Reversibilization in Functional and Concurrent Programming

Germán Vidal

VRAIN
Valencian Research Institute for Artificial Intelligence
Introduction

Functional
Landauer embedding
transformations
application: Bx

Concurrent
syntax (sequential)
syntax (concurrent)
core Erlang
semantics
reversible
semantics

Application:
reversible
debugging
logging semantics
causal consistency
replay semantics
controlled
semantics
reversible
debugging

Recap

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**: \( \text{if } c_1 \text{ then } s_1 \text{ else } s_2 \text{ fi } c_2 \)

Reversible languages are not universal (e.g., cannot compute non-injective functions)
Each execution step is **reversible**

Backward steps must be **deterministic**

E.g., **Janus**: ```
if c₁ then s₁ else s₂ fi c₂
```
Given an irreversible programming language $L$ with semantics $\text{Sem}$ over states $s_0, s_1, \ldots, s_n \in \text{State}$:

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

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

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

becomes reversible

This is known as a Landauer embedding and is the main technique for reversibilization.
Given an irreversible programming language $L$ with semantics $\text{Sem}$ over states $s_0, s_1, \ldots, s_n \in \text{State}$:

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

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

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

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.,

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y)) \\
\text{fst}(x, y) & \rightarrow x
\end{align*}
\]

An example reduction:

\[
\begin{align*}
\text{fst}(\text{add}(s(0), 0), 0) & \rightarrow \text{fst}(s(\text{add}(0, 0)), 0) \\
& \rightarrow \text{fst}(s(0), 0) \\
& \rightarrow s(0)
\end{align*}
\]
Defining a Landauer embedding

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

\[
\begin{align*}
\text{add}(0, y) & \leftarrow y \\
\text{add}(s(x), y) & \leftarrow s(\text{add}(x, y)) \\
\text{fst}(x, y) & \leftarrow x
\end{align*}
\]

An example reduction:

\[
\text{fst}(\text{add}(s(0), 0), 0) \leftarrow \text{fst}(s(\text{add}(0, 0)), 0) \rightarrow \text{fst}(s(0), 0) \rightarrow s(0)
\]

What should include a Landauer embedding?
Defining a Landauer embedding

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

\[
\begin{align*}
\text{add}(0, y) & \leftarrow y \\
\text{add}(s(x), y) & \leftarrow s(\text{add}(x, y)) \\
\text{fst}(x, y) & \leftarrow x
\end{align*}
\]

An example reduction:

\[
\text{fst}(\text{add}(s(0), 0), 0) \leftarrow \text{fst}(s(\text{add}(0, 0)), 0) \rightarrow \text{fst}(s(0), 0) \rightarrow s(0)
\]

What should include a Landauer embedding?

⇒ position of reduced expression
Functions defined by pattern-matching, e.g.,

\[ \beta_1 : \quad \text{add}(0, y) \leftarrow y \]
\[ \beta_2 : \quad \text{add}(s(x), y) \leftarrow s(\text{add}(x, y)) \]
\[ \beta_3 : \quad \text{fst}(x, y) \leftarrow x \]

An example reduction:

\[ \text{fst}(\text{add}(s(0), 0), 0) \leftarrow \text{fst}(s(\underline{\text{add}(0, 0)}), 0) \rightarrow \text{fst}(s(0), 0) \rightarrow s(0) \]

What should include a Landauer embedding?
⇒ position of reduced expression, rule
Defining a Landauer embedding

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

\[ \beta_1 : \text{add}(0, y) \leftarrow y \]
\[ \beta_2 : \text{add}(s(x), y) \leftarrow s(\text{add}(x, y)) \]
\[ \beta_3 : \text{fst}(x, [y]) \leftarrow x \]

An example reduction:

\[ \text{fst}(\text{add}(s(0), 0), 0) \leftarrow \text{fst}(s(\text{add}(0, 0)), 0) \leftarrow \text{fst}(s(0), 0) \leftarrow s(0) \]

What should include a Landauer embedding?

⇒ position of reduced expression, rule
Defining a Landauer embedding

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

\[
\begin{align*}
\beta_1 : & \quad \text{add}(0, y) \leftarrow y \\
\beta_2 : & \quad \text{add}(s(x), y) \leftarrow s(\text{add}(x, y)) \\
\beta_3 : & \quad \text{fst}(x, y) \leftarrow x
\end{align*}
\]

An example reduction:

\[
\text{fst}(\text{add}(s(0), 0), 0) \leftarrow \text{fst}(s(\text{add}(0, 0)), 0) \leftarrow \text{fst}(s(0), 0) \leftarrow s(0)
\]

What should include a Landauer embedding?

⇒ position of reduced expression, rule, erased values
We store a trace term $\beta(p, \sigma)$ at every reduction step:

$\langle \text{fst}(\text{add}(\text{s}(0), 0), 0), [] \rangle$

$\rightarrow \langle \text{fst}(\text{s}((\text{add}(0, 0)), 0), [\beta_2(1, \text{id})] \rangle$

$\rightarrow \langle \text{fst}(\text{s}(0), 0), [\beta_1(1.1, \text{id}), \beta_2(1, \text{id})] \rangle$

$\rightarrow \langle \text{s}(0), [\beta_3(\epsilon, \{y \mapsto 0\}), \beta_1(1.1, \text{id}), \beta_2(1, \text{id})] \rangle$

where

- $\rightarrow$ is the reversible forward reduction relation
- $\leftarrow$ is the reversible backward reduction relation
We store a trace term $\beta(p, \sigma)$ at every reduction step:

$$\langle \text{fst}(\text{add}(s(0), 0), 0), [\ ] \rangle$$

$$\rightarrow \langle \text{fst}(s(\text{add}(0, 0)), 0), [\beta_2(1, \text{id})] \rangle$$

$$\rightarrow \langle \text{fst}(s(0), 0), [\beta_1(1.1, \text{id}), \beta_2(1, \text{id})] \rangle$$

$$\rightarrow \langle s(0), [\beta_3(\epsilon, \{y \mapsto 0\}), \beta_1(1.1, \text{id}), \beta_2(1, \text{id})] \rangle$$

where

- $\rightarrow$ is the reversible forward reduction relation
- $\leftarrow$ 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

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

and

$$\langle s_2, \pi_2 \rangle \xleftarrow{\mathcal{R}} \langle s_1, \pi_1 \rangle \text{ iff } \langle s_2, \pi_2 \rangle \xrightarrow{\mathcal{R}_b} \langle s_1, \pi_1 \rangle$$
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.,

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y))
\end{align*}
\]

\[
\downarrow
\]

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{add}(s(x), y) & \rightarrow s(z) \leftarrow \text{add}(x, y) \rightarrow z
\end{align*}
\]

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.,

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{add}(s(x), y) & \rightarrow s(\text{add}(x, y))
\end{align*}
\]

\[\downarrow\]

\[
\begin{align*}
\text{add}(0, y) & \rightarrow y \\
\text{add}(s(x), y) & \rightarrow s(z) \leftarrow \text{add}(x, y) \rightarrow z
\end{align*}
\]

so that all function calls occur at root positions
Thus we can get rid of positions in trace terms...

$$\beta(p, \sigma) \Rightarrow \beta(\sigma)$$
A conditional rule:

\[ f(s_0) \rightarrow r \Leftarrow f_1(s_1) \rightarrow t_1, \ldots, f_n(s_n) \rightarrow t_n \]

is equivalent to (Haskell-like):

\[ f \; s_0 = r \text{ where } t_1 = f_1 \; s_1, \ldots, t_n = f_n \; s_n \]

or

\[ f \; s_0 = \text{let } t_1 = f_1 \; s_1, \ldots, t_n = f_n \; s_n \text{ in } r \]

E.g.,

\[
\begin{align*}
\text{add} \; 0 \; y &= y \\
\text{add} \; (s \; x) \; y &= \text{let } z = \text{add} \; x \; y \text{ in } s(z)
\end{align*}
\]
A conditional rule:

\[ f(s_0) \rightarrow r \iff f_1(s_1) \rightarrow t_1, \ldots, f_n(s_n) \rightarrow t_n \]

is equivalent to (Haskell-like):

\[ f \ s_0 = r \text{ where } t_1 = f_1 \ s_1, \ldots, t_n = f_n \ s_n \]

or

\[ f \ s_0 = \text{let } t_1 = f_1 \ s_1, \ldots, t_n = f_n \ s_n \text{ in } r \]

E.g.,

\[
\begin{align*}
\text{add} \ 0 \ y &= y \\
\text{add} \ (s \ x) \ y &= \text{let } z = \text{add} \ x \ y \text{ in } s(z)
\end{align*}
\]
Injectivization

We replace each rule

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

by a new rule of the form

$$f_i(s_0) \to \langle r, \beta(\overline{y}, w_n) \rangle \Leftarrow f_i^1(s_1) \to \langle t_1, w_1 \rangle, \ldots, f_i^1(s_n) \to \langle t_n, w_n \rangle$$

where \( \overline{y} = (\forall \text{ar}(s_0) \setminus \forall \text{ar}(r, s_n, t_n)) \cup \bigcup_{i=1}^n \forall \text{ar}(t_i) \setminus \forall \text{ar}(r, s_{i+1}, n) \)

Inversion

We replace each rule

$$f_i(s_0) \to \langle r, \beta(\overline{y}, w_n) \rangle \Leftarrow f_i^1(s_1) \to \langle t_1, w_1 \rangle, \ldots, f_i^1(s_n) \to \langle t_n, w_n \rangle$$

by a new rule of the form

$$f_i^{-1}(r, \beta(\overline{y}, w_n)) \to \langle s_0 \rangle \Leftarrow f_n^{-1}(t_n, w_n) \to \langle s_n \rangle, \ldots, f_1^{-1}(t_1, w_1) \to \langle s_1 \rangle$$
Injectivization & Inversion: An example

\[ \beta_1 : \quad \text{add}(0, y) \rightarrow y \]
\[ \beta_2 : \quad \text{add}(s(x), y) \rightarrow s(x_1) \leftarrow \text{add}(x, y) \rightarrow x_1 \]
\[ \beta_3 : \quad \text{fst}(x, y) \rightarrow x \]

\[ \text{add}^i(0, y) \rightarrow \langle y, \beta_1 \rangle \]
\[ \text{add}^i(s(x), y) \rightarrow \langle s(x_1), \beta_2(w_1) \rangle \leftarrow \text{add}^i(x, y) \rightarrow \langle x_1, w_1 \rangle \]
\[ \text{fst}^i(x, y) \rightarrow \langle x, \beta_3(y) \rangle \]

\[ \text{add}^{-1}(y, \beta_1) \rightarrow \langle 0, y \rangle \]
\[ \text{add}^{-1}(s(x_1), \beta_2(w_1)) \rightarrow \langle s(x), y \rangle \leftarrow \text{add}^{-1}(x_1, w_1) \rightarrow \langle x, y \rangle \]
\[ \text{fst}^{-1}(x, \beta_3(y)) \rightarrow \langle x, y \rangle \]
Injectivization & Inversion: An example

\[ \beta_1 : \quad \text{add}(0, y) \rightarrow y \]
\[ \beta_2 : \quad \text{add}(s(x), y) \rightarrow s(x_1) \leftarrow \text{add}(x, y) \rightarrow x_1 \]
\[ \beta_3 : \quad \text{fst}(x, y) \rightarrow x \]

\[ \text{add}^i(0, y) \rightarrow \langle y, \beta_1 \rangle \]
\[ \text{add}^i(s(x), y) \rightarrow \langle s(x_1), \beta_2(w_1) \rangle \leftarrow \text{add}^i(x, y) \rightarrow \langle x_1, w_1 \rangle \]
\[ \text{fst}^i(x, y) \rightarrow \langle x, \beta_3(y) \rangle \]

\[ \text{add}^{-1}(y, \beta_1) \rightarrow \langle 0, y \rangle \]
\[ \text{add}^{-1}(s(x_1), \beta_2(w_1)) \rightarrow \langle s(x), y \rangle \leftarrow \text{add}^{-1}(x_1, w_1) \rightarrow \langle x, y \rangle \]
\[ \text{fst}^{-1}(x, \beta_3(y)) \rightarrow \langle x, y \rangle \]
Injectivization & Inversion: An example

\[ \beta_1 : \text{add}(0, y) \rightarrow y \]
\[ \beta_2 : \text{add}(s(x), y) \rightarrow s(x_1) \leftrightarrow \text{add}(x, y) \rightarrow x_1 \]
\[ \beta_3 : \text{fst}(x, y) \rightarrow x \]

\[ \text{add}^i(0, y) \rightarrow \langle y, \beta_1 \rangle \]
\[ \text{add}^i(s(x), y) \rightarrow \langle s(x_1), \beta_2(w_1) \rangle \leftrightarrow \text{add}^i(x, y) \rightarrow \langle x_1, w_1 \rangle \]
\[ \text{fst}^i(x, y) \rightarrow \langle x, \beta_3(y) \rangle \]

\[ \text{add}^{-1}(y, \beta_1) \rightarrow \langle 0, y \rangle \]
\[ \text{add}^{-1}(s(x_1), \beta_2(w_1)) \rightarrow \langle s(x), y \rangle \leftrightarrow \text{add}^{-1}(x_1, w_1) \rightarrow \langle x, y \rangle \]
\[ \text{fst}^{-1}(x, \beta_3(y)) \rightarrow \langle x, y \rangle \]
Application: bidirectionalization
We have two data representations, called source and view with $\text{source} \supseteq \text{view}$ (assymmetric case)

Function $\text{get} : \text{source} \leftrightarrow \text{view}$

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

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

Function $\text{put} : \text{view} \times \text{source} \mapsto \text{source}$
We have two data representations, called source and view with source \( \supseteq \) view (assymmetric case)

Function \( \text{get} : \text{source} \leftrightarrow \text{view} \)

**Consistency:** \( s \in \text{source} \) is consistent with \( v \in \text{view} \) if \( \text{get}(s) = v \)

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

Function \( \text{put} : \text{view} \times \text{source} \mapsto \text{source} \)
We have two data representations, called source and view with source \( \supseteq \) view (assymmetric case)

Function \( \text{get} : \text{source} \leftrightarrow \text{view} \)

**Consistency:** \( s \in \text{source} \) is consistent with \( v \in \text{view} \) if \( \text{get}(s) = v \)

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

Function \( \text{put} : \text{view} \times \text{source} \leftrightarrow \text{source} \)

![Diagram showing bidirectional transformations]

- \( S \) to \( V \) via \( \text{get} \), \( S' \) to \( V' \) via \( \text{get} \)
- \( S \) to \( V \) via \( \text{put} \), \( S' \) to \( V' \) via \( \text{update} \)
Defining the right “put” is not easy
⇒ (syntactic) bidirectionalization
Stepwise approach to bidirectionalization

Example (first names)

\[
\begin{align*}
\text{fn}(\texttt{[]}) & \rightarrow \texttt{[]} \\
\text{fn}(& \texttt{person}(n, l): xs) \rightarrow n: ys \Leftarrow \text{fn}(xs) \rightarrow ys \\
\text{fn}(& \texttt{city}(c): xs) \rightarrow ys \Leftarrow \text{fn}(xs) \rightarrow ys
\end{align*}
\]

E.g., given

\[
s = \texttt{[person(john, smith), city(london), person(ada, lovelace)\]}\]

we have \(\text{fn}(s) = \texttt{[john, ada]}\)
Stepwise approach to bidirectionalization

Example (first names, injective version)

\[
\begin{align*}
\text{fn}^i([ ]) & \rightarrow \langle [ ], \beta_1 \rangle \\
\text{fn}^i(\text{person}(n, l) : xs) & \rightarrow \langle n : ys, \beta_2(l, w) \rangle \iff \text{fn}^i(xs) \rightarrow \langle ys, w \rangle \\
\text{fn}^i(\text{city}(c) : xs) & \rightarrow \langle ys, \beta_3(c, w) \rangle \iff \text{fn}^i(xs) \rightarrow \langle ys, w \rangle \\
\end{align*}
\]

E.g., given \( s = [\text{person}(\text{john}, \text{smith}), \text{city}(\text{london}), \text{person}(\text{ada}, \text{lovelace})] \), we have
\[
\text{fn}^i(s) = \langle [\text{john}, \text{ada}], \beta_2(\text{smith}, \beta_3(\text{london}, \beta_2(\text{lovelace}, \beta_1))) \rangle
\]

a complement!

(according to [Bancilhon & Spyratos, 1981])
Stepwise approach to bidirectionalization

Example (first names, injective version)

\[
\begin{align*}
\text{fn}^i([ ]) & \rightarrow \langle [ ], \beta_1 \rangle \\
\text{fn}^i(\text{person}(n, l): xs) & \rightarrow \langle n: ys, \beta_2(l, w) \rangle \Leftarrow \text{fn}^i(xs) \rightarrow \langle ys, w \rangle \\
\text{fn}^i(\text{city}(c): xs) & \rightarrow \langle ys, \beta_3(c, w) \rangle \Leftarrow \text{fn}^i(xs) \rightarrow \langle ys, w \rangle
\end{align*}
\]

E.g., given \( s = [\text{person}(\text{john}, \text{smith}), \text{city}(\text{london}), \text{person}(\text{ada}, \text{lovelace})] \),
we have
\[
\text{fn}^i(s) = \langle [\text{john}, \text{ada}], \beta_2(\text{smith}, \beta_3(\text{london}, \beta_2(\text{lovelace}, \beta_1))) \rangle
\]

a complement! (according to [Bancilhon & Spyratos, 1981])
Example (first names, injective version)

\[ fn^i([ ]) \rightarrow \langle [ ], \beta_1 \rangle \]
\[ fn^i(\text{person}(n, l) : xs) \rightarrow \langle n : ys, \beta_2(l, w) \rangle \leftrightarrow fn^i(xs) \rightarrow \langle ys, w \rangle \]
\[ fn^i(\text{city}(c) : xs) \rightarrow \langle ys, \beta_3(c, w) \rangle \leftrightarrow fn^i(xs) \rightarrow \langle ys, w \rangle \]

E.g., given \( s = \) \[\text{person}(\text{john}, \text{smith}), \text{city}(\text{london}), \text{person}(\text{ada}, \text{lovelace})\] , we have
\[ fn^i(s) = \langle [\text{john}, \text{ada}], \beta_2(\text{smith}, \beta_3(\text{london}, \beta_2(\text{lovelace}, \beta_1))) \rangle \]

a complement!

(according to [Bancilhon & Spyrouatos, 1981])
Introduction

Functional
Landauer embedding transformations
application: Bx

Concurrent
syntax (sequential)
syntax (concurrent)
core Erlang
semantics
reversible semantics

Application:
reversible debugging
logging semantics
causal consistency
replay semantics
controlled semantics
reversible debugging

Recap

Stepwise approach to bidirectionalization

\[
\begin{align*}
\text{fn}^i([ ]) & \rightarrow \langle [ ], \beta_1 \rangle \\
\text{fn}^i(\text{person}(n, l):xs) & \rightarrow \langle n:ys, \beta_2(l, w) \rangle \iff \text{fn}^i(xs) \rightarrow \langle ys, w \rangle \\
\text{fn}^i(\text{city}(c):xs) & \rightarrow \langle ys, \beta_3(c, w) \rangle \iff \text{fn}^i(xs) \rightarrow \langle ys, w \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{fn}^{-1}([ ], \beta_1) & \rightarrow [ ] \\
\text{fn}^{-1}(n:ys, \beta_2(l, w)) & \rightarrow \text{person}(n, l):xs \iff \text{fn}^{-1}(ys, w) \rightarrow xs \\
\text{fn}^{-1}(ys, \beta_3(c, w)) & \rightarrow \text{city}(c):xs \iff \text{fn}^{-1}(ys, w) \rightarrow xs \\
\end{align*}
\]

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

\[
\text{put}_f(v, s) \rightarrow s' \iff f^i(s) \rightarrow \langle _, \pi \rangle, f^{-1}(v, \pi) \rightarrow s'
\]
Stepwise approach to bidirectionalization

\[
\begin{align*}
\text{fn}^i([ ]) & \rightarrow \langle [ ], \beta_1 \rangle \\
\text{fn}^i(\text{person}(n, l):xs) & \rightarrow \langle n:ys, \beta_2(l, w) \rangle \iff \text{fn}^i(xs) \rightarrow \langle ys, w \rangle \\
\text{fn}^i(\text{city}(c):xs) & \rightarrow \langle ys, \beta_3(c, w) \rangle \iff \text{fn}^i(xs) \rightarrow \langle ys, w \rangle
\end{align*}
\]

\[
\begin{align*}
\text{fn}^{-1}( [ ], \beta_1 ) & \rightarrow [ ] \\
\text{fn}^{-1}(n:ys, \beta_2(l, w)) & \rightarrow \text{person}(n, l):xs \iff \text{fn}^{-1}(ys, w) \rightarrow xs \\
\text{fn}^{-1}(ys, \beta_3(c, w)) & \rightarrow \text{city}(c):xs \iff \text{fn}^{-1}(ys, w) \rightarrow xs
\end{align*}
\]

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

\[
\text{put}_f(v, s) \rightarrow s' \iff \left\{ \text{fn}^i(s) \rightarrow \langle \_ , \pi \rangle , \text{fn}^{-1}(v, \pi) \rightarrow s' \right\}
\]

1. compute the complement of the original source
**Stepwise approach to bidirectionalization**

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

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

\[
\text{put}_f(v, s) \rightarrow (s') \leftarrow f^i(s) \rightarrow \langle _, \pi \rangle, \quad f^{-1}(v, \pi) \rightarrow s'
\]
Given, $s = [\text{person(john, smith), city(london), person(ada, lovelace)}]$ and

$$fn^i(s) = \langle [\text{john, ada}], \beta_2(\text{smith, } \beta_3(\text{london, } \beta_2(\text{lovelace, } \beta_1))) \rangle$$

**Update 1 (compatible)**

$[\text{john, ada}] \Rightarrow [\text{peter, ada}] (v_1)$

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

**Update 2 (non-compatible)**

$[\text{john, ada}] \Rightarrow [\text{john}] (v_2)$

$$fn^{-1}(v_2, \pi) \text{ undefined}$$
Given, \( s = [\text{person(john, smith)}, \text{city(london)}, \text{person(ada, lovelace)}] \)
and
\[
fn^i(s) = \langle [\text{john, ada}], \beta_2(\text{smith}, \beta_3(\text{london}, \beta_2(\text{lovelace}, \beta_1)))) \rangle
\]

**Update 1 (compatible)**

\[
[\text{john, ada}] \Rightarrow [\text{peter, ada}] \ (v_1)
\]

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

**Update 2 (non-compatible)**

\[
[\text{john, ada}] \Rightarrow [\text{john}] \ (v_2)
\]

\[
fn^{-1}(v_2, \pi) \text{ undefined}
\]
Characterizing compatible updates [RC 2019]

**Definition (view skeleton)**

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

We compute the narrowing derivation:

\[ f^{-1}(x, \pi) \rightsquigarrow^*_\sigma s' \] (deterministic!)

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

E.g.,

\[ fn^{-1}(x, \beta_2(smith, \beta_3(london, \beta_2(lovelace, \beta_1)))) \rightsquigarrow^*_\{x\mapsto[x_1,x_2]\} s' \]

Therefore, the view skeleton is \([x_1, x_2]\)

**Consider a source \( s \) with \( f^i(s) = \langle v, \pi \rangle \)**

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 $f^i(s) = \langle v, \pi \rangle$

We compute the narrowing derivation:

$f^{-1}(x, \pi) \sim_{\sigma}^* s' \quad \text{(deterministic!)}$

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

E.g.,

$f_{n^{-1}}(x, \beta_2(\text{smith}, \beta_3(\text{london}, \beta_2(\text{lovelace}, \beta_1)))) \sim_{\{x \mapsto [x_1, x_2]\}}^* s'$

Therefore, the view skeleton is $[x_1, x_2]$

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

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
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
Introduction

Functional
Landauer embedding transformations application: Bx

Concurrent
syntax (sequential)
syntax (concurrent)
core Erlang semantics
reversible semantics

Application: reversible debugging
logging semantics causal consistency replay semantics controlled semantics reversible semantics

Recap

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**

```erlang
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:

```erlang
append(A, B) -> case A of
                   [H|T] -> [H|append(T, L)];
                   [] -> L
                  end.
```
Sequential Erlang in 5 examples

toint/1

\[
\begin{align*}
toint({s,N}) & \rightarrow \ \text{int}(N) + 1; \\
toint(\text{zero}) & \rightarrow \ 0.
\end{align*}
\]

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

No user-defined algebraic data types (so we cannot write \(s(s(s(\text{zero})))\))

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**

\[
\text{minmax}(L) \rightarrow \begin{cases} 
\text{Min} = \text{lists:} \text{min}(L), \\
\text{Max} = \text{lists:} \text{max}(L), \\
\{\text{Min}, \text{Max}\}.
\end{cases}
\]

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

Equation \(\text{pat} = \text{exp}\) evaluates \(\text{exp}\) and perform pattern matching with \(\text{pat}\)

Equivalent to

\[
\text{minmax}(L) \rightarrow \{\text{Min}, \text{Max}\} \leftarrow \text{lists:} \text{min}(L) \rightarrow \text{Min}, \\
\text{lists:} \text{max}(L) \rightarrow \text{Max}
\]
Sequential Erlang in 5 examples

minmax/1

\[
\text{minmax}(L) \rightarrow \begin{align*}
\text{Min} &= \text{lists:}\text{min}(L), \\
\text{Max} &= \text{lists:}\text{max}(L), \\
\{\text{Min}, \text{Max}\}.
\end{align*}
\]

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

Equation \(\text{pat} = \text{exp}\) evaluates \(\text{exp}\) and perform pattern matching with \(\text{pat}\)

Equivalent to

\[
\text{minmax}(L) \rightarrow \{\text{Min}, \text{Max}\} \leftarrow \text{lists:}\text{min}(L) \rightarrow \text{Min}, \\
\text{lists:}\text{max}(L) \rightarrow \text{Max}
\]
Sequential Erlang in 5 examples

inclist/1

\[
inclist(L) \rightarrow \text{lists:map}(\text{fun}(X) \rightarrow X + 1 \text{ end}, L).
\]

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.
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**

\[
a(42) \rightarrow \text{ok}; \\
a(N) \rightarrow M = N + 1, a(M).
\]

**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**

\[
a(42) \rightarrow \text{ok}; \\
a(N) \rightarrow M = N + 1, a(M).
\]

**core erlang**

\[
'a'/1 = \text{fun}(_@c0) \rightarrow \\
\text{case } _@c0 \text{ of} \\
< 42 > \text{ when 'true'} \rightarrow 'ok' \\
<_@c2 > \text{ when 'true'} \rightarrow \text{let } < _@c3 > = \text{call 'erlang':'+'}(N,1) \\
\text{ in apply } 'a'/1(_@c3) \\
\text{end}
\]

**Essentially:** one clause per function, case for pattern matching, let for sequences, apply for function applications, \ldots
We consider a subset of Core Erlang with this syntax:

\[
\begin{align*}
\text{Module} &::= \text{module } \text{Atom} = \text{fun}1, \ldots, \text{fun}n \\
\text{fun} &::= \text{fname} = \text{fun} (X_1, \ldots, X_n) \rightarrow \text{expr} \\
\text{fname} &::= \text{Atom}/\text{Integer} \\
\text{lit} &::= \text{Atom} | \text{Integer} | \text{Float} | [] \\
\text{expr} &::= \text{Var} | \text{lit} | \text{fname} | [\text{expr}_1|\text{expr}_2] | \{\text{expr}_1, \ldots, \text{expr}_n\} \\
&\quad | \text{call } \text{expr} (\text{expr}_1, \ldots, \text{expr}_n) \mid \text{apply } \text{expr} (\text{expr}_1, \ldots, \text{expr}_n) \\
&\quad | \text{case } \text{expr} \text{ of } \text{clause}_1; \ldots; \text{clause}_m \text{ end} \\
&\quad | \text{let } \text{Var} = \text{expr}_1 \text{ in } \text{expr}_2 \mid \text{receive } \text{clause}_1; \ldots; \text{clause}_n \text{ end} \\
&\quad | \text{spawn} (\text{expr}, [\text{expr}_1, \ldots, \text{expr}_n]) \mid \text{expr}_1 ! \text{expr}_2 \mid \text{self}() \\
\text{clause} &::= \text{pat} \text{ when } \text{expr}_1 \rightarrow \text{expr}_2 \\
\text{pat} &::= \text{Var} | \text{lit} | [\text{pat}_1|\text{pat}_2] \mid \{\text{pat}_1, \ldots, \text{pat}_n\}
\end{align*}
\]
We consider a subset of Core Erlang with this syntax:

```
Module ::= module Atom = fun1, . . . , funn
fun  ::=  fname = fun (X1, . . . , Xn) → expr
fname ::= Atom / Integer
lit  ::=  Atom | Integer | Float | []
expr ::= Var | lit | fname | [expr1|expr2] | {expr1, . . . , exprn}
       | call expr (expr1, . . . , exprn) | apply expr (expr1, . . . , exprn)
       | case expr of clause1; . . . ; clause m end
       | let Var = expr1 in expr2 | receive clause1; . . . ; clause n end
       | spawn(expr, [expr1, . . . , exprn]) | expr1 ! expr2 | self()
clause ::= pat when expr1 → expr2
pat   ::=  Var | lit | [pat1|pat2] | {pat1, . . . , patn}
```
Some preliminary definitions

### Definition (process)

A process is a triple \( \langle p, \theta, e \rangle \) where
- \( p \) is the pid of the process
- \( \theta \) is an environment
- \( e \) is the expression to be reduced

//no local queue!

### Definition (system)

A system is denoted by \( \Gamma; \Pi \), where
- \( \Gamma \) models the network & local queues (global mailbox); a multiset of triples \((sender\_pid, target\_pid, message)\)
- \( \Pi \) is a pool of processes

We use \( \Gamma; \langle p, \theta, e \rangle \& \Pi \) to denote an arbitrary system.
Definition (process) //no local queue!
A process is a triple $\langle p, \theta, e \rangle$ where
- $p$ is the pid of the process
- $\theta$ is an environment
- $e$ is the expression to be reduced

Definition (system)
A system is denoted by $\Gamma; \Pi$, where
- $\Gamma$ models the network & local queues (global mailbox); a multiset of triples $(\text{sender}_\text{pid}, \text{target}_\text{pid}, \text{message})$
- $\Pi$ is a pool of processes

We use $\Gamma; \langle p, \theta, e \rangle \& \Pi$ to denote an arbitrary system
Erlang guarantees that, if two messages are sent from process $\rho$ to process $\rho'$, and both are delivered, then the order of these messages is kept.

1. [LOPSTR16] ensures this restriction.

2. [JLAMP18,FLOPS18,FORTE19] ignore this restriction.
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
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)

**Standard sem (systems)**

- **Sequential exps**
- **Concurrent exps**

**Introduction**

- Functional
  - Landauer embedding
  - Transformations
  - Application: Bx

**Concurrent**

- Syntax (sequential)
- Syntax (concurrent)
- Core Erlang
- Semantics
  - Reversible semantics

**Application: reversible debugging**

- Logging semantics
- Causal consistency
- Replay semantics
- Controlled semantics
- Reversible debugging

**Recap**
**Introduction**

**Functional**
- Landauer embedding
- transformations
- application: Bx

**Concurrent**
- syntax (sequential)
- syntax (concurrent)
- core Erlang
- **semantics**
- reversible semantics

**Application:**
- reversible debugging
- logging semantics
- causal consistency
- replay semantics
- controlled semantics
- reversible debugging

**Recap**

---

**Reduction semantics (layers)**

- **reversible sem (systems)**
- sequential exps
- concurrent exps
Introduction

Functional
Landauer embedding
transformations
application: Bx

Concurrent
syntax (sequential)
syntax (concurrent)
core Erlang semantics
reversible semantics

Application: reversible debugging
logging semantics
causal consistency
replay semantics
controlled semantics
reversible debugging

Recap

Reduction semantics (layers)

controlled semantics
reversible sem (systems)
sequential exps
concurrent exps
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
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
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

(Var) \[ \begin{array}{c}
\theta, X \xrightarrow{\ell} \theta, \theta(X)
\end{array} \]

(Tuple) \[ \begin{array}{c}
\theta, \{ v_{1,i-1}, e_i, e_{i+1,n} \} \xrightarrow{\ell} \theta', \{ v_{1,i-1}, e'_i, e_{i+1,n} \}
\end{array} \]

(List1) \[ \begin{array}{c}
\theta, e_1 \xrightarrow{\ell} \theta', e'_1
\end{array} \]

\[ \begin{array}{c}
\theta, [e_1 | e_2] \xrightarrow{\ell} \theta', [e'_1 | e_2]
\end{array} \]

(List2) \[ \begin{array}{c}
\theta, e_2 \xrightarrow{\ell} \theta', e'_2
\end{array} \]

\[ \begin{array}{c}
\theta, [v_1 | e_2] \xrightarrow{\ell} \theta', [v_1 | e'_2]
\end{array} \]

(Let1) \[ \begin{array}{c}
\theta, e_1 \xrightarrow{\ell} \theta', e'_1
\end{array} \]

\[ \begin{array}{c}
\theta, \text{let } X = e_1 \text{ in } e_2 \xrightarrow{\ell} \theta', \text{let } X = e'_1 \text{ in } e_2
\end{array} \]

(Let2) \[ \begin{array}{c}
\theta, e \xrightarrow{\ell} \theta', e'
\end{array} \]

\[ \begin{array}{c}
\theta, \text{let } X = v \text{ in } e \xrightarrow{\tau} \theta[X \mapsto v], e
\end{array} \]

(Case1) \[ \begin{array}{c}
\theta, e \xrightarrow{\ell} \theta', e'
\end{array} \]

\[ \begin{array}{c}
\theta, \text{case } e \text{ of } cl_1 ; \ldots ; cl_n \text{ end} \xrightarrow{\ell} \theta', \text{case } e' \text{ of } cl_1 ; \ldots ; cl_n \text{ end}
\end{array} \]

(Case2) \[ \begin{array}{c}
\theta, e \xrightarrow{\ell} \theta', e'
\end{array} \]

\[ \begin{array}{c}
\text{match}(v, cl_1, \ldots, cl_n) = \langle \theta_i, e_i \rangle
\end{array} \]

\[ \begin{array}{c}
\theta, \text{case } v \text{ of } cl_1 ; \ldots ; cl_n \text{ end} \xrightarrow{\tau} \theta \theta_i, e_i
\end{array} \]

(Apply1) \[ \begin{array}{c}
\theta, e_i \xrightarrow{\ell} \theta', e'_i
\end{array} \]

\[ \begin{array}{c}
i \in \{1, \ldots, n\}
\end{array} \]

\[ \begin{array}{c}
\theta, \text{apply } a/n (v_{1,i-1}, e_i, e_{i+1,n}) \xrightarrow{\ell} \theta', \text{apply } a/n (v_{1,i-1}, e'_i, e_{i+1,n})
\end{array} \]

(Apply2) \[ \begin{array}{c}
\mu(a/n) = \text{fun } (X_1, \ldots, X_n) \rightarrow e
\end{array} \]

\[ \begin{array}{c}
\theta, \text{apply } a/n (v_1, \ldots, v_n) \xrightarrow{\tau} \{ X_1 \mapsto v_1, \ldots, X_n \mapsto v_n \}, e
\end{array} \]
Sending a message

(expression semantics)

\[(\text{Send}1) \quad \theta, e_1 \xrightarrow{\ell} \theta', e'_1 \quad \theta, e_2 \xrightarrow{\ell} \theta', e'_2 \]

\[\theta, e_1 ! e_2 \xrightarrow{\ell} \theta', e'_1 ! e_2 \quad \theta, v_1 ! e_2 \xrightarrow{\ell} \theta', v_1 ! e'_2 \]

\[(\text{Send}2) \quad \theta, v_1 ! v_2 \xrightarrow{\text{send}(v_1, v_2)} \theta, v_2 \]

(system semantics)

\[(\text{Send}) \quad \theta, e \xrightarrow{\text{send}(p', v)} \theta', e' \]

\[\Gamma; \langle p, \theta, e \rangle \& \Pi \leftrightarrow \Gamma \cup \{(p, p', v)\}; \langle p, \theta', e' \rangle \& \Pi \]
Sending a message

**Expression semantics**

\[(\text{Send1})\quad \theta, e_1 \xrightarrow{\ell} \theta', e'_1 \quad \theta, e_2 \xrightarrow{\ell} \theta', e'_2 \]

\[\theta, e_1 \oplus e_2 \xrightarrow{\ell} \theta', e'_1 \oplus e_2 \quad \theta, v_1 \oplus e_2 \xrightarrow{\ell} \theta', v_1 \oplus e'_2 \]

**System semantics**

\[(\text{Send})\quad \theta, e \xrightarrow{\text{send}(p',v)} \theta', e' \]

\[\Gamma; \langle p, \theta, e \rangle \& \Pi \leftrightarrow \Gamma \cup \{(p, p', v)\}; \langle p, \theta', e' \rangle \& \Pi \]
Introduction

Functional
- Landauer embedding
- transformations
- application: Bx

Concurrent
- syntax (sequential)
- syntax (concurrent)
- core Erlang
- semantics
- reversible semantics

Application: reversible debugging
- logging semantics
- causal consistency
- replay semantics
- controlled semantics
- reversible debugging

Recap

(expression semantics)

\[(Self) \quad \theta, \text{self}(\kappa) \rightarrow \theta, \kappa\]

(system semantics)

\[(Self) \quad \theta, e \rightarrow \theta', e'\]

\[\Gamma; \langle p, \theta, e \rangle \& \Pi \leftrightarrow \Gamma; \langle p, \theta', e'\{\kappa \mapsto p} \rangle \& \Pi\]
Expression semantics:

\[
\text{(Self)} \quad \theta, \text{self(κ)} \, \xrightarrow{\text{self(κ)}} \, \theta, \kappa
\]

System semantics:

\[
\text{(Self)} \quad \theta, e \, \xrightarrow{\text{self(κ)}} \, \theta', e'
\]

\[
\Gamma; \langle p, \theta, e \rangle \& \Pi \leftrightarrow \Gamma'; \quad \langle p, \theta', e' \{\kappa \mapsto p\} \rangle \& \Pi
\]
(expression semantics)

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

(system semantics)

(Spawn) \( \Gamma; \langle p, \theta, e \rangle & \Pi \mapsto \Gamma; \langle p, \theta', e' \{ \kappa \mapsto p' \} \rangle & \Pi \)
\( \langle p', \theta', \text{apply } a/n (\overline{v_n}) \rangle & \Pi \)
Spawning a process

(expression semantics)

\[(\text{Spawn}) \quad \theta, \text{spawn}(a/n, [v_1, \ldots, v_n]) \xrightarrow{\text{spawn}(\kappa, a/n, [\overline{v_n}])} \theta, \kappa\]

(system semantics)

\[(\text{Spawn}) \quad \theta, e \xrightarrow{\text{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}\]

\[\Gamma; \langle p, \theta, e \rangle \& \Pi \leftrightarrow \Gamma; \quad \langle p, \theta', e'\{\kappa \mapsto p'\}\rangle \& \langle p', \theta', \text{apply } a/n (\overline{v_n}) \rangle \& \Pi\]
Receiving a message

(expression semantics)

\[(\text{Receive}) \quad \theta, \text{receive } cl_1; \ldots; cl_n \text{ end} \xrightarrow{\text{rec}(\kappa, cl_n)} \theta, \kappa\]

(system semantics)

\[(\text{Receive}) \quad \theta, e \xrightarrow{\text{rec}(\kappa, cl_n)} \theta', e' \quad \text{matchrec}(\theta, cl_n, v) = (\theta_i, e_i) \quad \Gamma \cup \{(p', p, v)\}; \langle p, \theta, e \rangle \Pi \rightarrow \Gamma; \langle p, \theta' \theta_i, e' \{\kappa \mapsto e_i\} \rangle \Pi\]
Receiving a message

(expression semantics)

\[
\begin{align*}
\text{(Receive)} & : \theta, \text{receive } cl_1; \ldots; cl_n \text{ end} \xrightarrow{\text{rec}(\kappa, cl_n)} \theta, \kappa
\end{align*}
\]

(system semantics)

\[
\begin{align*}
\text{(Receive)} & : \theta, e \xrightarrow{\text{rec}(\kappa, cl_n)} \theta', e' \text{ matchrec}(\theta, cl_n, v) = (\theta_i, e_i) \\
& \quad \Gamma \cup \{(p', p, v)\}; \langle p, \theta, e \rangle & \Pi \leftrightarrow \Gamma '; \langle p, \theta' \theta_i, e' \{\kappa \mapsto e_i\} \rangle & \Pi
\end{align*}
\]
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
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 \( \text{seq, send, rec, spawn} \), and \( \text{self} \), and whose arguments are the information required to (deterministically) undo the step
Uncontrolled forward semantics

(Send)

\[
\begin{array}{c}
\theta, e \xrightarrow{\text{send}(p', v)} \theta', e' \\
\Gamma; \langle p, h, \theta, e \rangle | \Pi \\
\rightarrow p, \text{send}(\ell) \Gamma \cup \{(p, p', v)\}; \\
\langle p, \text{send}(\theta, e, p', v) : h, \theta', e' \rangle | \Pi
\end{array}
\]

(Receive)

\[
\begin{array}{c}
\theta, e \xrightarrow{\text{rec}(\kappa, \overline{c_l}_n)} \theta', e' \text{ and } \text{matchrec}(\theta, \overline{c_l}_n, v) = (\theta_i, e_i) \\
\Gamma \cup \{(p', p, v)\} \langle p, h, \theta, e \rangle | \Pi \\
\rightarrow p, \text{rec}(\ell) \Gamma; \langle p, \text{rec}(\theta, e, p', v) : h, \theta' \theta_i, e' \{\kappa \mapsto e_i\} \rangle | \Pi
\end{array}
\]

(Spawn)

\[
\begin{array}{c}
\theta, e \xrightarrow{\text{spawn}(\kappa, a/n, [v_n])} \theta', e' \text{ and } p' \text{ is a fresh pid} \\
\Gamma; \langle p, h, \theta, e \rangle | \Pi \\
\rightarrow p, \text{spawn}(p') \Gamma; \langle p, \text{spawn}(\theta, e, p') : h, \theta', e' \{\kappa \mapsto p'\} \rangle \\
\quad | \langle p', [\ ], id, \text{apply } a/n (v_n) \rangle | \Pi
\end{array}
\]
Uncontrolled backward semantics

\[ (\text{Send}) \quad \Gamma \cup \{(p, p', v)\}; \langle p, \text{send} (\theta, e, p', v): h, \theta', e' \rangle | \Pi \]
\[ \leftarrow p, \text{send}(\ell) \Gamma; \langle p, h, \theta, e \rangle | \Pi \]

\[ (\text{Receive}) \quad \Gamma; \langle p, \text{rec} (\theta, e, p', v): h, \theta', e' \rangle | \Pi \]
\[ \leftarrow p, \text{rec}(\ell) \Gamma \cup \{(p', p, v)\}; \langle p, h, \theta, e \rangle | \Pi \]
where \( V = \text{Dom}(\theta') \setminus \text{Dom}(\theta) \)

\[ (\text{Spawn}) \quad \Gamma; \langle p, \text{spawn} (\theta, e, p'): h, \theta', e' \rangle | \langle p', [ ], id, e'' \rangle | \Pi \]
\[ \leftarrow p, \text{spawn}(p') \Gamma; \langle p, h, \theta, e \rangle | \Pi \]

\( \Rightarrow \) reversible computations must be causal consistent
Uncontrolled backward semantics

\[ \begin{align*}
\text{(Send)} & \quad \Gamma \cup \{(p, p', v)\}; \langle p, \text{send}(\theta, e, p', v) : h, \theta', e' \rangle \mid \Pi \\
& \quad \leftarrow_{p,\text{send}(\ell)} \Gamma; \langle p, h, \theta, e \rangle \mid \Pi
\end{align*} \]

\[ \begin{align*}
\text{(Receive)} & \quad \Gamma; \langle p, \text{rec}(\theta, e, p', v) : h, \theta', e' \rangle \mid \Pi \\
& \quad \leftarrow_{p,\text{rec}(\ell)} \Gamma \cup \{(p', p, v)\}; \langle p, h, \theta, e \rangle \mid \Pi \\
& \quad \text{where } V = \text{Dom}(\theta') \setminus \text{Dom}(\theta)
\end{align*} \]

\[ \begin{align*}
\text{(Spawn)} & \quad \Gamma; \langle p, \text{spawn}(\theta, e, p') : h, \theta', e' \rangle \mid \langle p', [], id, e'' \rangle \mid \Pi \\
& \quad \leftarrow_{p,\text{spawn}(\ell')} \Gamma; \langle p, h, \theta, e \rangle \mid \Pi
\end{align*} \]

\[ \Rightarrow \text{ reversible computations must be causal consistent} \]
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).
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)
We tag messages with unique identifiers

\[ v \mapsto \{ v, \ell \}, \quad \text{where } \ell \text{ is fresh} \]

A log \( \mathcal{L}(d) \) of a derivation \( d \) is a sequence of items

- \( \text{spawn}(p) \), \( \text{send}(\ell) \) or \( \text{rec}(\ell) \) for each process in \( d \)

(logs are local to each process)
(Seq)

\[
\begin{align*}
\theta, e & \xrightarrow{\tau} \theta', e' \\
\Gamma; \langle p, \theta, e \rangle \mid \Pi & \xrightarrow{p, \text{seq}} \Gamma; \langle p', \theta', e' \rangle \mid \Pi
\end{align*}
\]

(Send)

\[
\begin{align*}
\theta, e & \xrightarrow{\text{send}(p', \nu)} \theta', e' \text{ and } \ell \text{ is a fresh symbol} \\
\Gamma; \langle p, \theta, e \rangle \mid \Pi & \xrightarrow{p, \text{send}(\ell)} \Gamma \cup \{ (p, p', \{ \nu, \ell \}) \}; \langle p, \theta', e' \rangle \mid \Pi
\end{align*}
\]

(Receive)

\[
\begin{align*}
\theta, e & \xrightarrow{\text{rec}(\kappa, \overline{c_n})} \theta', e' \text{ and } \text{matchrec}(\theta, \overline{c_n}, \nu) = (\theta_i, e_i) \\
\Gamma \cup \{ (p', p, \{ \nu, \ell \}) \}; \langle p, \theta, e \rangle \mid \Pi & \xrightarrow{p, \text{rec}(\ell)} \Gamma; \langle p, \theta' \theta_i, e' \{ \kappa \mapsto e_i \} \rangle \mid \Pi
\end{align*}
\]

(Spawn)

\[
\begin{align*}
\theta, e & \xrightarrow{\text{spawn}(\kappa, a/n, [\nu_n])} \theta', e' \text{ and } p' \text{ is a fresh pid} \\
\Gamma; \langle p, \theta, e \rangle \mid \Pi & \xrightarrow{p, \text{spawn}(p')} \Gamma; \langle p, \theta', e' \{ \kappa \mapsto p' \} \rangle \mid \langle p', \text{id}, \text{apply} \ a/n \ (\overline{v_n}) \rangle \mid \Pi
\end{align*}
\]

(Self)

\[
\begin{align*}
\theta, e & \xrightarrow{\text{self}(\kappa)} \theta', e' \\
\Gamma; \langle p, \theta, e \rangle \mid \Pi & \xrightarrow{p, \text{self}} \Gamma; \langle p, \theta', e' \{ \kappa \mapsto p \} \rangle \mid \Pi
\end{align*}
\]

(implemented by a program instrumentation)
Traditional reversible debuggers allow us to go backwards in exactly the inverse order of the forward computation.

If \( p_1 \) and \( p_2 \) are independent then

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

If $p_1$ and $p_2$ 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 \textbf{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 \(d_2\),

\[ \mathcal{L}(d_1) = \mathcal{L}(d_2) \text{ iff } d_1 \approx d_2 \]
Causally equivalent derivations

\[ d_1 \text{ and } d_2 \text{ are causally equivalent } (d_1 \approx d_2) \text{ if } d_1 \text{ can be obtained from } d_2 \text{ by switching consecutive transitions as long as}\]

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 \(d_2\), \[\mathcal{L}(d_1) = \mathcal{L}(d_2) \iff d_1 \approx d_2\]
\( d_1 \) 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 \( d_2 \), \( \mathcal{L}(d_1) = \mathcal{L}(d_2) \) iff \( d_1 \approx d_2 \)
Causally equivalent derivations

\[ d_1 \text{ and } d_2 \text{ are causally equivalent } (d_1 \approx d_2) \text{ if } d_1 \text{ can be obtained from } d_2 \text{ by switching consecutive transitions as long as} \]

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 \( d_2 \), \[ \mathcal{L}(d_1) = \mathcal{L}(d_2) \text{ iff } d_1 \approx d_2 \]
Processes have the form $\langle p, \omega, h, \theta, e \rangle$

with $\omega$ a log and $h$ a history

A history $h$ is a sequence of terms headed by constructors $\text{seq, send, rec, spawn, and self}$, and whose arguments are the information required to (deterministically) undo the step
### Uncontrolled forward semantics

**Send**

\[ \theta, e \xrightarrow{\text{send}(p', v)} \theta', e' \]

\[ \Gamma; \langle p, \text{send}(\ell): \omega, h, \theta, e \rangle | \Pi \]

\[ \vdash_p \text{send}(\ell), \{ s, \ell \uparrow \} \Gamma \cup \{(p, p', \{ v, \ell \})\} ; \]

\[ \langle p, \omega, \text{send}(\theta, e, p', \{ v, \ell \}): h, \theta', e' \rangle | \Pi \]

**Receive**

\[ \theta, e \xrightarrow{\text{rec}(\kappa, cl_n)} \theta', e' \text{ and } \text{matchrec}(\theta, cl_n, v) = (\theta_i, e_i) \]

\[ \Gamma \cup \{(p', p, \{ v, \ell \})\} \langle p, \text{rec}(\ell): \omega, h, \theta, e \rangle | \Pi \]

\[ \vdash_p \text{rec}(\ell), \{ s, \ell \Downarrow \} \Gamma; \langle p, \omega, \text{rec}(\theta, e, p', \{ v, \ell \}): h, \theta \theta_i, e' \{ \kappa \mapsto e_i \} \rangle | \Pi \]

**Spawn**

\[ \theta, e \xrightarrow{\text{spawn}(\kappa, a/n, [v_n])} \theta', e' \text{ and } \omega' = \text{trace}(d, p') \]

\[ \Gamma; \langle p, \text{spawn}(p'): \omega, h, \theta, e \rangle | \Pi \]

\[ \vdash_p \text{spawn}(p'), \{ s, sp_p \} \Gamma; \langle p, \omega, \text{spawn}(\theta, e, p'): h, \theta', e' \{ \kappa \mapsto p' \} \rangle \]

\[ | \langle p', \omega', [], id, \text{apply } a/n (v_n) \rangle | \Pi \]
Uncontrolled backward semantics

\[(\text{Send})\quad \Gamma \cup \{(p, p', \{v, \ell\})\}; \langle p, \omega, \text{send}(\theta, e, p', \{v, \ell\}) : h, \theta', e' \rangle | \Pi \]
\[\leftarrow p, \text{send}(\ell), \{s, \ell'\} \quad \Gamma; \langle p, \text{send}(\ell) : \omega, h, \theta, e \rangle | \Pi\]

\[(\text{Receive})\quad \Gamma; \langle p, \omega, \text{rec}(\theta, e, p', \{v, \ell\}) : h, \theta', e' \rangle | \Pi \]
\[\leftarrow p, \text{rec}(\ell), \{s, \ell''\} \cup \mathcal{V} \quad \Gamma \cup \{(p', p, \{v, \ell\})\}; \langle p, \text{rec}(\ell) : \omega, h, \theta, e \rangle | \Pi\]
\[\text{where } \mathcal{V} = \text{Dom}(\theta') \setminus \text{Dom}(\theta)\]

\[(\text{Spawn})\quad \Gamma; \langle p, \omega, \text{spawn}(\theta, e, p') : h, \theta', e' \rangle | \langle p', \omega', [\_], \text{id}, e'' \rangle | \Pi \]
\[\leftarrow p, \text{spawn}(p'), \{s, \text{sp}_p\} \quad \Gamma; \langle p, \text{spawn}(p') : \omega, h, \theta, e \rangle | \Pi\]
Coinitial derivations are cofinal iff they are causally equivalent

Misbehaviors are preserved by all causally equivalent derivations
We allow the user to start a replay/rollback until a particular action is performed, e.g.,

- \( \{p, s\} \): 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 \( \ell \)
- \( \{p, \ell_{\downarrow}\} \): a backward/forward derivation of process \( p \) up to the reception of the message tagged with \( \ell \)
- \( \{p, sp_{p'}\} \): 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 → 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 → 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 → 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
Reversible debugging

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

https://github.com/mistupv/tracer
https://github.com/mistupv/cauder/tree/replay
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
Every (irreversible) language can be made reversible by defining a Landauer embedding.
Reversibilization can be achieved by instrumenting the semantics or the program.
Reversibilization ≠ program inversion
For concurrent languages, causal consistency is essential
There are many other applications of reversible computation: quantum computing, discrete simulation, hardware design, computational biology, robotics, etc.
Thanks for your attention!

Questions?