A Reversible Semantics for Erlang

Adrián Palacios
(joint work with Ivan Lanese, Naoki Nishida and Germán Vidal)

Technical University of Valencia

STSMs in Nagoya (Japan) and Bologna (Italy)

March 30, 2017
Belgrade, Serbia

Motivation

We have recently introduced a reversible term rewriting principle.

We want to consider a programming language: **Erlang**

Why Erlang?

- Introduces concurrency, a challenge for reversibility
- Used in large-scale distributed systems
- Reversible Computation can contribute to its reliability (debugging, transactions, etc.)

Motivation

We have recently introduced a reversible term rewriting principle.

We want to consider a programming language: **Erlang**

Why Erlang?

- Introduces concurrency, a challenge for reversibility
- Used in large-scale distributed systems
- Reversible Computation can contribute to its reliability (debugging, transactions, etc.)

Erlang

Erlang's features

Main features of Erlang:

- integration of functional and concurrent features
- concurrency model based on asynchronous message-passing
- dynamic typing
- hot code loading

These features make it appropiate for distributed, fault-tolerant applications (Facebook, WhatsApp)

Erlang syntax

We consider a subset of Erlang with this syntax:

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

Erlang syntax

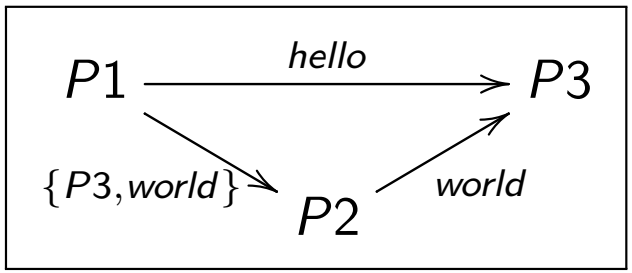
We consider a subset of Erlang with this syntax:

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

Hello, World!

$$
\begin{array}{l}
\mathit{main}/0 = \text{fun } () \to \text{let } P2 = \text{spawn}(\mathit{echo}/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let } P3 = \text{spawn}(\mathit{target}/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let } \_ = P3 \,!\, \mathit{hello} \\
\qquad\qquad\qquad\quad \text{in let } P2 \,!\, \{P3, \mathit{world}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{target}/0 = \text{fun } () \to \text{receive} \\
\qquad\qquad\qquad\qquad A \to \text{receive} \\
\qquad\qquad\qquad\qquad\qquad B \to \{A, B\} \\
\qquad\qquad\qquad\qquad \text{end} \\
\qquad\qquad\qquad \text{end}
\end{array}
$$

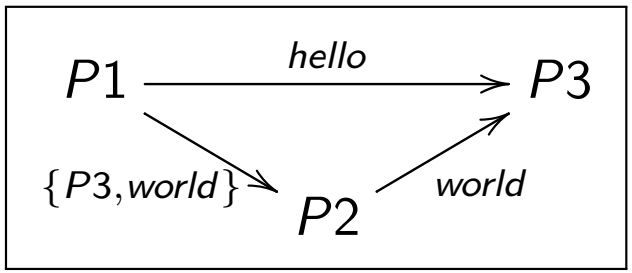


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun } () \to \text{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \to P \,!\, M \\
\qquad\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

$$
\begin{array}{l}
\mathit{main}/0 = \text{fun}\ () \rightarrow \text{let}\ P2 = \text{spawn}(\mathit{echo}/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ P3 = \text{spawn}(\mathit{target}/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ \_ = P3\ !\ \mathit{hello} \\
\qquad\qquad\qquad\quad \text{in let}\ P2\ !\ \{P3, \mathit{world}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{target}/0 = \text{fun}\ () \rightarrow \text{receive} \\
\qquad\qquad\qquad\quad A \rightarrow \text{receive} \\
\qquad\qquad\qquad\qquad\quad B \rightarrow \{A, B\} \\
\qquad\qquad\qquad\quad \text{end} \\
\qquad\qquad\quad \text{end}
\end{array}
$$

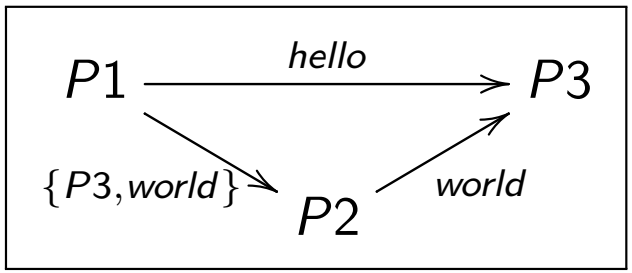


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \rightarrow \text{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \rightarrow P\ !\ M \\
\qquad\qquad\quad \text{end}
\end{array}
$$

Hello, World!

$$
\begin{array}{l}
main/0 = \text{fun}\ () \to \text{let}\ P2 = \text{spawn}(echo/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ P3 = \text{spawn}(target/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ \_ = P3\,!\,hello \\
\qquad\qquad\qquad\quad \text{in let}\ P2\,!\,\{P3, world\}
\end{array}
$$

$$
\begin{array}{l}
target/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad A \to \text{receive} \\
\qquad\qquad\qquad\qquad\quad B \to \{A, B\} \\
\qquad\qquad\qquad\quad \text{end} \\
\qquad\qquad\quad \text{end}
\end{array}
$$

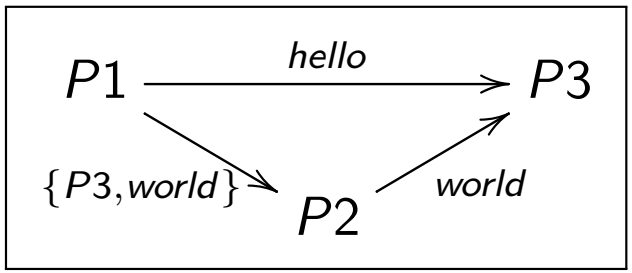


$$
\begin{array}{l}
echo/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

$$
\begin{array}{l}
main/0 = \text{fun}\ () \to \text{let}\ P2 = \text{spawn}(echo/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ P3 = \text{spawn}(target/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ \_ = P3\ !\ hello \\
\qquad\qquad\qquad\quad \text{in let}\ P2\ !\ \{P3, world\}
\end{array}
$$

$$
\begin{array}{l}
target/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad A \to \text{receive} \\
\qquad\qquad\qquad\qquad\quad B \to \{A, B\} \\
\qquad\qquad\qquad\quad \text{end} \\
\qquad\qquad\quad \text{end}
\end{array}
$$

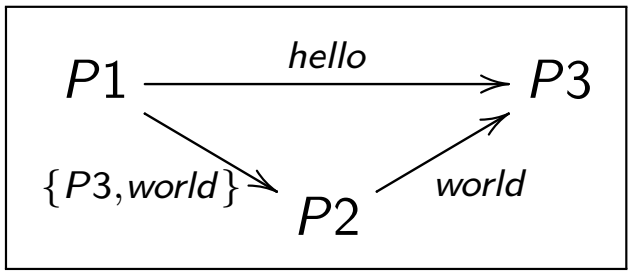


$$
\begin{array}{l}
echo/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\quad \{P, M\} \to P\ !\ M \\
\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

$$
\begin{array}{l}
\mathit{main}/0 = \mathsf{fun}\ () \to \mathsf{let}\ P2 = \mathsf{spawn}(\mathit{echo}/0, [\,]) \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ P3 = \mathsf{spawn}(\mathit{target}/0, [\,]) \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ \_ = P3\ !\ \mathit{hello} \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ P2\ !\ \{P3, \mathit{world}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{target}/0 = \mathsf{fun}\ () \to \mathsf{receive} \\
\qquad\qquad\qquad\quad A \to \mathsf{receive} \\
\qquad\qquad\qquad\qquad\quad B \to \{A, B\} \\
\qquad\qquad\qquad\quad \mathsf{end} \\
\qquad\qquad\quad \mathsf{end}
\end{array}
$$

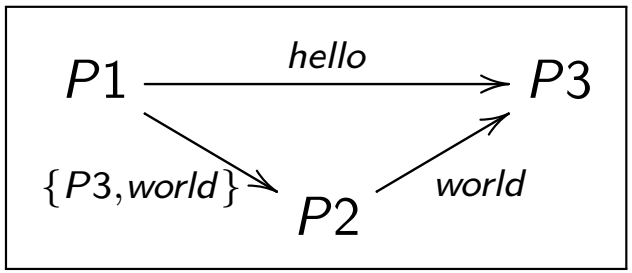


$$
\begin{array}{l}
\mathit{echo}/0 = \mathsf{fun}\ () \to \mathsf{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \to P\ !\ M \\
\qquad\qquad\quad \mathsf{end}
\end{array}
$$

Hello, World!

$$
\begin{array}{l}
\mathit{main}/0 = \mathsf{fun}\ () \to \mathsf{let}\ P2 = \mathsf{spawn}(\mathit{echo}/0, [\,]) \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ P3 = \mathsf{spawn}(\mathit{target}/0, [\,]) \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ \_ = P3\,!\,\mathit{hello} \\
\qquad\qquad\qquad\quad \mathsf{in\ let}\ P2\,!\,\{P3, \mathit{world}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{target}/0 = \mathsf{fun}\ () \to \mathsf{receive} \\
\qquad\qquad\qquad\quad A \to \mathsf{receive} \\
\qquad\qquad\qquad\qquad\quad B \to \{A, B\} \\
\qquad\qquad\qquad\qquad \mathsf{end} \\
\qquad\qquad\qquad \mathsf{end}
\end{array}
$$

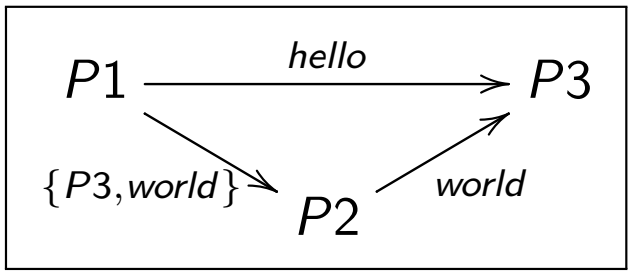


$$
\begin{array}{l}
\mathit{echo}/0 = \mathsf{fun}\ () \to \mathsf{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad\qquad \mathsf{end}
\end{array}
$$

Hello, World!

$$
\begin{array}{l}
\mathit{main}/0 = \text{fun}\ () \to \text{let}\ P2 = \text{spawn}(\mathit{echo}/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ P3 = \text{spawn}(\mathit{target}/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ \_ = P3\,!\,\mathit{hello} \\
\qquad\qquad\qquad\quad \text{in let}\ P2\,!\,\{P3, \mathit{world}\}
\end{array}
$$

$$
\begin{array}{l}
\mathit{target}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\qquad A \to \text{receive} \\
\qquad\qquad\qquad\qquad\qquad\quad B \to \{A, B\} \\
\qquad\qquad\qquad\qquad\quad \text{end} \\
\qquad\qquad\qquad \text{end}
\end{array}
$$

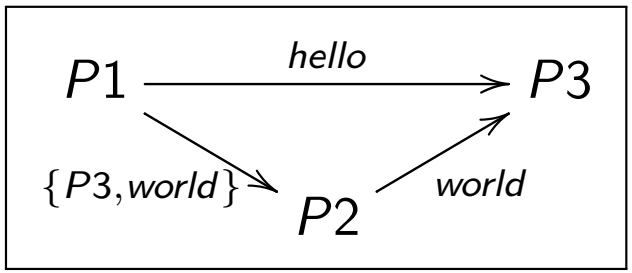


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

$$
\begin{array}{l}
main/0 = \text{fun}\ () \to \text{let}\ P2 = \text{spawn}(echo/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ P3 = \text{spawn}(target/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ \_ = P3\ !\ hello \\
\qquad\qquad\qquad\quad \text{in let}\ P2\ !\ \{P3, world\}
\end{array}
$$

$$
\begin{array}{l}
target/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad A \to \text{receive} \\
\qquad\qquad\qquad\qquad\quad B \to \{A, B\} \\
\qquad\qquad\qquad\quad \text{end} \\
\qquad\qquad\quad \text{end}
\end{array}
$$



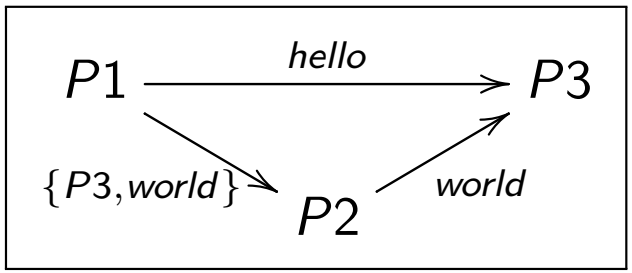
$$
\begin{array}{l}
echo/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\quad \{P, M\} \to P\ !\ M \\
\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

$$
\begin{array}{l}
main/0 = \text{fun}\ () \to \text{let}\ P2 = \text{spawn}(echo/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ P3 = \text{spawn}(target/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ \_ = P3\ !\ hello \\
\qquad\qquad\qquad\quad \text{in let}\ P2\ !\ \{P3, world\}
\end{array}
$$

$$
\begin{array}{l}
target/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\qquad A \to \text{receive} \\
\qquad\qquad\qquad\qquad\qquad\quad B \to \{A, B\} \\
\qquad\qquad\qquad\qquad\quad \text{end} \\
\qquad\qquad\qquad \text{end}
\end{array}
$$

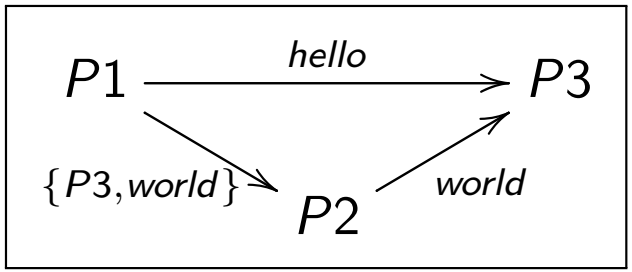
$$
\begin{array}{l}
echo/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \to P\ !\ M \\
\qquad\qquad\qquad \text{end}
\end{array}
$$



Hello, World!

$$
\begin{array}{l}
main/0 = \text{fun}\ () \to \text{let}\ P2 = \text{spawn}(echo/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ P3 = \text{spawn}(target/0, [\,]) \\
\qquad\qquad\qquad\quad \text{in let}\ \_ = P3\,!\,hello \\
\qquad\qquad\qquad\quad \text{in let}\ P2\,!\,\{P3, world\}
\end{array}
$$

$$
\begin{array}{l}
target/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad A \to \text{receive} \\
\qquad\qquad\qquad\qquad\quad B \to \{A, B\} \\
\qquad\qquad\qquad\quad \text{end} \\
\qquad\qquad\quad \text{end}
\end{array}
$$



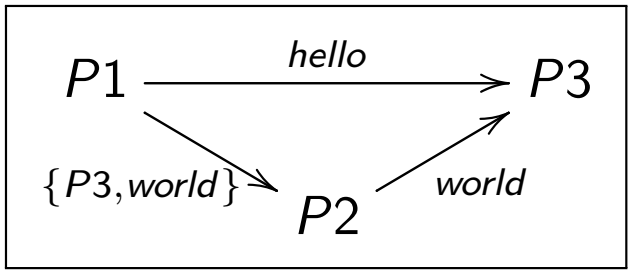
$$
\begin{array}{l}
echo/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad\quad \text{end}
\end{array}
$$

{hello,world}

Hello, World!

$$
\begin{aligned}
main/0 = \text{fun}\ () \rightarrow\ & \text{let}\ P2 = \text{spawn}(echo/0, [\,]) \\
& \text{in let}\ P3 = \text{spawn}(target/0, [\,]) \\
& \text{in let}\ \_ = P3\,!\,hello \\
& \text{in let}\ P2\,!\,\{P3, world\}
\end{aligned}
$$

$$
\begin{aligned}
target/0 = \text{fun}\ () \rightarrow\ & \text{receive} \\
& A \rightarrow \text{receive} \\
& \qquad B \rightarrow \{A, B\} \\
& \quad \text{end} \\
& \text{end}
\end{aligned}
$$



$$
\begin{aligned}
echo/0 = \text{fun}\ () \rightarrow\ & \text{receive} \\
& \{P, M\} \rightarrow P\,!\,M \\
& \text{end}
\end{aligned}
$$

{hello,world},{world,hello}

Semantics

Previous definitions

Definition (process)

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

- p is the pid of the process
- (θ, e) is the control of the state
- q is the process' local mailbox

Definition (system)

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

- Γ is the global mailbox of the system
- Π is a pool of processes

We often use $\Gamma; \langle p, (\theta, e), q \rangle \& \Pi$

Expression semantics: sequential actions

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

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

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

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

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

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

Expression semantics: concurrent problems

In concurrent actions, we face the following problems:

1. we do not 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
- At system level, labels are used to perform the associated actions

Expression semantics: concurrent problems

In concurrent actions, we face the following problems:

1. we do not 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
- At system level, labels are used to perform the associated actions

Expression semantics: rule Send

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

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

Expression semantics: rule Self

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

Expression semantics: rules Spawn and Receive

$$(\mathit{Spawn})\ \frac{}{\theta, \mathsf{spawn}(a/n, [e_1, \ldots, e_n]) \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{e_n}])} \theta, \kappa}$$

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

Semantics

System-level rules

System semantics: rule Seq

$$(Seq)\quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \& \Pi \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \& \Pi}$$

System semantics: rule Self

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

System semantics: rule Send

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

System semantics: rule Receive

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

System semantics: rule Spawn

$$(Spawn)\quad \frac{\theta, e \xrightarrow{\text{spawn}(\kappa, a/n, [\overline{e_n}])} \theta', e' \qquad p' \text{ is a fresh pid}}{\Gamma; \langle p, (\theta, e), q\rangle \& \Pi \hookrightarrow \Gamma; \quad \begin{array}{l}\langle p, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \& \\ \langle p', (\theta', \text{apply } a/n\ (\overline{e_n})), [\,]\rangle \& \Pi\end{array}}$$

System semantics: rule Sched

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, v)\}; \langle p, (\theta, e), q\rangle \& \Pi \hookrightarrow \Gamma; \langle p, (\theta, e), v : q\rangle \& \Pi}$$

Reversible Semantics

Reversible semantics

Our reversible semantics is composed of

- the forward semantics (we record steps given in h)
- the backward semantics (we use the recorded information)

Uncontrolled semantics

Rules (both forward and backward) are fired in a non-deterministic fashion

Forward (reversible) semantics

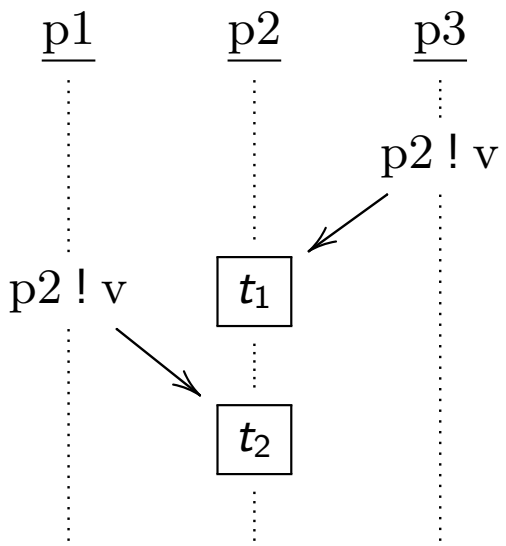
$$(Seq)\qquad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle h, p, (\theta, e), q\rangle \& \Pi \rightharpoonup \Gamma; \langle \tau(\theta, e) : h, p, (\theta', e'), q\rangle \& \Pi}$$

$$(Self)\qquad \frac{\theta, e \overset{\mathsf{self}(y)}{\rightarrow} \theta', e'}{\Gamma; \langle h, p, (\theta, e), q\rangle \& \Pi \rightharpoonup \Gamma; \langle \mathsf{self}(\theta, e) : h, p, (\theta', e'\{y \mapsto p\}), q\rangle \& \Pi}$$

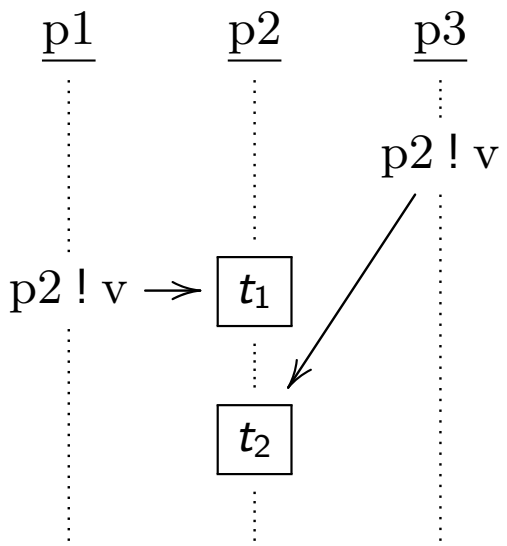
$$(Spawn)\qquad \frac{\theta, e \overset{\mathsf{spawn}(y, a/n, [e_1, \ldots, e_n])}{\rightarrow} \theta', e' \quad p' \text{ is a fresh pid}}{\begin{array}{ll}\Gamma; \langle h, p, (\theta, e), q\rangle \& \Pi \rightharpoonup & \Gamma; \langle \mathsf{spawn}(\theta, e, p') : h, p, (\theta', e'\{y \mapsto p'\}), q\rangle \\ & \& \langle [\,], p', (\theta, \mathsf{apply}\ a/n\ (e_1, \ldots, e_n)), [\,]\rangle \& \Pi\end{array}}$$

Communication with different processes

Messsages could be distinguished by attaching the pid of the processes



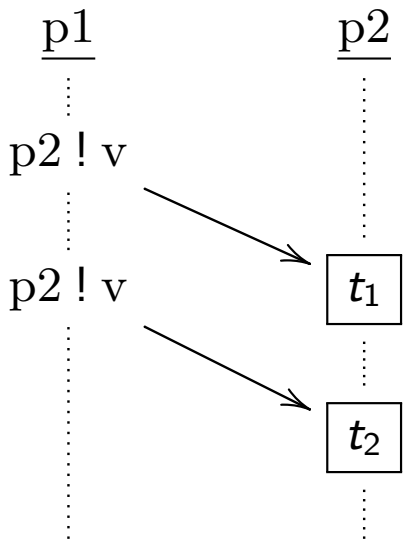
(a)



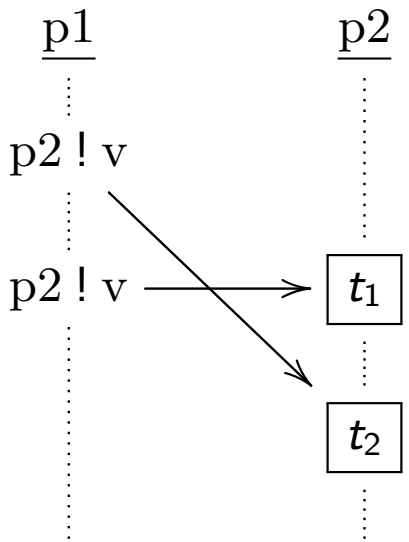
(b)

Communication with a single process

The pid of the process is not enough, we need timestamps!



(c)



(d)

Forward (reversible) semantics (2)

$$(Send)\quad \frac{\theta, e \overset{\mathsf{send}(p'', v)}{\rightarrow} \theta', e' \quad \text{and} \quad \lambda \text{ is a fresh identifier}}{\Gamma; \langle h, p, (\theta, e), q\rangle \& \Pi \rightharpoonup \Gamma \cup (p'', \{v, \lambda\}); \langle \mathsf{send}(\theta, e, \{v, \lambda\}) : h, p, (\theta', e'), q\rangle \& \Pi}$$

$$(Receive)\quad \frac{\theta, e \overset{\mathsf{rec}(y, \overline{cl_n})}{\rightarrow} \theta', e' \quad \mathsf{matchrec}(\overline{cl_n}, q) = (\theta_i, e_i, \{v, \lambda\})}{\Gamma; \langle h, p, (\theta, e), q\rangle \& \Pi \rightharpoonup \Gamma; \langle \begin{array}{l} \mathsf{rec}(\theta, e, \{v, \lambda\}, q, q \backslash\backslash \{v, k\}) : h, \\ p, (\theta' \theta_i, e' \{y \mapsto e_i\}), q \backslash\backslash \{v, k\} \end{array} \rangle \& \Pi}$$

$$(Sched)\quad \frac{}{\Gamma \cup \{(p, \{v, \lambda\})\}; \langle h, p, (\theta, e), q\rangle \& \Pi \rightharpoonup \Gamma; \langle h, p, (\theta, e), \{v, \lambda\} : q\rangle \& \Pi}$$

Backward semantics

$$(\overline{Seq}) \qquad \Gamma; \langle \tau(\theta, e) : h, p, (\theta', e'), q \rangle \ \& \ \Pi \leftharpoondown \Gamma; \langle h, p, (\theta, e), q \rangle \ \& \ \Pi$$

$$(\overline{Self}) \qquad \Gamma; \langle \mathsf{self}(\theta, e) : h, p, (\theta', e'), q \rangle \ \& \ \Pi \leftharpoondown \Gamma; \langle h, p, (\theta, e), q \rangle \ \& \ \Pi$$

$$(\overline{Spawn}) \qquad \begin{array}{l} \Gamma; \langle \mathsf{spawn}(\theta, e, p'') : h, p, (\theta, e), q \rangle \ \& \ \langle [\,], p'', (\theta'', e''), q'' \rangle \ \& \ \Pi \\ \leftharpoondown \Gamma; \langle h, p, (\theta, e), q \rangle \ \& \ \Pi \end{array}$$

Backward semantics (2)

$$(\overline{Send}) \quad \Gamma \cup \{(p', \{v, \lambda\})\}; \langle \mathsf{send}(\theta, e, \{v, \lambda\}) : h, p, (\theta', e'), q\rangle \ \&\ \Pi \leftharpoonup \Gamma; \langle h, p, (\theta, e), q\rangle \ \&\ \Pi$$

$$(\overline{Receive}) \quad \Gamma; \langle \mathsf{rec}(\theta, e, \{v, \lambda\}, q, q') : h, p, (\theta', e'), q'\rangle \ \&\ \Pi \leftharpoonup \Gamma; \langle h, p, (\theta, e), q\rangle \ \&\ \Pi$$

$$(\overline{Sched}) \quad \Gamma; \langle h, p, (\theta, e), \{v, \lambda\} : q\rangle \ \&\ \Pi \leftharpoonup \Gamma \cup (p, \{v, \lambda\}); \langle h, p, (\theta, e), q\rangle \ \&\ \Pi$$
if the topmost rec(...) item in h (if any) is different from $\mathsf{rec}(\theta', e', \{v, \lambda\}, q'', \{v, \lambda\} : q)$

Results

Theorem (conservative)

Let $\mathcal{P}$ be a program without occurrences of "check". Let s_1 be a system of the standard semantics and rs_1 a system of the reversible semantics with $\overline{rs_1} = s_1$. Then, $s_1 \longmapsto^* s_2$ iff $rs_1 \rightharpoonup^* rs_2$ and $\overline{rs_2} = s_2$.

Lemma (loop lemma)

For every pair of systems, s_1 and s_2, we have $s_1 \rightharpoonup_{p,r,k} s_2$ iff $s_2 \leftharpoondown_{p,\overline{r},k} s_1$.

Concurrent transitions

Definition (concurrent transitions)

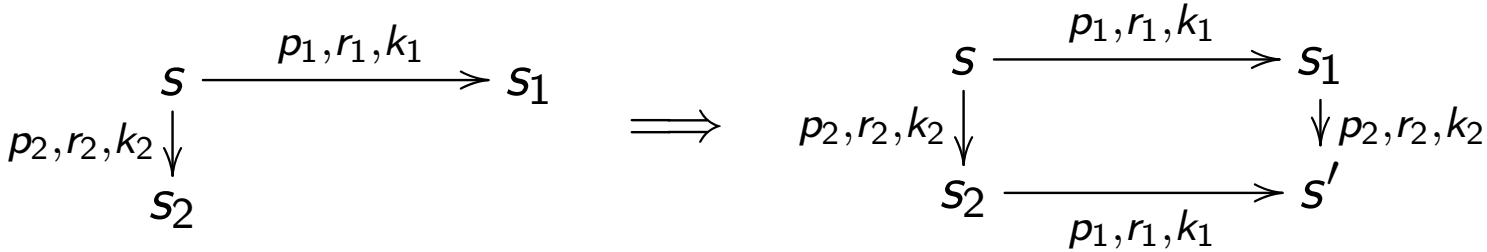
$t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$ are *in conflict* if:

- **both transitions are forward**, $p_1 = p_2$, and either $r_1 = r_2 = Sched$ or one is *Sched* and the other *Receive*
- one is **forward**, $p_1 = \mathrm{p}$, and the other one is a **backward** transition that undoes the creation of p
- one is a **forward** transition that delivers a message $\{v, \lambda\}$ and the other one is a **backward** transition that undoes the sending $\{v, \lambda\}$
- one is a **forward** transition and the other one is a **backward** transition such that $p_1 = p_2$ and either i) both applied rules are different from *Sched* and $\overline{Sched}$; ii) one rule is *Sched* and the other one is $\overline{Sched}$; iii) one rule is *Sched* and the other one is $\overline{Receive}$; or iv) one rule is $\overline{Sched}$ and the other one is *Receive*

Results

Lemma (square lemma)

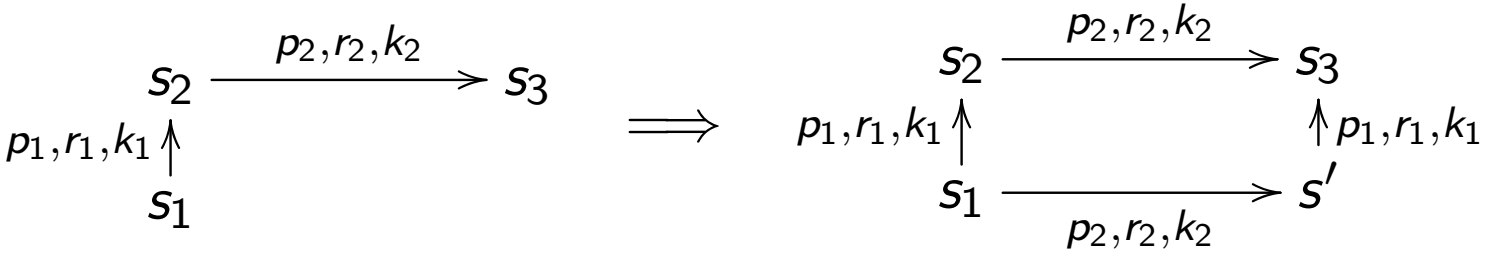
Let $\mathcal{P}$ be a program. Given two coinitial transitions $t_1 = (s \leftrightharpoons_{p_1,r_1,k_1} s_1)$ and $t_2 = (s \leftrightharpoons_{p_2,r_2,k_2} s_2)$, there exists two cofinal transitions $t_2/t_1 = (s_1 \leftrightharpoons_{p_2,r_2,k_2} s')$ and $t_1/t_2 = (s_2 \leftrightharpoons_{p_1,r_1,k_1} s')$.



Results

Lemma (switching lemma)

Given two composable transitions of the form $t_1 = (s_1 \leftrightharpoons_{p_1,r_1,k_1} s_2)$ *and* $t_2 = (s_2 \leftrightharpoons_{p_2,r_2,k_2} s_3)$ *such that* t_1^{-1} *and* t_2 *are concurrent, there exist a system* s_4 *and two composable transitions* $t_1' = (s_1 \leftrightharpoons_{p_2,r_2,k_2} s_4)$ *and* $t_2' = (s_4 \leftrightharpoons_{p_1,r_1,k_1} s_3)$.



Causal consistency

Definition (causal equivalence)

$$t_1; t_2/t_1 \approx t_2; t_1/t_2$$

$$t; t^{-1} \approx \epsilon_{init(t)}$$

Theorem (causal consistency)

Let d_1 and d_2 be coinitial derivations. Then, $d_1 \approx d_2$ iff d_1 and d_2 are cofinal.

Controlled semantics

Finally, we add control to this semantics (rollback operator):

$$\lfloor proc \rfloor_{\Psi}$$

There are **three kinds** of rollbacks:

- introduced by the programmer: $\#^{\mathtt{t}}_{\mathsf{ch}}$

$$\text{let } \_ = \text{check}(\mathtt{t}) \text{ in } expr$$

- internal: $\#_{\mathsf{sch}}$ (message dispatching) and $\#_{\mathsf{sp}}$ (spawning a process)

Rollback example

Undo actions from a process that has spawned another process

Rollback example (1)

Start rollback

To start a rollback, we fire the rule *Undo*

$$(\overline{Undo}) \quad \Gamma; \lfloor \langle h, p, (\theta', e'), q \rangle \rfloor_{\Psi} \,\&\, \Pi \leftharpoondown \Gamma; \lfloor \langle h, p, (\theta, e), q \rangle \rfloor_{\Psi \cup \{\#^{t}_{ch}\}} \,\&\, \Pi$$

Rollback example (2)

Rollback propagation

Propagate the rollback to the spawned process

$(\overline{Spawn2})$

$$\Gamma; \lfloor \langle \mathsf{spawn}(\theta, e, p'') : h, p, (\theta, e), q \rangle \rfloor_{\Psi} \;\&\; \lfloor \langle h', p'', (\theta'', e''), q'' \rangle \rfloor_{\Psi'} \;\&\; \Pi$$
$$\leftharpoondown \Gamma; \lfloor \langle \mathsf{spawn}(\theta, e, p'') : h, p, (\theta, e), q \rangle \rfloor_{\Psi} \;\&\; \lfloor \langle h', p'', (\theta'', e''), q'' \rangle \rfloor_{\Psi' \cup \{\#_{\mathsf{sp}}\}} \;\&\; \Pi$$
$$\text{if } h' \neq [\,] \text{ and } \#_{\mathsf{sp}} \notin \Psi$$

Rollback example (3)

Causal consistent undo

Undo action when all dependent actions have been undone

$(\overline{Spawn1})$

$$\Gamma; \lfloor \langle \mathsf{spawn}(\theta, e, p'') : h, p, (\theta, e), q \rangle \rfloor_{\Psi} \;\&\; \lfloor \langle [\,], p'', (\theta'', e''), q'' \rangle \rfloor_{\Psi'} \;\&\; \Pi \leftharpoondown \Gamma; \lfloor \langle h, p, (\theta, e), q \rangle \rfloor_{\Psi} \;\&\; \Pi$$

Rollback example (4)

Rollback removal

The rollback is removed when we reach the corresponding checkpoint

$$(\overline{Check}) \quad \Gamma; \lfloor \langle \mathsf{check}(\theta, e, \mathsf{t}) : h, p, (\theta', e'), q \rangle \rfloor_{\Psi} \;\&\; \Pi \;\leftharpoondown\; \Gamma; \lfloor \langle h, p, (\theta, e), q \rangle \rfloor_{\Psi \setminus \{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \;\&\; \Pi$$

Conclusions

Conclusions

We have:

- proposed a new semantics for Erlang
- defined a **reversible semantics for a subset of Erlang**
- proofs for the main properties
- developed a rollback operator for the reversible semantics

Ongoing work:

- implementation of the reversible semantics
- design concrete applications (debugging, fault-tolerance, etc.)

Thanks for your attention!