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)

September 2, 2017
Toruń, Poland

Motivation

We want to consider reversibility for **Erlang**

Many reasons to consider this language:

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

Motivation

We want to consider reversibility for **Erlang**

Many reasons to consider this language:

- 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
- hot code loading

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

Erlang syntax

We consider a subset of Erlang (basically, Core 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

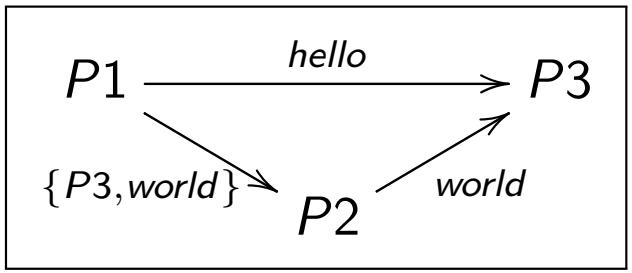
We consider a subset of Erlang (basically, Core Erlang) with this syntax:

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

Hello, World!

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

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

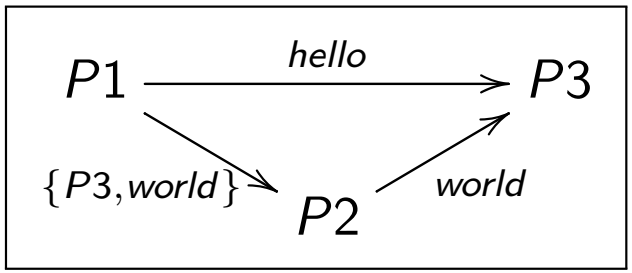


$$
\begin{aligned}
echo/0 = \text{fun}\ () \rightarrow\ & \text{receive} \\
& \quad \{P, M\} \rightarrow P\ !\ M \\
& \text{end}
\end{aligned}
$$

Hello, World!

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

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

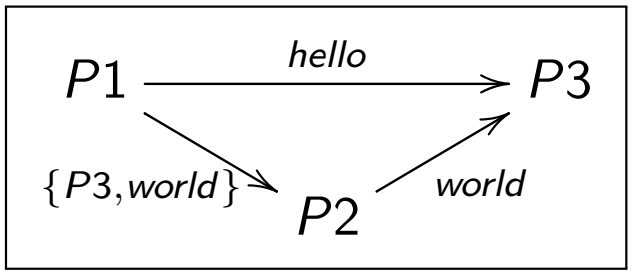


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \rightarrow \text{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \rightarrow P\,!\,M \\
\qquad\qquad\quad \text{end}
\end{array}
$$

Hello, World!

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

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

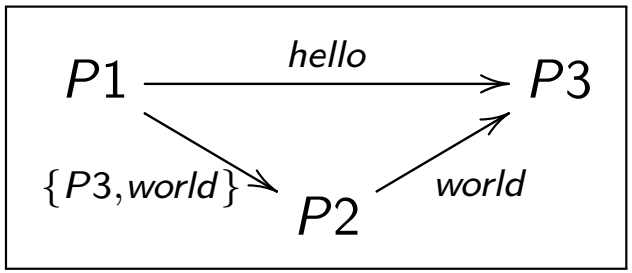


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

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

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

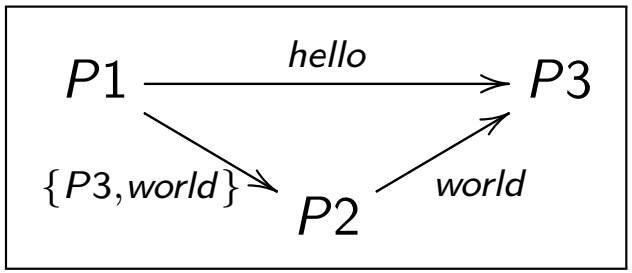


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\quad \{P, M\} \to P\ !\ M \\
\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

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

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

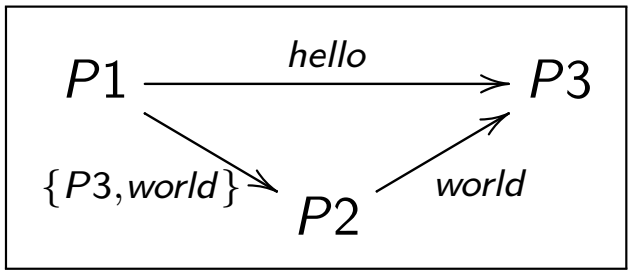


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

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

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

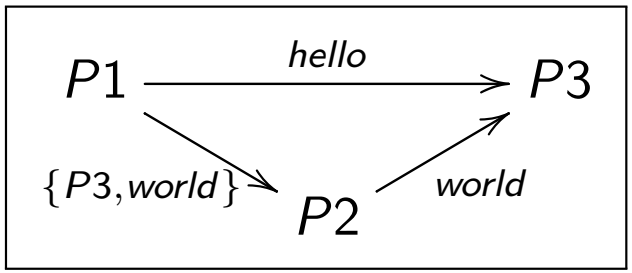


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

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

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

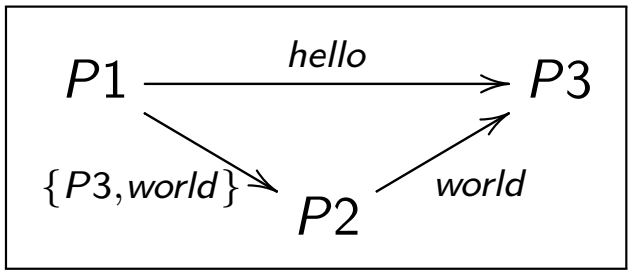


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \to P\ !\ M \\
\qquad\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

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

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

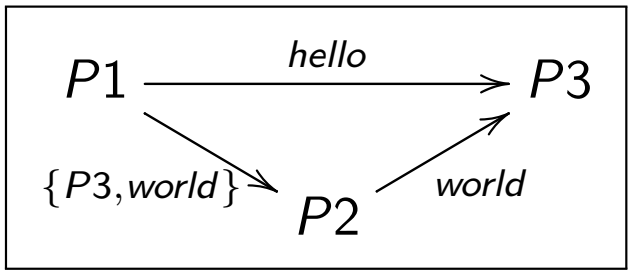


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\quad \{P, M\} \to P\ !\ M \\
\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

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

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

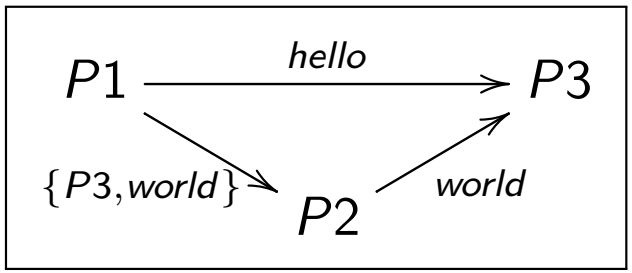


$$
\begin{array}{l}
\mathit{echo}/0 = \text{fun}\ () \to \text{receive} \\
\qquad\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad\qquad \text{end}
\end{array}
$$

Hello, World!

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

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



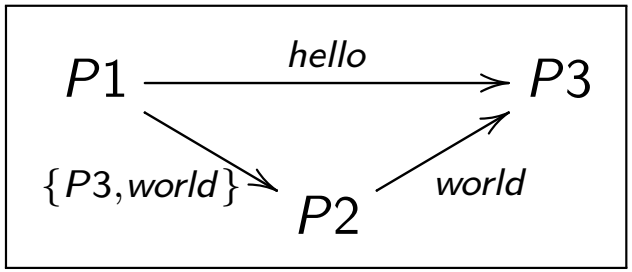
$$
\begin{array}{l}
\mathit{echo}/0 = \mathsf{fun}\ () \to \mathsf{receive} \\
\qquad\qquad\quad \{P, M\} \to P\,!\,M \\
\qquad\qquad\quad \mathsf{end}
\end{array}
$$

{hello,world}

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},{world,hello}

Semantics

Erlang semantics

There is no commonly accepted formal semantics for Erlang, the ones available

- are not modular [CMR+13]
- do not include evaluation of expressions [SFB10]
- ...

Our semantics

We proposed a semantics that

- has two levels: expression and system
- includes the evaluation of sequential expressions
- correctly models the semantics of the concurrent actions (it includes a global mailbox)

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 $\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi$

Semantics

Expression semantics

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: sequential actions

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

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

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

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

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

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

Expression semantics: rule Send

$$(Send1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \mathbin{!} e_2 \xrightarrow{\ell} \theta', e_1' \mathbin{!} e_2} \qquad (Send2)\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \mathbin{!} e_2 \xrightarrow{\ell} \theta', v_1 \mathbin{!} e_2'}$$

$$(Send3)\ \frac{}{\theta, v_1 \mathbin{!} v_2 \xrightarrow{\mathsf{send}(v_1, v_2)} \theta, v_2}$$

Expression semantics: rule Self

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

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

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

System semantics: rule Self

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

System semantics: rule Send

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

System semantics: rule Receive

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

System semantics: rule Spawn

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

System semantics: rule Sched

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

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

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

$$(Self)\qquad \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi}$$

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

Forward (reversible) semantics (2)

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

$$(\mathit{Receive})\quad \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\overline{cl_n}, q) = (\theta_i, e_i, \{v, \lambda\})}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi \rightharpoonup \Gamma; \langle \quad p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, \\ (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \setminus\setminus \{v, \lambda\} \quad \rangle \mid \Pi}$$

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

Backward semantics

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

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

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

Backward semantics (2)

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

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

$(\overline{Sched})$ $\quad \Gamma; \langle p, h, (\theta, e), \{v, \lambda\} : q\rangle \mid \Pi \leftharpoonup \Gamma \cup (p, \{v, \lambda\}); \langle p, h, (\theta, e), q\rangle \mid \Pi$
if the topmost rec(...) item in h (if any) has the form $\mathsf{rec}(\theta', e', \{v', \lambda'\}, q')$ with $q' \setminus\setminus \{v', \lambda'\} \neq \{v, \lambda\} : q$

Implementation

rev-erlang: A reversible Erlang simulator

rev-erlang

Proof of Concept implementation of the uncontrolled reversible semantics

In summary, rev-erlang

- is bundled together with a wxWidgets GUI
- has been developed in a modular way
- requires Erlang/OTP to be installed

rev-erlang is publicly available at
https://github.com/mistupv/rev-erlang

rev-erlang: How to use

Start an evaluation

1. Open an Erlang file (translation to Core Erlang)
2. Choose the function to be evaluated and write its arguments
3. Push the START button

Control evaluation

Two different modes for controlling an evaluation:

- **Manual**: Type pid, then choose any of the fireable rules
- **Automatic**:
  - **Steps (forward/backward)**:
    Introduce a number N in the Steps text box, then choose a direction
  - **Normalize**:
    Push the Normalize button

rev-erlang: How to use

Start an evaluation

1. Open an Erlang file (translation to Core Erlang)
2. Choose the function to be evaluated and write its arguments
3. Push the START button

Control evaluation

Two different modes for controlling an evaluation:

- **Manual**: Type pid, then choose any of the fireable rules
- **Automatic**:
  - **Steps (forward/backward)**:
    Introduce a number N in the Steps text box, then choose a direction
  - **Normalize**:
    Push the Normalize button

Sequential example: factorial

```
-module(factorial).
-export([fact/1]).

fact(0) -> 1;
fact(N) -> N * fact(N-1).
```

Concurrent example: hello_world

```
-module(hello_world).
-export([main/0]).

main() ->
  P2 = spawn(?MODULE, fun echo/0, []),
  P3 = spawn(?MODULE, fun target/0, []),
  P3 ! hello,
  P2 ! {P3, world}.

target() ->
  receive
    A ->
      receive
        B -> {A, B}
      end
  end.

echo() ->
  receive
    {P, M} -> P ! M
  end.
```

Conclusions

Conclusions

We have:

- proposed a new semantics for a subset of **Erlang**
- defined a **reversible semantics** for a subset of **Erlang**
- provided proofs (loop lemma, causal consistency, ...)
- developed a PoC implementation of this semantics

Ongoing work (design concrete applications):

- perform reversible debugging
- automate fault-tolerance techniques
- etc.

Thanks for your attention!