Erlang semantics in Coq

Peter Bereczky, Dániel Horpácsi, Simon Thompson

ELTE/Kent

December 2023

Overview

- Making refactorings trustworthy
- Focus on Core Erlang
- Sequential, process-local and inter-process semantics
- Results
- Collaboration?
- Arxiv paper: https://arxiv.org/abs/2311.10482

Making refactorings trustworthy

Refactoring tools modify source code: why should we trust them?

- What they do
- How they are built

Refactoring instances and refactorings

A *refactoring* is something like renaming, implemented to apply to all name changes in all code bases.

A *refactoring instance* is a particular case, e.g. renaming `print` to `show` in this project, github commit etc.

Testing

Use testing and property-based testing.

- Compare *before* and *after* by regression testing.
- Compare *before* and *after* on randomly generated inputs.
- Compare after_1 and after_2 given two refactoring tools.

Verification

Will need to be built on a fully formalised semantics of the object language, and definition(s) of equivalence.

- Proof of refactorings (cf Nik Sultana).
- Proof of refactoring instances . . .
- . . . where there is automation potential: SMT, tactics, etc.

Formalisation will support not only this project, but any work that requires meta-linguistic proof.

Semantic layers

Sequential semantics

Frame stack based, with unlabelled reduction relation

Semantic layers

Process-local semantics
Evaluation relation labelled by actions

Sequential semantics
Frame stack based, with unlabelled reduction relation

Semantic layers

Inter-process semantics

Labelled evaluation over a set of processes and an ether

Process-local semantics

Evaluation relation labelled by actions

Sequential semantics

Frame stack based, with unlabelled reduction relation

Syntax

$$v \in \mathit{Val} ::= i \mid a \mid \iota \mid \texttt{[]} \mid \texttt{[}v_1\,\texttt{|}\,v_2\texttt{]} \mid \texttt{fun}\ f/k(x_1, \ldots, x_k) \to e$$

$$p \in \mathit{Pat} ::= i \mid a \mid \iota \mid \texttt{[]} \mid \texttt{[}p_1\,\texttt{|}\,p_2\texttt{]} \mid x$$

$$\begin{aligned} e \in \mathit{Exp} ::= {} & v \mid x \mid f/k \mid \texttt{apply}\ e(e_1, \ldots, e_k) \\ \mid {} & \texttt{case}\ e\ \texttt{of}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\ \mid {} & \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \\ \mid {} & \texttt{[}e_1\,\texttt{|}\,e_2\texttt{]} \\ \mid {} & \texttt{letrec}\ f/k(x_1, \ldots, x_k) \to e_0\ \texttt{in}\ e_1 \\ \mid {} & \texttt{call}\ e(e_1, \ldots, e_k) \\ \mid {} & \texttt{receive}\ p_1 \to e_1; \ldots p_k \to e_k\ \texttt{end} \end{aligned}$$

Sequential semantics

The frame stack formalises the *continuation* of the computation.

$$\begin{aligned} F \in \mathit{Frame} ::= {} & \texttt{call}\ \square(e_1, \dots, e_k) \\ & \mid \texttt{call}\ v(\square, \dots, e_k) \mid \cdots \mid \texttt{call}\ v(v_1, \dots, \square) \\ & \mid \texttt{apply}\ \square(e_1, \dots, e_k) \\ & \mid \texttt{apply}\ v(\square, \dots, e_k) \mid \cdots \mid \texttt{apply}\ v(v_1, \dots, \square) \\ & \mid \texttt{let}\ x = \square\ \texttt{in}\ e_2 \\ & \mid \texttt{case}\ \square\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \\ & \mid \texttt{[}e_1 \,\texttt{|}\, \square\texttt{]} \mid \texttt{[}\square \,\texttt{|}\, v_2\texttt{]} \end{aligned}$$

$$K \in \mathit{FrameStack} ::= \mathcal{I}d \mid F :: K$$

Sequential semantic rules

1. Extract the first redex from language constructs, and put the remainder with a hole into the frame stack.
2. Modify the top frame of the stack by putting the calculated value into the hole, and then obtain the next reducible expression from the same frame.
3. Remove the top element of the stack, when the sub-expression has been completely reduced.

Sequential semantic rules: the rules for apply

$$\langle K, \texttt{apply}\ e(e_1, \ldots, e_k)\rangle \longrightarrow \langle \texttt{apply}\ \square(e_1, \ldots, e_k) :: K, e\rangle$$

$$\langle \texttt{apply}\ v(v_1, \ldots, v_{i-1}, \square, e_{i+1}, \ldots, e_k) :: K, v_i\rangle \longrightarrow$$
$$\langle \texttt{apply}\ v(v_1, \ldots, v_{i-1}, v_i, \square, e_{i+2}, \ldots, e_k) :: K, e_{i+1}\rangle \qquad (\text{if } i < k)$$

$$\langle \texttt{apply}\ \square() :: K, \texttt{fun}\ f/0() \to e\rangle \longrightarrow \langle K, e[f/0 \mapsto \texttt{fun}\ f/0() \to e]\rangle$$

$$\langle \texttt{apply}\ (\texttt{fun}\ f/k(x_1, \ldots, x_k) \to e)(v_1, \ldots, \square) :: K, v_k\rangle \longrightarrow$$
$$\langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1, \ldots, x_k) \to e, x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]\rangle$$

Sequential semantic rules: extract the redex

$$\langle K, \texttt{let}\ x = e_1\ \texttt{in}\ e_2\rangle \longrightarrow \langle \texttt{let}\ x = \square\ \texttt{in}\ e_2 :: K, e_1\rangle$$

$$\langle K, \texttt{[}e_1 \,|\, e_2\texttt{]}\rangle \longrightarrow \langle \texttt{[}e_1 \,|\, \square\texttt{]} :: K, e_2\rangle$$

$$\langle K, \texttt{apply}\ e(e_1, \ldots, e_k)\rangle \longrightarrow \langle \texttt{apply}\ \square(e_1, \ldots, e_k) :: K, e\rangle$$

$$\langle K, \texttt{call}\ e(e_1, \ldots, e_k)\rangle \longrightarrow \langle \texttt{call}\ \square(e_1, \ldots, e_k) :: K, e\rangle$$

$$\langle K, \texttt{letrec}\ f/k(x_1, \ldots, x_k) \to e_0\ \texttt{in}\ e\rangle \longrightarrow$$
$$\langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1, \ldots, x_k) \to e_0]\rangle$$

$$\langle K, \texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle \longrightarrow$$
$$\langle \texttt{case}\ \square\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, e_1\rangle$$

Sequential semantic rules: substitute value, get next redex

$$\langle \texttt{apply}\ \square(e_1, \ldots, e_k) :: K, v\rangle \longrightarrow \langle \texttt{apply}\ v(\square, \ldots, e_k) :: K, e_1\rangle$$

$$\langle \texttt{call}\ \square(e_1, \ldots, e_k) :: K, v\rangle \longrightarrow \langle \texttt{call}\ v(\square, \ldots, e_k) :: K, e_1\rangle$$

$$\langle \texttt{apply}\ v(v_1, \ldots, v_{i-1}, \square, e_{i+1}, \ldots, e_k) :: K, v_i\rangle \longrightarrow$$
$$\langle \texttt{apply}\ v(v_1, \ldots, v_{i-1}, v_i, \square, e_{i+2}, \ldots, e_k) :: K, e_{i+1}\rangle \qquad (\text{if } i < k)$$

$$\langle \texttt{call}\ v(v_1, \ldots, v_{i-1}, \square, e_{i+1}, \ldots, e_k) :: K, v_i\rangle \longrightarrow$$
$$\langle v(v_1, \ldots, v_{i-1}, v_i, \square, e_{i+2}, \ldots, e_k) :: K, e_{i+1}\rangle \qquad (\text{if } i < k)$$

$$\langle \texttt{[}e_1 \,|\, \square\texttt{]} :: K, v_2\rangle \longrightarrow \langle \texttt{[}\square \,|\, v_2\texttt{]} :: K, e_1\rangle$$

Sequential semantic rules: remove the top of the stack

$$\langle \texttt{apply } \Box() :: K, \texttt{fun } f/0() \to e \rangle \longrightarrow \langle K, e[f/0 \mapsto \texttt{fun } f/0() \to e] \rangle$$

$$\langle \texttt{apply } (\texttt{fun } f/k(x_1, \ldots, x_k) \to e)(v_1, \ldots, \Box) :: K, v_k \rangle \longrightarrow$$
$$\quad \langle K, e[f/k \mapsto \texttt{fun } f/k(x_1, \ldots, x_k) \to e, x_1 \mapsto v_1, \ldots, x_k \mapsto v_k] \rangle$$

$$\langle \texttt{call '+'}(i_1, \Box) :: K, i_2 \rangle \longrightarrow \langle K, i_1 + i_2 \rangle$$

$$\langle \texttt{let } x = \Box \texttt{ in } e_2 :: K, v \rangle \longrightarrow \langle K, e_2[x \mapsto v] \rangle$$

$$\langle \texttt{[}\Box \,|\, v_2\texttt{]} :: K, v_1 \rangle \longrightarrow \langle K, \texttt{[}v_1 \,|\, v_2\texttt{]} \rangle$$

$$\langle \texttt{case } \Box \texttt{ of } p \texttt{ then } e_2 \texttt{ else } e_3 :: K, v \rangle \longrightarrow$$
$$\quad \langle K, e_2[match(p, v)] \rangle \quad (\text{if } is\_match(p, v))$$

$$\langle \texttt{case } \Box \texttt{ of } p \texttt{ then } e_2 \texttt{ else } e_3 :: K, v \rangle \longrightarrow$$
$$\quad \langle K, e_3 \rangle \qquad\qquad (\text{if } \neg is\_match(p, v))$$

Process-local semantics

This covers both message passing and (exit) signals.

Rules are labelled with *actions*.

Rules work over $(K, e, q, pl, flag)$

- K denotes a frame stack
- e is an expression
- q is the mailbox (represented as a list of values)
- pl is the set of linked processes
- $flag$ is the status of the `'trap_exit'` flag

Signals and actions

$$s \in \mathit{Signal} \ ::= \mathit{msg}(v) \mid \mathit{exit}(v, b) \mid \mathit{link} \mid \mathit{unlink}$$

$$\begin{aligned} a \in \mathit{Action} \ ::= \ & \mathit{send}(\iota_1, \iota_2, s) \mid \mathit{rec}(v) \mid \mathit{self}(\iota) \mid \mathit{arr}(\iota_1, \iota_2, s) \\ & \mid \mathit{spawn}(\iota, e_1, e_2) \mid \tau \mid \Downarrow \mid \mathit{flag} \end{aligned}$$

Actions contain information about e.g. source and destination information; signals are simply values of various kinds.

- *send* from process to ether
- *arr*ive at mailbox from ether
- *rec*eive from mailbox within process.
- τ denotes sequential evaluation.

Process-local rules

We concentrate on the rules for message passing, but the rules for termination, `exit`, (un)link etc. follow similar lines. These rules proliferate, depending on the nature of the exit, and whether or not the recipient process is trapping exits.

$$\frac{\langle K, e\rangle \rightarrow \langle K', e'\rangle}{(K, e, q, pl, b) \xrightarrow{\tau} (K', e', q, pl, b)} \qquad \text{(Seq)}$$

$$(K, e, q, pl, b) \xrightarrow{arr(\iota_1,\iota_2,msg(v))} (K, e, q +\!\!+ [v], pl, b) \qquad \text{(Msg)}$$

Process-local rules

$$(\texttt{call '!'}(\iota_2, \Box) :: K, v, q, pl, b) \xrightarrow{send(\iota_1, \iota_2, msg(v))} (K, v, q, pl, b) \quad \text{(Send)}$$

$$(\texttt{call } \Box() :: K, \texttt{'self'}, q, pl, b) \xrightarrow{self(\iota)} (K, \iota, q, pl, b) \quad \text{(Self)}$$

$$\frac{f = \texttt{fun } f/k(x_1, \ldots, x_k) \to e}{(\texttt{call 'spawn'}(f, \Box) :: K, vs, q, pl, b) \xrightarrow{spawn(\iota, f, vs)} (K, \iota, q, pl, b)} \quad \text{(Spawn)}$$

Process-local rules

$$\frac{\begin{array}{c} l = match(p_i, v) \\ is\_match(p_i, v) \\ q = [v_1, \ldots, v_n, v, \ldots] \end{array} \quad \begin{array}{c} \forall j < i : \neg is\_match(p_j, v) \\ (\forall m, j : 1 \leq m \leq k \wedge 1 \leq j \leq n \implies \neg is\_match(p_m, v_j)) \end{array}}{(K, \texttt{receive}\ p_1 \to e_1; \ldots; p_k \to e_k\ \texttt{end}, q, pl, b) \xrightarrow{rec(v)} (K, e_i[l], rem_1(v, q), pl, b)} \text{(Receive)}$$

Inter-process semantics

A *node* is a pair ($(\Delta, \Pi) \in \mathit{Node}$) of an ether and a process pool.

- An ether (denoted by Δ) is a mapping of source and target identifier pairs to lists of signals.
- The process pool (denoted by Π) is a mapping that associates process identifiers with processes.

Inter-process semantics

$$\frac{p \xrightarrow{send(\iota_1,\iota_2,s)} p'}{(\Delta, \iota_1 : p \parallel \Pi) \xrightarrow{\iota_1 : send(\iota_1,\iota_2,s)} (\Delta[(\iota_1,\iota_2) \overset{+}{\mapsto} s], \iota_1 : p' \parallel \Pi)} \quad \text{(NSend)}$$

$$\frac{p \xrightarrow{arr(\iota_1,\iota_2,s)} p' \qquad remFirst(\Delta, \iota_1, \iota_2) = Some\ (s, \Delta')}{(\Delta, \iota_1 : p \parallel \Pi) \xrightarrow{\iota_1 : arr(\iota_1,\iota_2,s)} (\Delta', \iota_1 : p' \parallel \Pi)} \quad \text{(NArrive)}$$

$$(\Delta, \iota : [] \parallel \Pi) \xrightarrow{\iota : \Downarrow} (\Delta, \Pi \setminus \iota) \quad \text{(NTerm)}$$

Inter-process semantics

$$\frac{\iota_2 \notin (\iota_1 : p \parallel \Pi) \quad p \xrightarrow{spawn(\iota_2, v, vs)} p' \qquad v = \texttt{fun}\ f/k(x_1, \ldots, x_k) \to e \quad convert\_list(vs) = Some\ [v_1, \ldots, v_k]}{(\Delta, \iota_1 : p \parallel \Pi) \xrightarrow{\iota_1 : spawn(\iota_2, v, vs)} (\Delta, \iota_2 : ([], \texttt{apply}\ v(v_1, \ldots, v_k), [], [], f\!f) \parallel \iota_1 : p' \parallel \Pi)} \text{(NSpawn)}$$

$$\frac{p \xrightarrow{a} p' \qquad a \in \{self(\iota), \Downarrow, \tau, flag\} \cup \{rec(v) \mid v \in Value\}}{(\Delta, \iota : p \parallel \Pi) \xrightarrow{\iota : a} (\Delta, \iota : p' \parallel \Pi)} \text{(NOther)}$$

Alternatives and extensions

Sequential semantics: big step and natural semantics.

From Core Erlang to Erlang:

- Sequential semantics: exceptions, side-effects.
- Module system
- Distributed Erlang

Validation

Evaluate examples “by hand”.

- Investigate automated comparison.

Proofs of desirable meta-theoretical properties.

Metatheory

Sequential and process-local evaluation is deterministic

Confluence of sequential reductions in the same process

Theorem (Signal ordering guarantee)

For all nodes $\Sigma_1, \Sigma_2, \Sigma_3$*, process identifiers* ι, ι'*, and unique signals* $s_1 \neq s_2$*, if* $\Sigma_1 \xrightarrow{\iota:send(\iota,\iota',s_1)} \Sigma_2$ *and* $\Sigma_2 \xrightarrow{\iota:send(\iota,\iota',s_2)} \Sigma_3$*, then for all nodes* Σ_4 *and action traces* l *which satisfy* $\Sigma_3 \xrightarrow{l}^{*} \Sigma_4$ *and also* $(\iota', arr(\iota, \iota', s_1)) \notin l$ *there is no node* Σ_5 *at which* s_2 *can arrive:* $\Sigma_4 \xrightarrow{\iota':arr(\iota,\iota',s_2)} \Sigma_5$.

Metatheory

Theorem (Confluence of sequential reductions)

For all nodes $\Sigma_1, \Sigma_2, \Sigma_2', \Sigma_3$*, process identifier* ι*, and action* a*, if* $\Sigma_1 \longrightarrow^* \Sigma_2$*, and a reduction can be done in the starting and in the final configuration too:* $\Sigma_1 \xrightarrow{\iota:a} \Sigma_2'$*, and* $\Sigma_2 \xrightarrow{\iota:a} \Sigma_3$*, then* $\Sigma_2' \longrightarrow^* \Sigma_3$*.*

Program Equivalence

A relation R is a weak bisimulation iff

- For all nodes $\Sigma_1, \Sigma_2, \Sigma_1'$, process identifiers ι, and actions $a \neq \tau$, if $(\Sigma_1, \Sigma_2) \in R$ and $\Sigma_1 \xrightarrow{\iota:a} \Sigma_1'$, then there are nodes $\Sigma_2^1, \Sigma_2^2, \Sigma_2'$, which are reducible from Σ_2 in the following way: $\Sigma_2 \longrightarrow^* \Sigma_2^1$, $\Sigma_2^1 \xrightarrow{\iota:a} \Sigma_2^2$, and $\Sigma_2^2 \longrightarrow^* \Sigma_2'$, and $(\Sigma_1', \Sigma_2') \in R$.
- For all nodes $\Sigma_1, \Sigma_2, \Sigma_1'$, process identifiers ι, and actions $a \neq \tau$, if $(\Sigma_1, \Sigma_2) \in R$ and $\Sigma_2 \xrightarrow{\iota:a} \Sigma_2'$, then there are nodes $\Sigma_1^1, \Sigma_1^2, \Sigma_1'$, which are reducible from Σ_1 in the following way: $\Sigma_1 \longrightarrow^* \Sigma_1^1$, $\Sigma_1^1 \xrightarrow{\iota:a} \Sigma_1^2$, and $\Sigma_1^2 \longrightarrow^* \Sigma_1'$, and $(\Sigma_1', \Sigma_2') \in R$.

Theorem

$\longrightarrow^*$ *(between nodes) is a weak bisimulation.*

Program Equivalence

Next steps: look at other equivalence notions

- barbed bisimulation
- index nodes with active Pids
- ...