g., store nothing when applying an injective function)
- . . .

It may seem impractical at first…

However,

- in some cases, performance is not critical (e.g., debugging)
- in some other cases, the history can be optimized (e.g., store nothing when applying an injective function)
- …

Functional Programming

A first-order, eager functional language

Defining a Landauer embedding

Functions defined by pattern-matching, e.g.,

$$
\begin{array}{rcl}
\mathit{add}(0, y) & \to & y \\
\mathit{add}(\mathrm{s}(x), y) & \to & \mathrm{s}(\mathit{add}(x, y)) \\
\mathit{fst}(x, y) & \to & x
\end{array}
$$

An example reduction:

$$\mathit{fst}(\underline{\mathit{add}(\mathrm{s}(0), 0)}, 0) \to \mathit{fst}(\mathrm{s}(\underline{\mathit{add}(0, 0)}), 0) \to \underline{\mathit{fst}(\mathrm{s}(0), 0)} \to \mathrm{s}(0)$$

Defining a Landauer embedding

Functions defined by pattern-matching, e.g.,

$$\begin{array}{rcl} add(0, y) & \leftarrow & y \\ add(s(x), y) & \leftarrow & s(add(x, y)) \\ fst(x, y) & \leftarrow & x \end{array}$$

An example reduction:

$$\boxed{fst(add(s(0), 0), 0) \leftarrow fst(s(add(0, 0)), 0)} \rightarrow fst(s(0), 0) \rightarrow s(0)$$

What should include a Landauer embedding?

Defining a Landauer embedding

Functions defined by pattern-matching, e.g.,

$$\begin{array}{rcl} \mathit{add}(0, y) & \leftarrow & y \\ \mathit{add}(\mathsf{s}(x), y) & \leftarrow & \mathsf{s}(\mathit{add}(x, y)) \\[1ex] \mathit{fst}(x, y) & \leftarrow & x \end{array}$$

An example reduction:

$$\mathit{fst}(\mathit{add}(\mathsf{s}(0), 0), 0) \leftarrow \mathit{fst}(\underline{\mathsf{s}(\mathit{add}(0, 0))}, 0) \rightarrow \mathit{fst}(\mathsf{s}(0), 0) \rightarrow \mathsf{s}(0)$$

What should include a Landauer embedding?

$\Rightarrow$ position of reduced expression

Defining a Landauer embedding

Functions defined by pattern-matching, e.g.,

$$\begin{array}{rrcl} \beta_1 : & add(0, y) & \leftarrow & y \\ \beta_2 : & add(s(x), y) & \leftarrow & s(add(x, y)) \\ \beta_3 : & fst(x, y) & \leftarrow & x \end{array}$$

An example reduction:

$$fst(add(s(0), 0), 0) \leftarrow fst(\underline{s(add(0, 0))}, 0) \rightarrow fst(s(0), 0) \rightarrow s(0)$$

What should include a Landauer embedding?

$\Rightarrow$ position of reduced expression, rule

Defining a Landauer embedding

Functions defined by pattern-matching, e.g.,

$$
\begin{array}{rrcl}
\beta_1: & \mathit{add}(0, y) & \leftarrow & y \\
\beta_2: & \mathit{add}(\mathsf{s}(x), y) & \leftarrow & \mathsf{s}(\mathit{add}(x, y)) \\
\beta_3: & \mathit{fst}(x, \boxed{y}) & \leftarrow & x
\end{array}
$$

An example reduction:

$$
\mathit{fst}(\mathit{add}(\mathsf{s}(0), 0), 0) \leftarrow \mathit{fst}(\mathsf{s}(\mathit{add}(0, 0)), 0) \leftarrow \boxed{\mathit{fst}(\mathsf{s}(0), 0) \leftarrow \mathsf{s}(0)}
$$

What should include a Landauer embedding?

$\Rightarrow$ position of reduced expression, rule

Defining a Landauer embedding

Functions defined by pattern-matching, e.g.,

$$\begin{array}{lrcl} \beta_1: & add(0, y) & \leftarrow & y \\ \beta_2: & add(s(x), y) & \leftarrow & s(add(x, y)) \\ \beta_3: & fst(x, y) & \leftarrow & x \end{array}$$

An example reduction:

$$fst(add(s(0), 0), 0) \leftarrow fst(s(add(0, 0)), 0) \leftarrow fst(s(0), 0) \leftarrow s(0)$$

What should include a Landauer embedding?

$\Rightarrow$ position of reduced expression, rule, erased values

We store a trace term $\beta(p,\sigma)$ at every reduction step:

$$
\begin{array}{l}
\langle \mathit{fst}(\mathit{add}(\mathsf{s}(0),0),0),[\,]\rangle \\
\qquad \rightharpoonup \langle \mathit{fst}(\mathsf{s}(\mathit{add}(0,0)),0),[\beta_2(1,\mathit{id})]\rangle \\
\qquad \rightharpoonup \langle \mathit{fst}(\mathsf{s}(0),0),[\beta_1(1.1,\mathit{id}),\beta_2(1,\mathit{id})]\rangle \\
\qquad \rightharpoonup \langle \mathsf{s}(0),[\beta_3(\epsilon,\{y\mapsto 0\}),\beta_1(1.1,\mathit{id}),\beta_2(1,\mathit{id})]\rangle
\end{array}
$$

where

- $\rightharpoonup$ is the reversible *forward* reduction relation
- $\leftharpoondown$ is the reversible *backward* reduction relation

We store a trace term $\beta(p,\sigma)$ at every reduction step:

$$
\begin{array}{l}
\langle \mathit{fst}(\mathit{add}(\mathsf{s}(0),0),0), [\,]\rangle \\
\qquad \rightharpoonup \langle \mathit{fst}(\mathsf{s}(\mathit{add}(0,0)),0), [\beta_2(1,\mathit{id})]\rangle \\
\qquad \rightharpoonup \langle \mathit{fst}(\mathsf{s}(0),0), [\beta_1(1.1,\mathit{id}),\beta_2(1,\mathit{id})]\rangle \\
\qquad \rightharpoonup \langle \mathsf{s}(0), [\beta_3(\epsilon,\{y\mapsto 0\}),\beta_1(1.1,\mathit{id}),\beta_2(1,\mathit{id})]\rangle
\end{array}
$$

where

- $\rightharpoonup$ is the reversible *forward* reduction relation
- $\leftharpoondown$ is the reversible *backward* reduction relation

Can we move from the instrumented semantics to an instrumented program?

I.e., given a program $\mathcal{R}$, define $\mathcal{R}_f$ and $\mathcal{R}_b$ such that

$$\langle s_1, \pi_1 \rangle \rightharpoonup_{\mathcal{R}} \langle s_2, \pi_2 \rangle \quad \text{iff} \quad \langle s_1, \pi_1 \rangle \rightarrow_{\mathcal{R}_f} \langle s_2, \pi_2 \rangle$$

and

$$\langle s_2, \pi_2 \rangle \leftharpoondown_{\mathcal{R}} \langle s_1, \pi_1 \rangle \quad \text{iff} \quad \langle s_2, \pi_2 \rangle \rightarrow_{\mathcal{R}_b} \langle s_1, \pi_1 \rangle$$

Instrumenting the rules to store the applied rule and the erased values is easy (static)

…but storing positions is rather difficult (dynamic)

Alternative: program transformation (flattening), e.g.,

$$\begin{array}{rcl} add(0,y) & \to & y \\ add(s(x),y) & \to & s(add(x,y)) \end{array}$$

$$\Downarrow$$

$$\begin{array}{rcl} add(0,y) & \to & y \\ add(s(x),y) & \to & s(z) \Leftarrow add(x,y) \twoheadrightarrow z \end{array}$$

so that all function calls occur at root positions

Instrumenting the rules to store the applied rule and the erased values is easy (static)

...but storing positions is rather difficult (dynamic)

Alternative: program transformation (flattening), e.g.,

$$\begin{array}{rcl} add(0,y) & \rightarrow & y \\ add(\mathsf{s}(x),y) & \rightarrow & \mathsf{s}(add(x,y)) \end{array}$$

$$\Downarrow$$

$$\begin{array}{rcl} add(0,y) & \rightarrow & y \\ add(\mathsf{s}(x),y) & \rightarrow & \mathsf{s}(z) \Leftarrow add(x,y) \twoheadrightarrow z \end{array}$$

so that all function calls occur at root positions

Thus we can get rid of positions in trace terms...

$$\beta(p, \sigma) \;\Rightarrow\; \beta(\sigma)$$

Conditional Programs

A conditional rule:

$$f(s_0) \to r \Leftarrow f_1(s_1) \twoheadrightarrow t_1, \ldots, f_n(s_n) \twoheadrightarrow t_n$$

is equivalent to (Haskell-like):

$$f\ s_0 = r \textbf{ where } t_1 = f_1\ s_1, \ldots, t_n = f_n\ s_n$$

or

$$f\ s_0 = \textbf{let } t_1 = f_1\ s_1, \ldots, t_n = f_n\ s_n \textbf{ in } r$$

E.g.,

$$\begin{array}{rcl} add\ 0\ y & = & y \\ add\ (\mathsf{s}\ x)\ y & = & \textbf{let } z = add\ x\ y \textbf{ in } \mathsf{s}(z) \end{array}$$

Conditional Programs

A conditional rule:

$$f(s_0) \to r \Leftarrow f_1(s_1) \twoheadrightarrow t_1, \ldots, f_n(s_n) \twoheadrightarrow t_n$$

is equivalent to (Haskell-like):

$$f\ s_0 = r\ \textbf{where}\ t_1 = f_1\ s_1, \ldots, t_n = f_n\ s_n$$

or

$$f\ s_0 = \textbf{let}\ t_1 = f_1\ s_1, \ldots, t_n = f_n\ s_n\ \textbf{in}\ r$$

E.g.,

$$\begin{array}{rcl} add\ 0\ y & = & y \\ add\ (\mathrm{s}\ x)\ y & = & \textbf{let}\ z = add\ x\ y\ \textbf{in}\ \mathrm{s}(z) \end{array}$$

Injectivization & Inversion

Injectivization

We replace each rule

$$\beta : \mathsf{f}(s_0) \to r \Leftarrow \mathsf{f}_1(s_1) \twoheadrightarrow t_1, \ldots, \mathsf{f}_\mathsf{n}(s_n) \twoheadrightarrow t_n$$

by a new rule of the form

$$\mathsf{f}^i(s_0) \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow \mathsf{f}^i_1(s_1) \twoheadrightarrow \langle t_1, w_1\rangle, \ldots, \mathsf{f}^i_\mathsf{n}(s_n) \twoheadrightarrow \langle t_n, w_n\rangle$$

where $\{\overline{y}\} = (\mathcal{V}ar(s_0) \backslash \mathcal{V}ar(r, s_n, t_n)) \cup \bigcup_{i=1}^{n} \mathcal{V}ar(t_i) \backslash \mathcal{V}ar(r, s_{i+1,n})$

Inversion

We replace each rule

$$\mathsf{f}^i(s_0) \to \langle r, \beta(\overline{y}, \overline{w_n})\rangle \Leftarrow \mathsf{f}^i_1(s_1) \twoheadrightarrow \langle t_1, w_1\rangle, \ldots, \mathsf{f}^i_\mathsf{n}(s_n) \twoheadrightarrow \langle t_n, w_n\rangle$$

by a new rule of the form

$$\mathsf{f}^{-1}(r, \beta(\overline{y}, \overline{w_n})) \to \langle s_0\rangle \Leftarrow \mathsf{f}_\mathsf{n}^{-1}(t_n, w_n) \twoheadrightarrow \langle s_n\rangle, \ldots, \mathsf{f}_1^{-1}(t_1, w_1) \twoheadrightarrow \langle s_1\rangle$$

Injectivization & Inversion: An example

$$
\begin{array}{rrcl}
\beta_1: & \mathrm{add}(0,y) & \to & y \\
\beta_2: & \mathrm{add}(\mathrm{s}(x),y) & \to & \mathrm{s}(x_1) \Leftarrow \mathrm{add}(x,y) \twoheadrightarrow x_1 \\
\beta_3: & \mathrm{fst}(x,y) & \to & x
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{add}^i(0,y) & \to & \langle y, \beta_1 \rangle \\
\mathrm{add}^i(\mathrm{s}(x),y) & \to & \langle \mathrm{s}(x_1), \beta_2(w_1) \rangle \Leftarrow \mathrm{add}^i(x,y) \twoheadrightarrow \langle x_1, w_1 \rangle \\
\mathrm{fst}^i(x,y) & \to & \langle x, \beta_3(y) \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathrm{add}^{-1}(y, \beta_1) & \to & \langle 0, y \rangle \\
\mathrm{add}^{-1}(\mathrm{s}(x_1), \beta_2(w_1)) & \to & \langle \mathrm{s}(x), y \rangle \Leftarrow \mathrm{add}^{-1}(x_1, w_1) \twoheadrightarrow \langle x, y \rangle \\
\mathrm{fst}^{-1}(x, \beta_3(y)) & \to & \langle x, y \rangle
\end{array}
$$

Injectivization & Inversion: An example

$$\begin{array}{rrcl}
\beta_1: & \mathsf{add}(0,y) & \to & y \\
\beta_2: & \mathsf{add}(\mathsf{s}(x),y) & \to & \mathsf{s}(x_1) \Leftarrow \mathsf{add}(x,y) \twoheadrightarrow x_1 \\
\beta_3: & \mathsf{fst}(x,y) & \to & x
\end{array}$$

$$\begin{array}{rcl}
\mathsf{add}^i(0,y) & \to & \langle y,\beta_1\rangle \\
\mathsf{add}^i(\mathsf{s}(x),y) & \to & \langle \mathsf{s}(x_1),\beta_2(w_1)\rangle \Leftarrow \mathsf{add}^i(x,y) \twoheadrightarrow \langle x_1,w_1\rangle \\
\mathsf{fst}^i(x,y) & \to & \langle x,\beta_3(y)\rangle
\end{array}$$

$$\begin{array}{rcl}
\mathsf{add}^{-1}(y,\beta_1) & \to & \langle 0,y\rangle \\
\mathsf{add}^{-1}(\mathsf{s}(x_1),\beta_2(w_1)) & \to & \langle \mathsf{s}(x),y\rangle \Leftarrow \mathsf{add}^{-1}(x_1,w_1) \twoheadrightarrow \langle x,y\rangle \\
\mathsf{fst}^{-1}(x,\beta_3(y)) & \to & \langle x,y\rangle
\end{array}$$

Injectivization & Inversion: An example

$$
\begin{array}{rrcl}
\beta_1: & \mathsf{add}(0,y) & \to & y \\
\beta_2: & \mathsf{add}(\mathsf{s}(x),y) & \to & \mathsf{s}(x_1) \Leftarrow \mathsf{add}(x,y) \twoheadrightarrow x_1 \\
\beta_3: & \mathsf{fst}(x,y) & \to & x
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{add}^i(0,y) & \to & \langle y, \beta_1 \rangle \\
\mathsf{add}^i(\mathsf{s}(x),y) & \to & \langle \mathsf{s}(x_1), \beta_2(w_1) \rangle \Leftarrow \mathsf{add}^i(x,y) \twoheadrightarrow \langle x_1, w_1 \rangle \\
\mathsf{fst}^i(x,y) & \to & \langle x, \beta_3(y) \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{add}^{-1}(y,\beta_1) & \to & \langle 0, y \rangle \\
\mathsf{add}^{-1}(\mathsf{s}(x_1),\beta_2(w_1)) & \to & \langle \mathsf{s}(x), y \rangle \Leftarrow \mathsf{add}^{-1}(x_1,w_1) \twoheadrightarrow \langle x, y \rangle \\
\mathsf{fst}^{-1}(x,\beta_3(y)) & \to & \langle x, y \rangle
\end{array}
$$

More details in "Nishida, Palacios & Vidal: Reversible computation in term rewriting. *J. Log. Algebr. Meth. Program.* 94: 128-149 (2018)"

Application: bidirectionalization

Bidirectional transformations (Bx)

We have two data representations, called source and view with source $\supseteq$ view (assymmetric case)

Function get : source $\mapsto$ view

Consistency: $s \in$ source is consistent with $v \in$ view if $get(s) = v$

We accept updates in both the source and the view $\Rightarrow$ recover consistency!

Function put : view $\times$ source $\mapsto$ source

$$\begin{tikzcd} s \arrow[r, "get"] \arrow[d, "update"'] & v \\ s' \arrow[r, "get"] & v' \end{tikzcd}$$

$$\begin{tikzcd} s \arrow[r, "get"] \arrow[d, bend left] & v \arrow[d, "update"] \\ s' & v' \arrow[l, "put"] \end{tikzcd}$$

Bidirectional transformations (Bx)

We have two data representations, called source and view with source $\supseteq$ view (assymmetric case)

Function get : source $\mapsto$ view

Consistency: $s \in$ source is consistent with $v \in$ view if $get(s) = v$

We accept updates in both the source and the view $\Rightarrow$ recover consistency!

Function put : view $\times$ source $\mapsto$ source

$$\begin{tikzcd} s \arrow[r, bend left, "get"] \arrow[d, "update"'] & v \\ s' \arrow[r, bend left, "get"] & v' \end{tikzcd}$$

$$\begin{tikzcd} s \arrow[r, bend left, "get"] & v \arrow[d, "update"] \\ s' & v' \arrow[l, bend left, "put"] \end{tikzcd}$$

Bidirectional transformations (Bx)

We have two data representations, called source and view with source $\supseteq$ view (assymmetric case)

Function *get* : source $\mapsto$ view

Consistency: $s \in$ source is consistent with $v \in$ view if $get(s) = v$

We accept updates in both the source and the view $\Rightarrow$ recover consistency!

Function *put* : view $\times$ source $\mapsto$ source

$$\begin{tikzcd} s \arrow[r, bend left, "get"] \arrow[d, "update"'] & v \\ s' \arrow[r, bend left, "get"] & v' \end{tikzcd} \qquad \begin{tikzcd} s \arrow[r, bend left, "get"] \arrow[d, bend left=60] & v \arrow[d, "update"] \\ s' & v' \arrow[l, bend right, "put"'] \end{tikzcd}$$

Motivation

Defining the right "put" is not easy

$\Rightarrow$ (syntactic) **bidirectionalization**

Stepwise approach to bidirectionalization

Example (first names)

$$\begin{array}{l} \mathrm{fn}([\,]) \to [\,] \\ \mathrm{fn}(\mathrm{person}(n, l) : xs) \to n : ys \Leftarrow \mathrm{fn}(xs) \twoheadrightarrow ys \\ \mathrm{fn}(\mathrm{city}(c) : xs) \to ys \Leftarrow \mathrm{fn}(xs) \twoheadrightarrow ys \end{array}$$

E.g., given
$s = [\mathrm{person}(\mathrm{john}, \mathrm{smith}), \mathrm{city}(\mathrm{london}), \mathrm{person}(\mathrm{ada}, \mathrm{lovelace})]$,
we have $\mathrm{fn}(s) = [\mathrm{john}, \mathrm{ada}]$

Stepwise approach to bidirectionalization

Example (first names, injective version)

$$\mathsf{fn}^{\mathsf{i}}([\,]) \to \langle [\,], \beta_1 \rangle$$
$$\mathsf{fn}^{\mathsf{i}}(\mathsf{person}(n, l) : xs) \to \langle n : ys, \beta_2(l, w) \rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w \rangle$$
$$\mathsf{fn}^{\mathsf{i}}(\mathsf{city}(c) : xs) \to \langle ys, \beta_3(c, w) \rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w \rangle$$

E.g., given $s =$ $[\mathsf{person}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{person}(\mathsf{ada}, \mathsf{lovelace})]$, we have

$$\mathsf{fn}^{\mathsf{i}}(s) = \langle [\mathsf{john}, \mathsf{ada}], \underbrace{\beta_2(\mathsf{smith}, \beta_3(\mathit{london}, \beta_2(\mathit{lovelace}, \beta_1)))}_{\text{a complement!}} \rangle$$

(according to [Bancilhon & Spyratos, 1981])

Stepwise approach to bidirectionalization

Example (first names, injective version)

$$
\begin{array}{l}
\mathsf{fn}^{\mathsf{i}}([\,]) \to \langle [\,], \beta_1 \rangle \\
\mathsf{fn}^{\mathsf{i}}(\mathsf{person}(n, l) : xs) \to \langle n : ys, \beta_2(l, w) \rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w \rangle \\
\mathsf{fn}^{\mathsf{i}}(\mathsf{city}(c) : xs) \to \langle ys, \beta_3(c, w) \rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w \rangle
\end{array}
$$

E.g., given $s =$
$[\mathsf{person}(\mathsf{john}, \mathsf{smith}), \mathsf{city}(\mathsf{london}), \mathsf{person}(\mathsf{ada}, \mathsf{lovelace})]$,
we have

$$\mathsf{fn}^{\mathsf{i}}(s) = \langle [\mathsf{john}, \mathsf{ada}], \underbrace{\beta_2(\mathsf{smith}, \beta_3(\mathit{london}, \beta_2(\mathit{lovelace}, \beta_1)))}_{\text{a complement!}} \rangle$$

(according to [Bancilhon & Spyratos, 1981])

Stepwise approach to bidirectionalization

Example (first names, injective version)

$$\mathrm{fn}^{\mathrm{i}}([\,]) \to \langle [\,], \beta_1 \rangle$$
$$\mathrm{fn}^{\mathrm{i}}(\mathrm{person}(n, l) : xs) \to \langle n : ys, \beta_2(l, w) \rangle \Leftarrow \mathrm{fn}^{\mathrm{i}}(xs) \twoheadrightarrow \langle ys, w \rangle$$
$$\mathrm{fn}^{\mathrm{i}}(\mathrm{city}(c) : xs) \to \langle ys, \beta_3(c, w) \rangle \Leftarrow \mathrm{fn}^{\mathrm{i}}(xs) \twoheadrightarrow \langle ys, w \rangle$$

E.g., given $s =$ $[\mathrm{person}(\mathrm{john}, \mathrm{smith}), \mathrm{city}(\mathrm{london}), \mathrm{person}(\mathrm{ada}, \mathrm{lovelace})]$, we have

$$\mathrm{fn}^{\mathrm{i}}(s) = \langle [\mathrm{john}, \mathrm{ada}], \underbrace{\beta_2(\mathrm{smith}, \beta_3(\mathit{london}, \beta_2(\mathit{lovelace}, \beta_1)))}_{\text{a complement!}} \rangle$$

(according to [Bancilhon & Spyratos, 1981])

Stepwise approach to bidirectionalization

$$
\begin{array}{l}
\mathsf{fn}^{\mathsf{i}}([\,]) \to \langle [\,], \beta_1 \rangle \\
\mathsf{fn}^{\mathsf{i}}(\mathsf{person}(n,l):xs) \to \langle n:ys, \beta_2(l,w)\rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w\rangle \\
\mathsf{fn}^{\mathsf{i}}(\mathsf{city}(c):xs) \to \langle ys, \beta_3(c,w)\rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fn}^{-1}([\,], \beta_1) \to [\,] \\
\mathsf{fn}^{-1}(n:ys, \beta_2(l,w)) \to \mathsf{person}(n,l):xs \Leftarrow \mathsf{fn}^{-1}(ys,w) \twoheadrightarrow xs \\
\mathsf{fn}^{-1}(ys, \beta_3(c,w)) \to \mathsf{city}(c):xs \Leftarrow \mathsf{fn}^{-1}(ys,w) \twoheadrightarrow xs
\end{array}
$$

Generation of a “put” function (given a “get” function f):

$$
\mathsf{put}_{\mathsf{f}}(v,s) \to s' \Leftarrow \mathsf{f}^{\mathsf{i}}(s) \twoheadrightarrow \langle \_, \pi\rangle, \mathsf{f}^{-1}(v,\pi) \twoheadrightarrow s'
$$

Stepwise approach to bidirectionalization

$$\mathsf{fn}^{\mathsf{i}}([\,]) \to \langle [\,], \beta_1 \rangle$$
$$\mathsf{fn}^{\mathsf{i}}(\mathsf{person}(n, l) : xs) \to \langle n : ys, \beta_2(l, w) \rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w \rangle$$
$$\mathsf{fn}^{\mathsf{i}}(\mathsf{city}(c) : xs) \to \langle ys, \beta_3(c, w) \rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w \rangle$$

$$\mathsf{fn}^{-1}([\,], \beta_1) \to [\,]$$
$$\mathsf{fn}^{-1}(n : ys, \beta_2(l, w)) \to \mathsf{person}(n, l) : xs \Leftarrow \mathsf{fn}^{-1}(ys, w) \twoheadrightarrow xs$$
$$\mathsf{fn}^{-1}(ys, \beta_3(c, w)) \to \mathsf{city}(c) : xs \Leftarrow \mathsf{fn}^{-1}(ys, w) \twoheadrightarrow xs$$

Generation of a “put” function (given a “get” function f):

$$\mathsf{put}_{\mathsf{f}}(v, s) \to s' \Leftarrow \boxed{\mathsf{f}^{\mathsf{i}}(s) \twoheadrightarrow \langle \_, \pi \rangle}, \mathsf{f}^{-1}(v, \pi) \twoheadrightarrow s'$$

1. compute the complement of the original source

Stepwise approach to bidirectionalization

$$
\begin{array}{l}
\mathsf{fn}^{\mathsf{i}}([\,]) \to \langle [\,], \beta_1 \rangle \\
\mathsf{fn}^{\mathsf{i}}(\mathsf{person}(n, l) : xs) \to \langle n : ys, \beta_2(l, w) \rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w \rangle \\
\mathsf{fn}^{\mathsf{i}}(\mathsf{city}(c) : xs) \to \langle ys, \beta_3(c, w) \rangle \Leftarrow \mathsf{fn}^{\mathsf{i}}(xs) \twoheadrightarrow \langle ys, w \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fn}^{-1}([\,], \beta_1) \to [\,] \\
\mathsf{fn}^{-1}(n : ys, \beta_2(l, w)) \to \mathsf{person}(n, l) : xs \Leftarrow \mathsf{fn}^{-1}(ys, w) \twoheadrightarrow xs \\
\mathsf{fn}^{-1}(ys, \beta_3(c, w)) \to \mathsf{city}(c) : xs \Leftarrow \mathsf{fn}^{-1}(ys, w) \twoheadrightarrow xs
\end{array}
$$

Generation of a “put” function (given a “get” function f):

$$
\mathsf{put}_{\mathsf{f}}(v, s) \to \boxed{s'} \Leftarrow \mathsf{f}^{\mathsf{i}}(s) \twoheadrightarrow \langle \_, \pi \rangle, \boxed{\mathsf{f}^{-1}(v, \pi) \twoheadrightarrow s'}
$$

1. compute the complement of the original source
2. compute the updated source

Limitations

Given, $s =$
[person(john, smith), city(london), person(ada, lovelace)]
and
$\mathrm{fn}^{\mathrm{i}}(s) = \langle [\mathrm{john}, \mathrm{ada}], \underbrace{\beta_2(\mathrm{smith}, \beta_3(\mathit{london}, \beta_2(\mathit{lovelace}, \beta_1)))}_{\pi}\rangle$

Update 1 (compatible)

$[\mathrm{john}, \mathrm{ada}] \Rightarrow [\mathrm{peter}, \mathrm{ada}]\ (v_1)$
$\mathrm{fn}^{-1}(v_1, \pi) =$
[person(peter, smith), city(london), person(ada, lovelace)]

Update 2 (non-compatible)

$[\mathrm{john}, \mathrm{ada}] \Rightarrow [\mathrm{john}]\ (v_2)$
$\mathrm{fn}^{-1}(v_2, \pi)$ undefined

Limitations

Given, $s =$
$[\mathsf{person}(\mathsf{john},\mathsf{smith}),\mathsf{city}(\mathsf{london}),\mathsf{person}(\mathsf{ada},\mathsf{lovelace})]$
and
$\mathsf{fn}^{\mathsf{i}}(s) = \langle[\mathsf{john},\mathsf{ada}], \underbrace{\beta_2(\mathsf{smith},\beta_3(\mathit{london},\beta_2(\mathit{lovelace},\beta_1)))}_{\pi}\rangle$

Update 1 (compatible)

$[\mathsf{john},\mathsf{ada}] \Rightarrow [\mathsf{peter},\mathsf{ada}]\ (v_1)$
$\mathsf{fn}^{-1}(v_1,\pi) =$
$[\mathsf{person}(\mathsf{peter},\mathsf{smith}),\mathsf{city}(\mathsf{london}),\mathsf{person}(\mathsf{ada},\mathsf{lovelace})]$

Update 2 (non-compatible)

$[\mathsf{john},\mathsf{ada}] \Rightarrow [\mathsf{john}]\ (v_2)$
$\mathsf{fn}^{-1}(v_2,\pi)$ undefined

Characterizing compatible updates [RC 2019]

Definition (view skeleton)

Consider a source s with $\mathsf{f}^{\mathsf{i}}(s) = \langle v, \pi \rangle$

We compute the narrowing derivation:
$\mathsf{f}^{-1}(x, \pi) \leadsto^{*}_{\sigma} s'$ (deterministic!)

Then, $x\sigma = v'$ is the view skeleton

E.g.,
$\mathsf{fn}^{-1}(x, \beta_2(\mathsf{smith}, \beta_3(\mathit{london}, \beta_2(\mathit{lovelace}, \beta_1)))) \leadsto^{*}_{\{x \mapsto [x_1, x_2]\}} s'$
Therefore, the view skeleton is $[x_1, x_2]$

Consider a source s with $\mathsf{f}^{\mathsf{i}}(s) = \langle v, \pi \rangle$

An update v' is **compatible** if it is an instance of the view skeleton

Characterizing compatible updates [RC 2019]

Definition (view skeleton)

Consider a source s with $\mathsf{f}^{\mathsf{i}}(s) = \langle v, \pi \rangle$

We compute the narrowing derivation:
$\mathsf{f}^{-1}(x, \pi) \leadsto_{\sigma}^{*} s'$ (deterministic!)

Then, $x\sigma = v'$ is the view skeleton

E.g.,
$\mathsf{fn}^{-1}(x, \beta_2(\mathsf{smith}, \beta_3(\mathit{london}, \beta_2(\mathit{lovelace}, \beta_1)))) \leadsto_{\{x \mapsto [x_1, x_2]\}}^{*} s'$
Therefore, the view skeleton is $[x_1, x_2]$

Consider a source s with $\mathsf{f}^{\mathsf{i}}(s) = \langle v, \pi \rangle$

An update v' is **compatible** if it is an instance of the view skeleton

Ongoing work: non-compatible updates...

Concurrent Programming

A first-order, eager functional and concurrent language based on message-passing

The language

We consider a simple functional and concurrent programming language similar to **Erlang**

- No shared memory, only message passing (asynchronous communication)
- Each process has a pid and a local queue (mailbox)
- A system is a collection of processes

Sequential Erlang in 5 examples

append/2

```
append([H|T], L) -> [H|append(T,L)];
append([],    L) -> L.
```

Variables start with an uppercase letter

Function names and atoms (i.e., constants) start with a lowercase letter

Alternative definition:

append/2

```
append(A,B) -> case A of
                 [H|T]   -> [H|append(T,L)];
                 []      -> L
               end.
```

Sequential Erlang in 5 examples

append/2

```
append([H|T], L) -> [H|append(T,L)];
append([],    L) -> L.
```

Variables start with an uppercase letter

Function names and atoms (i.e., constants) start with a lowercase letter

Alternative definition:

append/2

```
append(A,B) -> case A of
                [H|T]   -> [H|append(T,L)];
                []      -> L
               end.
```

Sequential Erlang in 5 examples

toint/1

```
toint({s,N})  ->  int(N)+1;
toint(zero)   ->  0.
```

E.g., `toint({s,{s,{s,zero}}})` evaluates to `3`

No user-defined algebraic data types (so we cannot write `s(s(s(zero)))`)

Main data types: numbers, atoms, lists, and tuples

Sequential Erlang in 5 examples

factorial/1

```
factorial(N) when N > 0  ->  N * factorial(N - 1);
factorial(1)             ->  0.
```

Besides pattern matching, we can have guards

Only built-in functions are allowed in guards

Sequential Erlang in 5 examples

minmax/1

$$\begin{array}{lll} \texttt{minmax(L)} & \texttt{->} & \texttt{Min} = \texttt{lists:min(L)}, \\ & & \texttt{Max} = \texttt{lists:max(L)}, \\ & & \{\texttt{Min},\texttt{Max}\}. \end{array}$$

Sequence $e_1, \ldots, e_n$ evaluates all expressions, returns the evaluation of e_n

Equation $\texttt{pat} = \texttt{exp}$ evaluates $\texttt{exp}$ and perform pattern matching with $\texttt{pat}$

Equivalent to

$$\texttt{minmax(L)} \texttt{->} \{\texttt{Min},\texttt{Max}\} \Leftarrow \texttt{lists:min(L)} \twoheadrightarrow \texttt{Min}, \; \texttt{lists:max(L)} \twoheadrightarrow \texttt{Max}$$

Sequential Erlang in 5 examples

minmax/1

$$\begin{array}{lll} \mathtt{minmax(L)} & \mathtt{->} & \mathtt{Min = lists:min(L),} \\ & & \mathtt{Max = lists:max(L),} \\ & & \mathtt{\{Min,Max\}.} \end{array}$$

Sequence $e_1, \ldots, e_n$ evaluates all expressions, returns the evaluation of e_n

Equation $\mathtt{pat} = \mathtt{exp}$ evaluates $\mathtt{exp}$ and perform pattern matching with $\mathtt{pat}$

Equivalent to

$$\begin{array}{l} \mathtt{minmax(L)\ ->\ \{Min,Max\}} \Leftarrow \mathtt{lists:min(L)} \twoheadrightarrow \mathtt{Min}, \\ \qquad\qquad\qquad\qquad\qquad\quad \mathtt{lists:max(L)} \twoheadrightarrow \mathtt{Max} \end{array}$$

Sequential Erlang in 5 examples

inclist/1

```
inclist(L) -> lists:map(fun(X) -> X + 1 end, L).
```

Higher-order functions

Anonymous functions

No partial applications

Concurrency features

- spawn: creates a new process as a side-effect and returns the pid of the new process
- self: returns the pid of the current process
- pid ! val: sends val to process pid as a side-effect and returns val
- receive ... end: waits for a message that matches some pattern (otherwise, blocks execution) and returns the expression in the selected branch

Concurrent Erlang in 1 example


```
main()      ->  S = spawn(server([])),
                client(S).

client(S)   ->  S ! {self(), {add, paper}},
                S ! {self(), {add, pencil}},
                S ! {self(), take},
                receive
                  X -> X
                end.

server(L)   ->  receive
                  {_, {add, Item}} -> server([Item|L]);
                  {C, take} -> C ! hd(L), server(tl(L))
                end.
```

From Erlang to Core Erlang

Core Erlang is an intermediate representation used during the compilation of Erlang programs

It is a convenient representation for defining analyses and other tools

Not as readable as Erlang. . .

From Erlang to Core Erlang

erlang

```
a(42)  ->  ok;
a(N)   ->  M = N + 1, a(M).
```

core erlang

```
'a'/1 = fun(_@c0) ->
    case _@c0 of
      <42> when 'true' -> 'ok'
      <_@c2> when 'true' -> let <_@c3> = call 'erlang':'+'(N, 1)
                              in apply 'a'/1 (_@c3)
    end
```

Essentially: one clause per function, case for pattern matching, let for sequences, apply for function applications, ...

From Erlang to Core Erlang

erlang

```
a(42) -> ok;
a(N)  -> M = N + 1, a(M).
```

core erlang

```
'a'/1 = fun(_@c0) ->
     case _@c0 of
       < 42 > when 'true' -> 'ok'
       < _@c2 > when 'true' -> let < _@c3 >= call 'erlang':'+'(N, 1)
                                 in apply 'a'/1 (_@c3)
     end
```

Essentially: one clause per function, case for pattern matching, let for sequences, apply for function applications, ...

Core Erlang syntax

We consider a subset of Core Erlang with this syntax:

$$
\begin{array}{rcl}
\mathit{Module} & ::= & \mathrm{module}\ \mathit{Atom} = \mathit{fun}_1, \ldots, \mathit{fun}_n \\
\mathit{fun} & ::= & \mathit{fname} = \mathrm{fun}\ (X_1, \ldots, X_n) \to \mathit{expr} \\
\mathit{fname} & ::= & \mathit{Atom}/\mathit{Integer} \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid [\,] \\
\mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid [\mathit{expr}_1|\mathit{expr}_2] \mid \{\mathit{expr}_1, \ldots, \mathit{expr}_n\} \\
& \mid & \mathrm{call}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathrm{apply}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \\
& \mid & \mathrm{case}\ \mathit{expr}\ \mathrm{of}\ \mathit{clause}_1; \ldots; \mathit{clause}_m\ \mathrm{end} \\
& \mid & \mathrm{let}\ \mathit{Var} = \mathit{expr}_1\ \mathrm{in}\ \mathit{expr}_2 \mid \mathrm{receive}\ \mathit{clause}_1; \ldots; \mathit{clause}_n\ \mathrm{end} \\
& \mid & \mathrm{spawn}(\mathit{expr}, [\mathit{expr}_1, \ldots, \mathit{expr}_n]) \mid \mathit{expr}_1\ !\ \mathit{expr}_2 \mid \mathrm{self}() \\
\mathit{clause} & ::= & \mathit{pat}\ \mathrm{when}\ \mathit{expr}_1 \to \mathit{expr}_2 \\
\mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [\mathit{pat}_1|\mathit{pat}_2] \mid \{\mathit{pat}_1, \ldots, \mathit{pat}_n\}
\end{array}
$$

Core Erlang syntax

We consider a subset of Core Erlang with this syntax:

$$
\begin{array}{rcl}
\mathit{Module} & ::= & \mathrm{module}\ \mathit{Atom} = \mathit{fun}_1, \ldots, \mathit{fun}_n \\
\mathit{fun} & ::= & \mathit{fname} = \mathrm{fun}\ (X_1, \ldots, X_n) \to \mathit{expr} \\
\mathit{fname} & ::= & \mathit{Atom}/\mathit{Integer} \\
\mathit{lit} & ::= & \mathit{Atom} \mid \mathit{Integer} \mid \mathit{Float} \mid [\,] \\
\mathit{expr} & ::= & \mathit{Var} \mid \mathit{lit} \mid \mathit{fname} \mid [\mathit{expr}_1|\mathit{expr}_2] \mid \{\mathit{expr}_1, \ldots, \mathit{expr}_n\} \\
& \mid & \mathrm{call}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \mid \mathrm{apply}\ \mathit{expr}\ (\mathit{expr}_1, \ldots, \mathit{expr}_n) \\
& \mid & \mathrm{case}\ \mathit{expr}\ \mathrm{of}\ \mathit{clause}_1; \ldots; \mathit{clause}_m\ \mathrm{end} \\
& \mid & \mathrm{let}\ \mathit{Var} = \mathit{expr}_1\ \mathrm{in}\ \mathit{expr}_2 \mid \mathrm{receive}\ \mathit{clause}_1; \ldots; \mathit{clause}_n\ \mathrm{end} \\
& \mid & \mathrm{spawn}(\mathit{expr}, [\mathit{expr}_1, \ldots, \mathit{expr}_n]) \mid \mathit{expr}_1\ !\ \mathit{expr}_2 \mid \mathrm{self}() \\
\mathit{clause} & ::= & \mathit{pat}\ \mathrm{when}\ \mathit{expr}_1 \to \mathit{expr}_2 \\
\mathit{pat} & ::= & \mathit{Var} \mid \mathit{lit} \mid [\mathit{pat}_1|\mathit{pat}_2] \mid \{\mathit{pat}_1, \ldots, \mathit{pat}_n\}
\end{array}
$$

Some preliminary definitions

Definition (process) //no local queue!

A process is a triple $\langle p, \theta, e \rangle$ where

- p is the pid of the process
- θ is an environment
- e is the expression to be reduced

Definition (system)

A system is denoted by $\Gamma; \Pi$, where

- Γ models the network & local queues (global mailbox); a multiset of triples $(sender\_pid, target\_pid, message)$
- Π is a pool of processes

We use $\boxed{\Gamma; \langle p, \theta, e \rangle \ \& \ \Pi}$ *to denote an arbitrary system*

Some preliminary definitions

Definition (process) //no local queue!

A process is a triple $\langle p, \theta, e \rangle$ where

- p is the pid of the process
- θ is an environment
- e is the expression to be reduced

Definition (system)

A system is denoted by $\Gamma; \Pi$, where

- Γ models the network & local queues (global mailbox); a multiset of triples $(\mathit{sender\_pid}, \mathit{target\_pid}, \mathit{message})$
- Π is a pool of processes

We use $\boxed{\Gamma; \langle p, \theta, e \rangle \;\&\; \Pi}$ *to denote an arbitrary system*

Semantics

Erlang guarantees that, if two messages are sent from process p to process p', and both are delivered, then the order of these messages is kept

1. [LOPSTR16] ensures this restriction
2. [JLAMP18,FLOPS18,FORTE19] ignore this restriction

Semantics

Erlang guarantees that, if two messages are sent from process p to process p', and both are delivered, then the order of these messages is kept

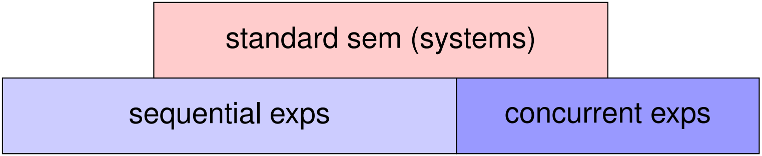
1. [LOPSTR16] ensures this restriction
2. [JLAMP18,FLOPS18,FORTE19] ignore this restriction

Semantics

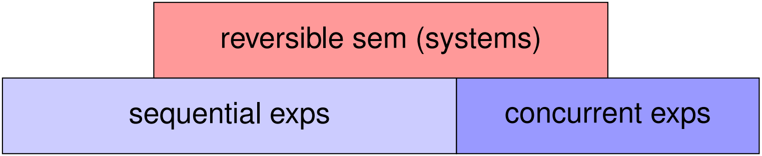
Erlang guarantees that, if two messages are sent from process p to process p', and both are delivered, then the order of these messages is kept

1. [LOPSTR16] ensures this restriction
2. [JLAMP18,FLOPS18,FORTE19] ignore this restriction

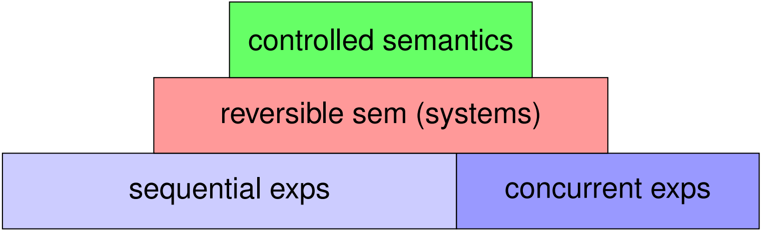
Reduction semantics (layers)



Reduction semantics (layers)



Reduction semantics (layers)



Semantics of concurrent expressions

For concurrent actions, we face the following problems:

1. we don't know the result of the actions (fresh variables)
2. we must perform side effects (labels)

Labels

- At expression level, transitions for concurrent actions are labelled with enough information
- At system level, labels are used to perform the associated actions

Semantics of concurrent expressions

For concurrent actions, we face the following problems:

1. we don't know the result of the actions (fresh variables)
2. we must perform side effects (labels)

Labels

- At expression level, transitions for concurrent actions are labelled with enough information
- At system level, labels are used to perform the associated actions

Semantics of concurrent expressions

For concurrent actions, we face the following problems:

1. we don't know the result of the actions (fresh variables)
2. we must perform side effects (labels)

Labels

- At expression level, transitions for concurrent actions are labelled with enough information
- At system level, labels are used to perform the associated actions

Expression semantics: sequential expressions

$$(\mathit{Var})\ \frac{}{\theta, X \xrightarrow{\tau} \theta, \theta(X)} \qquad (\mathit{Tuple})\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i'}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \xrightarrow{\ell} \theta', \{\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}\}}$$

$$(\mathit{List1})\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, [e_1 | e_2] \xrightarrow{\ell} \theta', [e_1' | e_2]} \qquad (\mathit{List2})\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, [v_1 | e_2] \xrightarrow{\ell} \theta', [v_1 | e_2']}$$

$$(\mathit{Let1})\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \xrightarrow{\ell} \theta', \mathsf{let}\ X = e_1'\ \mathsf{in}\ e_2} \qquad (\mathit{Let2})\ \frac{}{\theta, \mathsf{let}\ X = v\ \mathsf{in}\ e \xrightarrow{\tau} \theta[X \mapsto v], e}$$

$$(\mathit{Case1})\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\begin{array}{l}\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \\ \xrightarrow{\ell} \theta', \mathsf{case}\ e'\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end}\end{array}} \qquad (\mathit{Case2})\ \frac{\mathsf{match}(v, cl_1, \ldots, cl_n) = \langle \theta_i, e_i \rangle}{\theta, \mathsf{case}\ v\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\tau} \theta\theta_i, e_i}$$

$$(\mathit{Apply1})\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{apply}\ a/n\ (\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\ell} \theta', \mathsf{apply}\ a/n\ (\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$(\mathit{Apply2})\ \frac{\mu(a/n) = \mathsf{fun}\ (X_1, \ldots, X_n) \to e}{\theta, \mathsf{apply}\ a/n\ (v_1, \ldots, v_n) \xrightarrow{\tau} \{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}, e}$$

Sending a message

(expression semantics)

$$(\mathit{Send1})\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1' \,!\, e_2} \qquad \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \,!\, e_2 \xrightarrow{\ell} \theta', v_1 \,!\, e_2'}$$

$$(\mathit{Send2})\ \frac{}{\theta, v_1 \,!\, v_2 \xrightarrow{\mathsf{send}(v_1, v_2)} \theta, v_2}$$

(system semantics)

$$(\mathit{Send})\ \frac{\theta, e \xrightarrow{\mathsf{send}(p', v)} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma \cup \{(p, p', v)\}; \langle p, \theta', e'\rangle \& \Pi}$$

Sending a message

(expression semantics)

$$(\mathit{Send1})\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1' \,!\, e_2} \qquad \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \,!\, e_2 \xrightarrow{\ell} \theta', v_1 \,!\, e_2'}$$

$$(\mathit{Send2})\ \frac{}{\theta, v_1 \,!\, v_2 \xrightarrow{\mathsf{send}(v_1, v_2)} \theta, v_2}$$

(system semantics)

$$(\mathit{Send})\ \frac{\theta, e \xrightarrow{\mathsf{send}(p', v)} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma \cup \{(p, p', v)\}; \langle p, \theta', e'\rangle \& \Pi}$$

Self

(expression semantics)

$$(\mathit{Self})\ \frac{}{\theta, \mathsf{self}() \xrightarrow{\mathit{self}(\kappa)} \theta, \kappa}$$

(system semantics)

$$(\mathit{Self})\ \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, \theta, e \rangle \& \Pi \hookrightarrow \Gamma; \langle p, \theta', e'\{\kappa \mapsto p\} \rangle \& \Pi}$$

Self

(expression semantics)

$$(\mathit{Self})\ \frac{}{\theta, \mathsf{self}() \xrightarrow{\mathit{self}(\kappa)} \theta, \kappa}$$

(system semantics)

$$(\mathit{Self})\ \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma;\ \ \langle p, \theta', e'\{\kappa \mapsto p\}\rangle \& \Pi}$$

Spawning a process

(expression semantics)

$$(\mathit{Spawn})\ \frac{}{\theta, \mathsf{spawn}(a/n, [v_1, \ldots, v_n]) \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta, \kappa}$$

(system semantics)

$$(\mathit{Spawn})\ \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma; \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \& \langle p', \theta', \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \& \Pi}$$

Spawning a process

(expression semantics)

$$(\mathit{Spawn})\ \frac{}{\theta, \mathsf{spawn}(a/n, [v_1, \ldots, v_n]) \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta, \kappa}$$

(system semantics)

$$(\mathit{Spawn})\ \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma; \quad \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \& \langle p', \theta', \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \& \Pi}$$

Receiving a message

(expression semantics)

$$(\textit{Receive})\ \frac{}{\theta, \mathsf{receive}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta, \kappa}$$

(system semantics)

$$(\textit{Receive})\ \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, v)\}; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \& \Pi}$$

Receiving a message

(expression semantics)

$$(\mathit{Receive})\ \frac{}{\theta, \text{receive } cl_1; \ldots; cl_n \text{ end} \xrightarrow{\text{rec}(\kappa, \overline{cl_n})} \theta, \kappa}$$

(system semantics)

$$(\mathit{Receive})\ \frac{\theta, e \xrightarrow{\text{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \text{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, v)\}; \langle p, \theta, e\rangle \& \Pi \hookrightarrow \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \& \Pi}$$

Reversible semantics (uncontrolled)

1. Forward reversible semantics: we instrument the system rules using a Landauer embedding
2. Backward reversible semantics: straightforward inversion of the previous rules

Processes have now the form $\langle p, h, \theta, e \rangle$

history h

is a sequence of terms headed by constructors seq, send, rec, spawn, and self, and whose arguments are the information required to (deterministically) undo the step

Reversible semantics (uncontrolled)

1. Forward reversible semantics: we instrument the system rules using a Landauer embedding
2. Backward reversible semantics: straightforward inversion of the previous rules

Processes have now the form $\langle p, h, \theta, e \rangle$

history h

is a sequence of terms headed by constructors seq, send, rec, spawn, and self, and whose arguments are the information required to (deterministically) undo the step

Uncontrolled forward semantics

(Send)

$$\frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e'}{\begin{array}{l}\Gamma; \langle p, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{send}(\ell)} \Gamma \cup \{(p, p', v)\}; \\ \qquad \langle p, \mathsf{send}(\theta, e, p', v) : h, \theta', e'\rangle \mid \Pi\end{array}}$$

$(\mathit{Receive})$

$$\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e' \ \text{ and } \ \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\begin{array}{l}\Gamma \cup \{(p', p, v)\}\langle p, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{rec}(\ell)} \Gamma; \langle p, \mathsf{rec}(\theta, e, p', v) : h, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi\end{array}}$$

(Spawn)

$$\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta', e' \quad \text{and } p' \text{ is a fresh pid}}{\begin{array}{l}\Gamma; \langle p, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{spawn}(p')} \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, \theta', e'\{\kappa \mapsto p'\}\rangle \\ \qquad \mid \langle p', [\,], id, \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \mid \Pi\end{array}}$$

Uncontrolled backward semantics

$$(\overline{Send})\quad \begin{array}{l} \Gamma \cup \{(p, p', v)\}; \langle p, \mathsf{send}(\theta, e, p', v) : h, \theta', e'\rangle \mid \Pi \\ \quad \leftharpoondown_{p,\mathsf{send}(\ell)} \Gamma; \langle p, h, \theta, e\rangle \mid \Pi \end{array}$$

$$(\overline{Receive})\quad \begin{array}{l} \Gamma; \langle p, \mathsf{rec}(\theta, e, p', v) : h, \theta', e'\rangle \mid \Pi \\ \quad \leftharpoondown_{p,\mathsf{rec}(\ell)} \Gamma \cup \{(p', p, v)\}; \langle p, h, \theta, e\rangle \mid \Pi \\ \quad\quad \text{where } \mathcal{V} = \mathcal{D}om(\theta') \backslash \mathcal{D}om(\theta) \end{array}$$

$$(\overline{Spawn})\quad \begin{array}{l} \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, \theta', e'\rangle \mid \langle p', [\,], id, e''\rangle \mid \Pi \\ \quad \leftharpoondown_{p,\mathsf{spawn}(p')} \Gamma; \langle p, h, \theta, e\rangle \mid \Pi \end{array}$$

$\Rightarrow$ reversible computations must be **causal consistent**

Uncontrolled backward semantics

$$(\overline{\mathit{Send}}) \quad \begin{array}{l} \Gamma \cup \{(p, p', v)\}; \langle p, \mathsf{send}(\theta, e, p', v) : h, \theta', e' \rangle \mid \Pi \\ \quad \leftharpoondown_{p,\mathsf{send}(\ell)} \Gamma; \langle p, h, \theta, e \rangle \mid \Pi \end{array}$$

$$(\overline{\mathit{Receive}}) \quad \begin{array}{l} \Gamma; \langle p, \mathsf{rec}(\theta, e, p', v) : h, \theta', e' \rangle \mid \Pi \\ \quad \leftharpoondown_{p,\mathsf{rec}(\ell)} \Gamma \cup \{(p', p, v)\}; \langle p, h, \theta, e \rangle \mid \Pi \\ \quad\quad \text{where } \mathcal{V} = \mathcal{D}om(\theta') \backslash \mathcal{D}om(\theta) \end{array}$$

$$(\overline{\mathit{Spawn}}) \quad \begin{array}{l} \Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, \theta', e' \rangle \mid \langle p', [\,], id, e'' \rangle \mid \Pi \\ \quad \leftharpoondown_{p,\mathsf{spawn}(p')} \Gamma; \langle p, h, \theta, e \rangle \mid \Pi \end{array}$$

$\Rightarrow$ reversible computations must be **causal consistent**

Designing a reversible debugger

A common problem: in concurrent languages, replaying a particular computation might be difficult (even impossible) given the nondeterminism of the language

Solution

1. instrument the code so that it generates a **log** (a sequence of messages received by each process)
2. forward reversible semantics is **driven** by the log (causal-consistent replay semantics)
3. **controlled** reversible semantics driven by user requests (both replay requests and rollbacks)

Designing a reversible debugger

A common problem: in concurrent languages, replaying a particular computation might be difficult (even impossible) given the nondeterminism of the language

Solution

1. instrument the code so that it generates a **log** (a sequence of messages received by each process)
2. forward reversible semantics is **driven** by the log (causal-consistent replay semantics)
3. **controlled** reversible semantics driven by user requests (both replay requests and rollbacks)

Logging semantics

We tag messages with unique identifiers

$$v \quad \mapsto \quad \{v, \ell\}, \qquad \text{where } \ell \text{ is fresh}$$

A log $\mathcal{L}(d)$ of a derivation d is a sequence of items spawn(p), send(ℓ) or rec(ℓ) for each process in d

(logs are local to each process)

(*Seq*)

$$\frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{seq}} \Gamma; \langle p, \theta', e'\rangle \mid \Pi}$$

(*Send*)

$$\frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e' \text{ and } \ell \text{ is a fresh symbol}}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{send}(\ell)} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \theta', e'\rangle \mid \Pi}$$

(*Receive*)

$$\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e' \quad \text{and} \quad \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{rec}(\ell)} \Gamma; \langle p, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi}$$

(*Spawn*)

$$\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta', e' \quad \text{and} \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{spawn}(p')} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \mid \langle p', id, \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \mid \Pi}$$

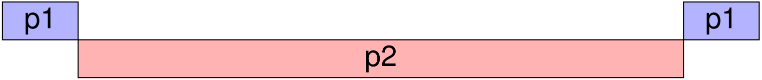
(*Self*)

$$\frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, \theta, e\rangle \mid \Pi \hookrightarrow_{p,\mathsf{self}} \Gamma; \langle p, \theta', e'\{\kappa \mapsto p\}\rangle \mid \Pi}$$

(implemented by a program instrumentation)

Causally equivalent derivations

Traditional reversible debuggers allow us to go backwards in exactly the inverse order of the forward computation

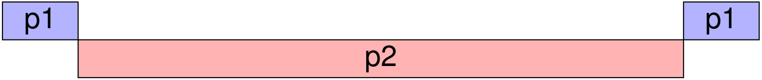


If p1 and p2 are independent then

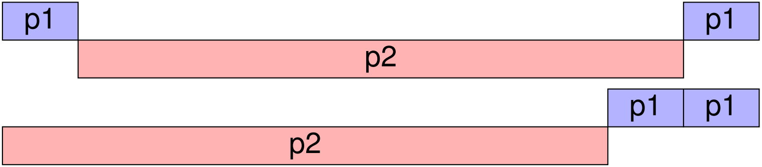
are **causally equivalent**

Causally equivalent derivations

Traditional reversible debuggers allow us to go backwards in exactly the inverse order of the forward computation



If p1 and p2 are independent then



are **causally equivalent**

Causally equivalent derivations

d_1 and d_2 are causally equivalent ($d_1 \approx d_2$) if d_1 can be obtained from d_2 by switching consecutive transitions **as long as**

1. the actions of a given process cannot be switched
2. no message can be received before it is sent
3. a process cannot perform any action before it is spawned

Given (coinitial) derivations d_1 and d_2, $\boxed{\mathcal{L}(d_1) = \mathcal{L}(d_2) \text{ iff } d_1 \approx d_2}$

Causally equivalent derivations

d_1 and d_2 are causally equivalent ($d_1 \approx d_2$) if d_1 can be obtained from d_2 by switching consecutive transitions **as long as**

1. the actions of a given process cannot be switched
2. no message can be received before it is sent
3. a process cannot perform any action before it is spawned

Given (coinitial) derivations d_1 and d_2, $\boxed{\mathcal{L}(d_1) = \mathcal{L}(d_2) \text{ iff } d_1 \approx d_2}$

Causally equivalent derivations

d_1 and d_2 are causally equivalent ($d_1 \approx d_2$) if d_1 can be obtained from d_2 by switching consecutive transitions **as long as**

1. the actions of a given process cannot be switched
2. no message can be received before it is sent
3. a process cannot perform any action before it is spawned

Given (coinitial) derivations d_1 and d_2, $\boxed{\mathcal{L}(d_1) = \mathcal{L}(d_2) \text{ iff } d_1 \approx d_2}$

Causally equivalent derivations

d_1 and d_2 are causally equivalent ($d_1 \approx d_2$) if d_1 can be obtained from d_2 by switching consecutive transitions **as long as**

1. the actions of a given process cannot be switched
2. no message can be received before it is sent
3. a process cannot perform any action before it is spawned

Given (coinitial) derivations d_1 and d_2, $\boxed{\mathcal{L}(d_1) = \mathcal{L}(d_2) \text{ iff } d_1 \approx d_2}$

Reversible Semantics

Processes have the form $\langle p, \omega, h, \theta, e \rangle$
with ω a *log* and h a *history*

A history h is a sequence of terms headed by constructors seq, send, rec, spawn, and self, and whose arguments are the information required to (deterministically) undo the step

Uncontrolled forward semantics

(Send)

$$\frac{\theta, e \xrightarrow{\mathsf{send}(p',v)} \theta', e'}{\begin{array}{l}\Gamma; \langle p, \mathsf{send}(\ell)\!:\!\omega, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell\Uparrow\}} \Gamma \cup \{(p, p', \{v,\ell\})\}; \\ \qquad \langle p, \omega, \mathsf{send}(\theta, e, p', \{v,\ell\})\!:\!h, \theta', e'\rangle \mid \Pi\end{array}}$$

$(\mathit{Receive})$

$$\frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa,\overline{cl_n})} \theta', e' \;\text{ and }\; \mathsf{matchrec}(\theta, \overline{cl_n}, v) = (\theta_i, e_i)}{\begin{array}{l}\Gamma \cup \{(p', p, \{v,\ell\})\}\langle p, \mathsf{rec}(\ell)\!:\!\omega, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell\Downarrow\}} \Gamma; \langle p, \omega, \mathsf{rec}(\theta, e, p', \{v,\ell\})\!:\!h, \theta'\theta_i, e'\{\kappa \mapsto e_i\}\rangle \mid \Pi\end{array}}$$

(Spawn)

$$\frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa,a/n,[\overline{v_n}])} \theta', e' \;\text{ and }\; \omega' = \mathsf{trace}(d, p')}{\begin{array}{l}\Gamma; \langle p, \mathsf{spawn}(p')\!:\!\omega, h, \theta, e\rangle \mid \Pi \\ \rightharpoonup_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \Gamma; \langle p, \omega, \mathsf{spawn}(\theta, e, p')\!:\!h, \theta', e'\{\kappa \mapsto p'\}\rangle \\ \qquad \mid \langle p', \omega', [\,], id, \mathsf{apply}\ a/n\ (\overline{v_n})\rangle \mid \Pi\end{array}}$$

Uncontrolled backward semantics

$$(\overline{Send}) \quad \begin{array}{l} \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \omega, \mathsf{send}(\theta, e, p', \{v, \ell\}) : h, \theta', e' \rangle \mid \Pi \\ \leftharpoondown_{p,\mathsf{send}(\ell),\{\mathsf{s},\ell\Uparrow\}} \; \Gamma; \langle p, \mathsf{send}(\ell) : \omega, h, \theta, e \rangle \mid \Pi \end{array}$$

$$(\overline{Receive}) \quad \begin{array}{l} \Gamma; \langle p, \omega, \mathsf{rec}(\theta, e, p', \{v, \ell\}) : h, \theta', e' \rangle \mid \Pi \\ \leftharpoondown_{p,\mathsf{rec}(\ell),\{\mathsf{s},\ell\Downarrow\}\cup\mathcal{V}} \; \Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \mathsf{rec}(\ell) : \omega, h, \theta, e \rangle \mid \Pi \\ \text{where } \mathcal{V} = \mathcal{D}om(\theta') \backslash \mathcal{D}om(\theta) \end{array}$$

$$(\overline{Spawn}) \quad \begin{array}{l} \Gamma; \langle p, \omega, \mathsf{spawn}(\theta, e, p') : h, \theta', e' \rangle \mid \langle p', \omega', [\,], id, e'' \rangle \mid \Pi \\ \leftharpoondown_{p,\mathsf{spawn}(p'),\{\mathsf{s},\mathsf{sp}_{p'}\}} \; \Gamma; \langle p, \mathsf{spawn}(p') : \omega, h, \theta, e \rangle \mid \Pi \end{array}$$

Some results...

Coinitial derivations are cofinal
iff they are causally equivalent

Misbehaviors are preserved
by all causally equivalent derivations

Controlled replay/rollback semantics

We allow the user to start a replay/rollback until a particular action is performed, e.g.,

- $\{p, \mathsf{s}\}$: one step backward/forward of process p
- $\{p, \ell^{\Uparrow}\}$: a backward/forward derivation of process p up to the sending of the message tagged with ℓ
- $\{p, \ell^{\Downarrow}\}$: a backward/forward derivation of process p up to the reception of the message tagged with ℓ
- $\{p, \mathsf{sp}_{p'}\}$: a backward/forward derivation of process p up to the spawning of the process with pid p'
- $\{p, X\}$: a backward derivation of process p up to the introduction of variable X
- …

Controlled semantics takes a stack of requests (initially one)

It is defined as a layer on top of the uncontrolled semantics:

- If a process can perform a step satisfying the request on top of the stack → do it and remove the request
- If a process can perform a step but it doesn't satisfy the request → update the system but keep the request
- If a step on the process is not possible → track dependencies and add new requests on top of the stack

Controlled semantics takes a stack of requests (initially one)

It is defined as a layer on top of the uncontrolled semantics:

- If a process can perform a step satisfying the request on top of the stack $\rightarrow$ do it and remove the request
- If a process can perform a step but it doesn't satisfy the request $\rightarrow$ update the system but keep the request
- If a step on the process is not possible $\rightarrow$ track dependencies and add new requests on top of the stack

Controlled semantics takes a stack of requests (initially one)

It is defined as a layer on top of the uncontrolled semantics:

- If a process can perform a step satisfying the request on top of the stack $\rightarrow$ do it and remove the request
- If a process can perform a step but it doesn't satisfy the request $\rightarrow$ update the system but keep the request
- If a step on the process is not possible $\rightarrow$ track dependencies and add new requests on top of the stack

Controlled semantics takes a stack of requests (initially one)

It is defined as a layer on top of the uncontrolled semantics:

- If a process can perform a step satisfying the request on top of the stack $\rightarrow$ do it and remove the request
- If a process can perform a step but it doesn't satisfy the request $\rightarrow$ update the system but keep the request
- If a step on the process is not possible $\rightarrow$ track dependencies and add new requests on top of the stack

Reversible debugging

Two components: code instrumentation (logging)
$+$ causal-consistent reversible debugger (CauDEr)

`https://github.com/mistupv/tracer`
`https://github.com/mistupv/cauder/tree/replay`

A Note on the implementation

Current prototypes show good potential, but more implementation effort is still required:

- move from Core Erlang to Erlang
- graphical representation of logs
- consider more Erlang features: links, monitors, built-in's, input/output, behaviours, etc
- combine it with program slicing / automatic bug location

Recap

Every (irreversible) language can be made reversible by defining a Landauer embedding

Recap

Reversibilization can be achieved by instrumenting the semantics or the program

Recap

Reversibilization $\neq$ program inversion

Recap

For concurrent languages, causal consistency is essential

Recap

There are many other applications of reversible computation: quantum computing, discrete simulation, hardware design, computational biology, robotics, etc

Thanks for your attention!

Questions?