Acta Cybernetica — *online–first paper version* — pages 1–32.

A Formalisation of Core Erlang, a Concurrent Actor Language*

Péter Bereczky[ab], Dániel Horpácsi[ac], and Simon Thompson[ade]

Abstract

In order to reason about the behaviour of programs described in a programming language, a mathematically rigorous definition of that language is needed. In this paper, we present a machine-checked formalisation of concurrent Core Erlang (a subset of Erlang) based on our previous formalisations of its sequential sublanguage. We define a modular, frame stack semantics, show how program evaluation is carried out with it, and prove a number of properties (e.g. determinism, confluence). Finally, we define program equivalence based on bisimulations and prove that side-effect-free evaluation is a bisimulation. This research is part of a wider project that aims to verify refactorings to prove that particular program code transformations preserve program behaviour.

Keywords: formal semantics, formal verification, concurrency, actor model, program equivalence, bisimulation, Erlang, Core Erlang, Coq

1 Introduction

Our work here contributes to a wider project [17] to reason about the correctness of refactorings for functional languages in general, and for Erlang [7] in particular. In our terminology, refactoring is a code transformation that preserves the observable behaviour of programs. Our understanding of the state-of-the-art refactoring tools scene suggests that behaviour preservation (i.e. correctness) is subject to extensive testing, but formal verification is not yet used in practice. We aim to change this, at least in the case of Erlang, and develop higher assurance for refactorings by

*Supported by the ÚNKP-21-3 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund. Supported by "Application Domain Specific Highly Reliable IT Solutions" financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme by the National Research, Development and Innovation Fund of Hungary.

[a]Eötvös Loránd University, Budapest, Hungary
[b]E-mail: berpeti@inf.elte.hu, ORCID: 0000-0003-3183-0712
[c]E-mail: daniel-h@elte.hu, ORCID: 0000-0003-0261-0091
[d]University of Kent, United Kingdom
[e]E-mail: S.J.Thompson@kent.ac.uk, ORCID: 0000-0002-2350-301X

DOI: 10.14232/actacyb.298977

developing formal, machine-checked theories for program semantics, equivalence and program transformation.

Erlang is a dynamically-typed, impure, functional programming language, which excels at concurrency. Core Erlang [6] is a standard subset of Erlang that contains all the essential elements of Erlang, so that a semantics of Core Erlang can be extended to a semantics for the full language in a straightforward way. In earlier work we defined and implemented formal semantics for the sequential parts of Erlang and Core Erlang, including a reduction semantics for a subset of Erlang using the $\mathbb{K}$ framework [20], and a natural semantics for a subset of Core Erlang, implemented in Coq [2, 3]. We have also implemented a functional big-step [28] semantics for this subset of Core Erlang, and shown [8] that this semantics is equivalent to the natural semantics. In turn, the semantics was validated [4] against the reference implementation of Erlang, namely the Erlang/OTP compiler [11].

Having these semantics defined, we focused on proving the equivalence of programs. On the one hand, we are interested in using the semantics to prove particular pairs of programs equivalent, and on the other, the correctness of many local refactoring steps can be reduced to the equivalence of simple expressions. When developing precise, standard definitions of equivalence, we decided to bring our results to smaller-step semantics and developed a frame stack semantics and equivalence definitions [32] built on that for sequential Core Erlang [19]. The frame stack style for semantics is beneficial for two reasons: it is well-suited to express various standard equivalence definitions [29], and furthermore, the semantics of concurrent expressions can be defined more easily in small-step approaches [25].

Our formalisations of (Core) Erlang are not the first ones. There are a number of other semantics for both sequential and concurrent subsets of (Core) Erlang on which our work has been based. The novelty of our work presented here lies in the fact that it remains more faithful to the language specification [6] and the reference manual [12] than the others; for instance, unlike other works, we formalised exit signals and the signal ordering guarantee closely following the specification. We give a more detailed comparison and an overview of the related research in Section 6. Also worth pointing out is that our formal development is accompanied by a machine-checked implementation [9].

We continue our formalisation efforts and in this paper we add concurrency to our frame stack semantics for Core Erlang. In particular, we create the definition in a modular way: the sequential and process-local parts of the semantics can be replaced by a more complete formalised part of Core Erlang or Erlang (or indeed another programming language) without the need to rewrite the whole semantics. The main contributions of this paper are the following:

- A modular, frame stack semantics for a concurrent subset of Core Erlang;
- Proofs about the properties of the concurrent semantics;
- Results on Core Erlang program equivalence verification using bisimulation.

The rest of the paper is structured as follows. In Section 2 we introduce (Core) Erlang and our previous work informally, and define the syntax of the formalised

sublanguage. In Section 3 we describe a modular, dynamic semantics of Core Erlang, focusing on the concurrent sublanguage, then in Section 4 we show the evaluation of simple concurrent programs and prove properties of the semantics. Section 5 defines the concepts of program equivalence and the corresponding results, and then we discuss related work in Section 6. Finally, Section 7 discusses future work and concludes.

2 Background

As mentioned before, Erlang is a dynamically-typed, impure functional programming language. The biggest strength of Erlang is that it really excels at concurrent computation, based on the actor model [1]. For this reason, Erlang was initially used in telecommunication and banking systems, but it now plays a role in high-availability, scalable web-based systems.

2.1 The Erlang Model of Concurrency

Erlang implements and extends the actor model [1]. An Erlang system contains lightweight processes (actors) that can spawn other processes to execute a particular task. Each process executes in its own space, and so they do not share memory. Processes can only communicate by asynchronous message passing. Each process has a message queue (mailbox), where incoming messages are stored in the order of their arrival. A process can select which messages to handle from its mailbox: messages do not need to be handled in the order in which they are received.

Besides messages, processes can also send and receive other signals [13], such as *link*, *unlink* and *exit*. These additional signals can trigger potential changes in the state of the process immediately upon their arrival without being placed into the mailbox. The *link* and *unlink* signals create and remove, respectively, a bi-directional link between two processes, which represents a mutual dependency, and affects the handling of *exit* signals. In general, *exit* signals are used to indicate and initiate termination; they include a reason (describing why they were sent), and a flag indicating whether they were sent through a link (we call this value the *link flag* of the *exit* signal). If one of a pair of linked processes terminates, it will send an *exit* signal to the other process via the link. Processes can terminate for a number of reasons: having finished evaluation, receiving a particular *exit* signal, or terminating abnormally (e.g. with an exception).

Processes have a flag called `'trap_exit'` which, when set, causes *exit* signals to be converted into messages (except in very particular circumstances), i.e. the process *traps exits*. Based on this flag, the reason of the exit signal, and whether the *exit* signal was sent through a link, there are three different outcomes (see [13] and Section 3.3): the receiver process 1) terminates, 2) drops the *exit* signal, or 3) converts the *exit* signal to a message and adds it at the end of its mailbox.

In the next section, we present the syntax of the language under formalisation, which is a sublanguage of Core Erlang. Note that Core Erlang is not merely a stan-

dard subset of Erlang, it is also used in the compilation process as an intermediate step, and numerous programming languages based on the BEAM platform can be compiled to Core Erlang [16]. Furthermore, as for concurrency, the two languages implement essentially the same model. For more details, we refer to the Erlang Programming book [7] and the reference manual [12].

2.2 Language Syntax

In this section we discuss and extend the formal syntax of the sequential sublanguage of Core Erlang as presented in our previous work [19]. For better readability, we use a syntax definition that abstracts over the concrete syntax of the language; however, any expressions written in this syntax can be simply transformed to Core Erlang.

Definition 1 (Language syntax)**.**

$$
\begin{aligned}
v \in \mathit{Val} \ &::= i \mid a \mid \iota \mid \texttt{[]} \mid \texttt{[}v_1\texttt{|}v_2\texttt{]} \mid \texttt{fun}\ f/k(x_1,\ldots,x_k) \to e \\
p \in \mathit{Pat} \ &::= i \mid a \mid \iota \mid \texttt{[]} \mid \texttt{[}p_1\texttt{|}p_2\texttt{]} \mid x \\
e \in \mathit{Exp} \ &::= v \mid x \mid f/k \mid \texttt{apply}\ e(e_1,\ldots,e_k) \mid \texttt{case}\ e\ \texttt{of}\ p\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \\
&\quad \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \texttt{[}e_1\texttt{|}e_2\texttt{]} \mid \texttt{letrec}\ f/k(x_1,\ldots,x_k) \to e_0\ \texttt{in}\ e_1 \\
&\quad \mid \texttt{call}\ e(e_1,\ldots,e_k) \mid \texttt{receive}\ p_1 \to e_1;\ldots p_k \to e_k\ \texttt{end}
\end{aligned}
$$

We use i, k, n to range over integers, a, f over atoms, x over variables, and ι over process identifiers. f/k denotes a function identifier, where f is the function name, and k is its arity. The primitive values of the language are integers (denoted by numbers), atoms (strings of characters, enclosed in single quotation marks), and process identifiers (for simplicity, also denoted by numbers). Besides these, lists and functions are also values, and patterns are built from variables, integers, atoms and process identifiers, and formed into composite patterns as lists[1].

Note that process identifiers are not patterns in Core Erlang, but with process identifiers as patterns, we can distinguish them from other values of the language; in Erlang, the `is_pid` function can be used instead. This distinction is needed to maintain the proof of coincidence of sequential equivalence definitions described in [19] (namely, the coincidence of behavioural and contextual equivalence).

For simplicity of formalisation, functions are always named to enable explicit recursive calls, but in this paper we omit function names for readability when there are no recursive calls in the body expression. For lists, we use the standard notations of Erlang, that is a list $\texttt{[}e_1\texttt{|[}e_2\texttt{|[}\ldots\texttt{|[}e_n\texttt{|[]]}\ldots\texttt{]]}$ will be denoted by $\texttt{[}e_1\texttt{,}e_2\texttt{,}\ldots\texttt{,}e_n\texttt{]}$. Note that we also include Erlang's improper lists (such as `[1|2]`), but these do not require specific care in the semantics rules.

Expressions of the sequential sublanguage are values, variables, function identifiers, binding expressions (both `let` and `letrec`), function applications (`apply`), pattern matching (`case`) expressions[2].

[1] Tuples are not included in this language, but would be handled similarly to parameter lists.

[2] This expression is a simplified version of Core Erlang's `case`, restricting it to only two branches.

We extend the syntax (as in [19]) with two language elements in this work, the first one is BIF (built-in function) call (denoted by `call` $e(e_1, \ldots, e_k)$), the second is the `receive` expression. BIFs are used to implement both sequential (e.g. addition of integers) and concurrent features of the language.

In particular, the concurrency model introduced in the previous subsection is implemented as follows:

- the `'!'` BIF is used to send messages;
- a `receive` expression is used to select a message from the process mailbox by means of pattern matching;
- processes are created with the `'spawn'` BIF (taking a function and its parameters as arguments for the new process to evaluate);
- *link*, *unlink* and *exit* signals can be sent with the identically named BIFs;
- the `'process_flag'` BIF is used to set the `'trap_exit'` flag.

The syntax we presented here is implemented in Coq using the nameless variable representation [10]. This way, we reuse existing approaches to define capture-avoiding, parallel substitutions [30]. Nonetheless, we use named variables in this paper for readability. Substitutions are denoted by $e[x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]$, which results in replacing $x_1, \ldots, x_k$ variables simultaneously with $v_1, \ldots, v_k$ values in the expression e. We omit further details about substitutions and static semantics since they are not in the scope of this paper. For further details we refer to our previous work [19] and to the formalisation [9]. Next, we show an example expression in the syntax presented above:

Example 1 (A simple map function in Core Erlang)**.** The following snippet shows a simple sequential Core Erlang function that transforms the elements of a list by applying the function `F` to each member. Since it is a rather simple definition, we present it in concrete syntax for better readability. To evaluate the function, it suffices to substitute the body of the `letrec` (denoted by `...`) with an application of `'mm'/2`.

```
letrec 'mm'/2 =
  fun(F, E) ->
    case E of [H|T]
      then [ apply F(H) | apply 'mm'/2(F, T) ]
      else []
    end
  in ...
```

△

The syntax of the sequential sublanguage is minimal, but representative.

3 Dynamic Semantics

In this section we explain the dynamic semantics of the formalised Core Erlang subset. We present a three-layered, modular semantics for the language such that the sequential parts of the semantics can be replaced by a more complete formalised part of Core Erlang, Erlang, or another programming language entirely.

Table 1: Layers of the semantics

Layer name	Notation	Description
Inter-process semantics (Section 3.4)	$\xrightarrow{\iota:a}$	System-level reductions
Process-local semantics (Section 3.3)	$\xrightarrow{a}$	Process-level reductions
Sequential semantics (Section 3.1)	$\longrightarrow$	Computational reductions

3.1 Sequential Semantics

First, we briefly present the sequential semantics [19] on which we base the concurrent formalisation. We highlight that the specification of Core Erlang [6] does not define the evaluation order of subexpressions, but the compiler employs a leftmost-innermost strategy [26]: during the standard translation of Erlang, the evaluation order is enforced by nested `let` expressions in Core Erlang. Furthermore, lists in Core Erlang are evaluated from the right[3]. The compiler's evaluation strategy *is* reflected in our definition.

The semantics has been formally defined as a frame stack semantics [29]. This definition style resembles reduction semantics [14], but the reduction context is deconstructed into a stack of evaluation frames with holes denoted by $\square$. The frame stack can be regarded as the continuation of the computation.

Definition 2 (Syntax of frames, frame stacks)**.**

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

For the stacks, we use the following notations: $\mathfrak{Id}$ denotes the empty stack and $F :: K$ denotes adding frame F to the top of stack K. Next, we introduce two metatheoretical functions for pattern matching:

[3]The reference implementation generates a bytecode sequence that evaluates the list tail before evaluating the list head.

$\langle K, \texttt{let}\ x = e_1\ \texttt{in}\ e_2\rangle \longrightarrow \langle \texttt{let}\ x = \Box\ \texttt{in}\ e_2 :: K, e_1\rangle$
$\langle K, \texttt{[}e_1\texttt{|}e_2\texttt{]}\rangle \longrightarrow \langle \texttt{[}e_1\texttt{|}\Box\texttt{]} :: K, e_2\rangle$
$\langle K, \texttt{apply}\ e(e_1, \ldots, e_k)\rangle \longrightarrow \langle \texttt{apply}\ \Box(e_1, \ldots, e_k) :: K, e\rangle$
$\langle K, \texttt{call}\ e(e_1, \ldots, e_k)\rangle \longrightarrow \langle \texttt{call}\ \Box(e_1, \ldots, e_k) :: K, e\rangle$
$\langle K, \texttt{letrec}\ f/k(x_1, \ldots, x_k) \to e_0\ \texttt{in}\ e\rangle \longrightarrow$
$\quad \langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1, \ldots, x_k) \to e_0]\rangle$
$\langle K, \texttt{case}\ e_1\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3\rangle \longrightarrow \langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, e_1\rangle$

$\langle \texttt{apply}\ \Box(e_1, \ldots, e_k) :: K, v\rangle \longrightarrow \langle \texttt{apply}\ v(\Box, \ldots, e_k) :: K, e_1\rangle$
$\langle \texttt{call}\ \Box(e_1, \ldots, e_k) :: K, v\rangle \longrightarrow \langle \texttt{call}\ v(\Box, \ldots, e_k) :: K, e_1\rangle$
$\langle \texttt{apply}\ v(v_1, \ldots, v_{i-1}, \Box, e_{i+1}, \ldots, e_k) :: K, v_i\rangle \longrightarrow$
$\quad \langle \texttt{apply}\ v(v_1, \ldots, v_{i-1}, v_i, \Box, e_{i+2}, \ldots, e_k) :: K, e_{i+1}\rangle \qquad (\text{if } i < k)$
$\langle \texttt{call}\ v(v_1, \ldots, v_{i-1}, \Box, e_{i+1}, \ldots, e_k) :: K, v_i\rangle \longrightarrow$
$\quad \langle v(v_1, \ldots, v_{i-1}, v_i, \Box, e_{i+2}, \ldots, e_k) :: K, e_{i+1}\rangle \qquad (\text{if } i < k)$
$\langle \texttt{[}e_1\texttt{|}\Box\texttt{]} :: K, v_2\rangle \longrightarrow \langle \texttt{[}\Box\texttt{|}v_2\texttt{]} :: K, e_1\rangle$

$\langle \texttt{apply}\ \Box() :: K, \texttt{fun}\ f/0() \to e\rangle \longrightarrow \langle K, e[f/0 \mapsto \texttt{fun}\ f/0() \to e]\rangle$
$\langle \texttt{apply}\ (\texttt{fun}\ f/k(x_1, \ldots, x_k) \to e)(v_1, \ldots, \Box) :: K, v_k\rangle \longrightarrow$
$\quad \langle K, e[f/k \mapsto \texttt{fun}\ f/k(x_1, \ldots, x_k) \to e, x_1 \mapsto v_1, \ldots, x_k \mapsto v_k]\rangle$
$\langle \texttt{call}\ \texttt{'+'}(i_1, \Box) :: K, i_2\rangle \longrightarrow \langle K, i_1 + i_2\rangle$
$\langle \texttt{let}\ x = \Box\ \texttt{in}\ e_2 :: K, v\rangle \longrightarrow \langle K, e_2[x \mapsto v]\rangle$
$\langle \texttt{[}\Box\texttt{|}v_2\texttt{]} :: K, v_1\rangle \longrightarrow \langle K, \texttt{[}v_1\texttt{|}v_2\texttt{]}\rangle$
$\langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, v\rangle \longrightarrow \langle K, e_2[\mathit{match}(p, v)]\rangle \quad (\text{if } \mathit{is\_match}(p, v))$
$\langle \texttt{case}\ \Box\ \texttt{of}\ p\ \texttt{then}\ e_2\ \texttt{else}\ e_3 :: K, v\rangle \longrightarrow \langle K, e_3\rangle \quad (\text{if } \neg \mathit{is\_match}(p, v))$

Figure 1: Sequential semantics of Core Erlang

- $\mathit{is\_match}(p, v)$: determines whether the value v matches the pattern p: that is they have been built up with the same constructs of Core Erlang up to pattern variables.

- $\mathit{match}(p, v)$: if the value v matches the pattern p, this function returns a substitution which contains the result of the pattern matching in form of a mapping from pattern variables to values.

We present the sequential semantics rules in Section 1. We use $\langle K, e\rangle \longrightarrow \langle K', e'\rangle$ to denote one reduction step between configurations consisting of a frame stack and an expression to be evaluated. Recall that v, v_i are used for values, i, i_j for integer values, and e, e_k for (unevaluated) expressions.

The biggest advantage of this semantics definition is that there are no premises in the reduction rules about the reduction of subexpressions since they have been put into the frame stack. Therefore the propagation of concurrent actions to this level is not necessary (i.e. there are no labels on the reduction rules). On the other hand, its disadvantage is that the complex syntax of frames is needed to be defined separately from the syntax of the language.

The reduction rules can be categorised into three groups:

- Rules that extract the first redex from language constructs, and put the remainder with a hole into the frame stack.
- Rules that modify the top frame of the stack by putting the calculated value into the hole, and obtaining the next reducible expression from the same frame.
- Rules that remove the top element of the frame stack, which also marks that the subexpression has been completely reduced.

The evaluation of any language element (except `letrec`) includes using exactly one rule once from the first and third categories. We note that this would change if exceptions and exception handler expressions were present in the sequential language. The connection between exceptions and signals is that when an exception terminates a process, it will emit an exit signal with the details of the exception. The presence of exceptions does not affect the modularity of the definition, but it would require consistent modifications in multiple layers.

Example 2 (Sequential evaluation of Section 1)**.** We use $\longrightarrow^*$ to denote the reflexive, transitive closure of the relation $\longrightarrow$. For simplicity, we denote the successor function `fun(X)` $\rightarrow$ `call '+'/2(X,1)` with f in the following example. We also use mm to denote the function bound inside the `letrec` expression in Section 1.

The first step is to evaluate the head of the application mm to itself (since it is a function). Next, the parameter function f is reduced to itself. Thereafter, the parameter list is reduced (`[0,1,2]`) by deconstructing it starting from the back, pushing the head elements of the sublists into the frame stack. Actually, the semantics just checks in this case that all of these elements are values, and then the list is reconstructed. These actions transform the type of the parameter list from `[`e_1`|`e_2`]` to `[`v_1`|`v_2`]`, this is the reason why they are necessary, although, there are

two seemingly identical configurations in the reduction sequence.

$$\begin{aligned}
&\langle \mathcal{I}d, \texttt{letrec 'mm'/2} = mm \texttt{ in apply 'mm'/2}(f, \texttt{[0,1,2]})\rangle \longrightarrow \\
&\langle \mathcal{I}d, \texttt{apply } mm(f, \texttt{[0,1,2]})\rangle \longrightarrow \\
&\langle \texttt{apply } mm(\square, \texttt{[0,1,2]}) :: \mathcal{I}d, f\rangle \longrightarrow \\
&\langle \texttt{apply } mm(f, \square) :: \mathcal{I}d, \texttt{[0,1,2]}\rangle \longrightarrow^* \\
&\langle \texttt{[2|}\square\texttt{]} :: \texttt{[1|}\square\texttt{]} :: \texttt{[0|}\square\texttt{]} :: \texttt{apply } mm(f, \square) :: \mathcal{I}d, \texttt{[]}\rangle \longrightarrow^* \\
&\langle \texttt{apply } mm(f, \square) :: \mathcal{I}d, \texttt{[0,1,2]}\rangle
\end{aligned}$$

Thereafter, the function mm is applied by substituting the previous list into its body expression. The pattern in the `case` expression matches the parameter list, thus the first clause will be evaluated.

$$\begin{aligned}
&\langle \texttt{apply } mm(f, \square) :: \mathcal{I}d, \texttt{[0,1,2]}\rangle \longrightarrow \\
&\langle \mathcal{I}d, \texttt{case [0,1,2] of} \\
&\qquad \texttt{[H|T] then [apply } f(\texttt{H})\texttt{|apply } mm(f, \texttt{T})\texttt{] else []}\rangle \longrightarrow \\
&\langle \mathcal{I}d, \texttt{[apply } f(0)\texttt{|apply } mm(f, \texttt{[1,2]})\texttt{]}\rangle
\end{aligned}$$

Next, we continue the evaluation with the tail of the list (since lists are evaluated backwards). Again we evaluate the application of mm and reduce the `case` expression, etc. The recursion stops when the list has been consumed. The last sublist, `[]` will not match the pattern of the `case` expression, thus the application of mm will leave `[]` unchanged while being removed from the stack. These reduction steps built up a sequence of applications inside the stack.

$$\begin{aligned}
&\langle \mathcal{I}d, \texttt{[apply } f(0)\texttt{|apply } mm(f, \texttt{[1,2]})\texttt{]}\rangle \longrightarrow^* \\
&\langle \texttt{apply } mm(f, \square) :: \texttt{[apply } f(0)\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[1,2]}\rangle \longrightarrow^* \\
&\langle \texttt{apply } mm(f, \square) :: \texttt{[apply } f(1)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(0)\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[2]}\rangle \longrightarrow^* \\
&\langle \texttt{apply } mm(f, \square) :: \texttt{[apply } f(2)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(1)\texttt{|}\square\texttt{]} :: \\
&\quad \texttt{[apply } f(0)\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[]}\rangle \longrightarrow^* \\
&\langle \texttt{[apply } f(2)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(1)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(0)\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[]}\rangle
\end{aligned}$$

Thereafter, the function applications can be evaluated for the elements of the list. First, the top element of the frame is extracted while `[]` is placed back. The application of f increases `2` to `3`. Combining the top element of the frame (`[`$\square$`|[]]`) and `3`, we obtain the list value `[3]`.

$$\begin{aligned}
&\langle \texttt{[apply } f(2)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(1)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(0)\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[]}\rangle \longrightarrow \\
&\langle \texttt{[}\square\texttt{|[]]} :: \texttt{[apply } f(1)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(0)\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{apply } f(2)\rangle \longrightarrow^* \\
&\langle \texttt{[}\square\texttt{|[]]} :: \texttt{[apply } f(1)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(0)\texttt{|}\square\texttt{]} :: \mathcal{I}d, 3\rangle \longrightarrow \\
&\langle \texttt{[apply } f(1)\texttt{|}\square\texttt{]} :: \texttt{[apply } f(0)\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[3]}\rangle
\end{aligned}$$

For the other two elements, we omit the previous steps and just show how the list inside the frame stack is reconstructed.

$$\langle \texttt{[apply } f(\texttt{1})\texttt{|}\square\texttt{]} :: \texttt{[apply } f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[3]} \rangle \longrightarrow^*$$
$$\langle \texttt{[apply } f(\texttt{0})\texttt{|}\square\texttt{]} :: \mathcal{I}d, \texttt{[2,3]} \rangle \longrightarrow^* \langle \mathcal{I}d, \texttt{[1,2,3]} \rangle$$

△

In the next section, we show how we built the concurrent semantics on top of the frame stack relation.

3.2 Processes, Signals and Actions

In this section we formalise the notions of *processes*, *signals* and *actions*, on which we build in the next two sections where we describe the concurrent semantics of Core Erlang, first the process-local semantics and then the inter-process semantics. In the remainder of this section we also establish some metatheoretical notation that we use in presenting the semantics.

Definition 3 (Core Erlang processes)**.** *A process ($p \in Process$) is either dead or alive.*

- *A live process is a quintuple $(K, e, q, pl, flag)$, where 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 (a list of process identifiers), and flag is the status of the* `'trap_exit'` *flag.*
- *A terminated (or dead) process is a list of linked process identifiers.*

As described earlier, Erlang and Core Erlang implement the actor model [1] for asynchronous communication between processes by message passing. Besides messages, there are other signals that can be sent between the processes (we formalise *exit*, *link*, and *unlink* signals beside messages) which potentially change the state of the process upon arrival without being put into the mailbox.

Actions represent the effects that characterise concurrency: message send and arrival in a mailbox, processing a mailbox with *receive*, process creation, and so on. An action will have an effect on individual processes (in the process-local semantics) and also *between* processes in the system level, inter-process semantics.

We define the following signals and actions of the semantics.

Definition 4 (Signals and Actions)**.**

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

Signals can be messages (parametrised by a value), exits (parametrised by a reason value and a flag whether the exit was sent through a link), links, and unlinks (which do not have parameters). The source and destination process identifiers are handled by actions, thus they are not included in the signals. We explain the syntax of actions as follows:

- Signal sending (*send*) and signal arrival (*arr*) actions carry a signal as a parameter, as well as the source and target process identifiers which are propagated from the inter-process semantics.
- *rec* actions have as parameter the message that is to be removed from the mailbox. There is no need to include process identifiers since these actions denote a process-local step and the removable message is already present in the mailbox of the process.
- *self* actions contain the identifier of the executing process as a parameter, which was obtained from the inter-process semantics.
- *spawn* actions include the new process identifier (propagated from the inter-process semantics), a function expression, and its actual parameters (as a Core Erlang list). The spawned process will execute this function with the given parameters.
- A sequential (τ) action denotes one reduction step with the sequential semantics.
- Termination ($\Downarrow$) actions denote either normal termination or the execution of the single-parameter `'exit'` BIF.
- *flag* actions denote the execution of the `'process_flag'` BIF (which does not necessarily change the state of the `'trap_exit'` flag).

Actions are used as the labels of the one-step evaluation relation. Next, we define the following *metatheoretical* functions and notations for the next sections:

- *tt* denotes the metatheoretical true, while *ff* denotes false.
- $x :: xs$ denotes a list with x as the first element and xs as the tail.
- $[]$ denotes the empty list.
- $[x_1, \ldots, x_n] = x_1 :: (x_2 :: (\ldots x_n :: []) \ldots)$.
- $rem_1(x, l)$: creates a list by removing the first occurrence of x from l.
- $rem(x, l)$: creates a list by removing all occurrences of x from l.
- $map(fn, l)$: constructs a list by applying the metatheoretical function fn to the elements of l.
- $l_1 +\!\!+ l_2$: constructs a list to represent the concatenation of l_1 and l_2.

- *convert*(*b*): maps *tt* to `'true'` and *ff* to `'false'`.
- *convert*(*v*): maps `'true'` to *Some tt* and `'false'` to *Some ff*, for other inputs, it returns *None*.

3.3 Process-Local Semantics

Next, we show the process-local semantics (see Section 2, Section 3, and Section 4), denoted by $p \xrightarrow{a} p'$, which describes the one-step evaluation of actions by individual processes. We primarily built this semantics by following the techniques of Fredlund's formalisation [15], since it has the widest coverage of language features among previous semantics. We note that the evaluation of the parameters of BIF calls `call` $e(e_1, \ldots, e_k)$ is handled by the sequential semantics (see Section 3.1), while the final reductions are formalised in the process-local level of BIF calls, with concrete BIF names, as shown in Section 3 below.

In the following, we make a brief description of the process-local reduction rules. The process identifiers in the reduction rules are propagated from the inter-process semantics via actions. First we detail the rule for sequential steps, and the rules for signal arrival (Figure 2):

- Seq lifts the computational layer to the process-local level. This is the sequential (τ) reduction rule of the semantics. In this rule, the computational layer could be replaced by any other frame stack semantics, such as a semantics for Erlang.
- Msg describes message arrival. Whenever a message arrives, it is appended to the mailbox of the process.
- ExitDrop describes when should an exit signal be dropped without modifying the state of the process [13, Receiving Exit Signals].
- ExitTerm describes when an exit signal terminates the process [13, Receiving Exit Signals]. The process becomes a terminated process by pairing the exit reason with the linked process identifiers. When an exit signal was sent explicitly, and the reason was `'kill'`[4], it also has to be converted to `'killed'` for the links (to prevent unnecessary termination of additional processes that are trapping exits).
- ExitConv describes when an exit signal should be converted to a message and appended at the end of the mailbox [13, Receiving Exit Signals]. This action can only occur when the `'trap_exit'` flag of the process is set.
- LinkArr, UnlinkArr rules describe arrival of link and unlink signals. In the first case, a process identifier is added to the links of the process, while in the second case, all occurrences of the process identifier are removed from the links.

[4]The `'kill'` reason causes unconditional termination almost always. We explain the only exception in Section 4.1 with Example 4.

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

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

$$\frac{(\iota_1 \neq \iota_2 \wedge b = ff \wedge v = \texttt{'normal'}) \vee (\iota_1 \notin pl \wedge b_e = tt \wedge \iota_1 \neq \iota_2)}{(K, e, q, pl, b) \xrightarrow{arr(\iota_1, \iota_2, exit(v, b_e))} (K, e, q, pl, b)} \quad \text{(ExitDrop)}$$

$$\frac{\begin{array}{c}(v = \texttt{'kill'} \wedge b_e = ff \wedge v' = \texttt{'killed'}) \vee \\ (b = ff \wedge v \neq \texttt{'normal'} \wedge v' = v \wedge (b_e = tt \to \iota_1 \in pl)) \vee \\ (b = ff \wedge v = \texttt{'normal'} = v' \wedge \iota_1 = \iota_2)\end{array}}{(K, e, q, pl, b) \xrightarrow{arr(\iota_1, \iota_2, exit(v, b_e))} map\ (\lambda\iota \Rightarrow (\iota, v'))\ pl} \quad \text{(ExitTerm)}$$

$$\frac{b = tt \wedge ((b_e = ff \wedge v \neq \texttt{'kill'}) \vee (b_e = tt \wedge \iota_1 \in pl))}{(K, e, q, pl, b) \xrightarrow{arr(\iota_1, \iota_2, exit(v, b_e))} (K, e, q +\!\!+ [\texttt{['EXIT',}\iota_1\texttt{,}v\texttt{]}], pl, b)} \quad \text{(ExitConv)}$$

$$(K, e, q, pl, b) \xrightarrow{arr(\iota_1, \iota_2, link)} (K, e, q, \iota_1 :: pl, b) \quad \text{(LinkArr)}$$

$$(K, e, q, pl, b) \xrightarrow{arr(\iota_1, \iota_2, unlink)} (K, e, q, rem(\iota_1, pl), b) \quad \text{(UnlinkArr)}$$

Figure 2: Process local semantics (part 1)

Next, we describe the formal rules of signal sending (Figure 3):

- Send describes message sending. If the BIF `'!'` is on the top of the frame stack with the target process identifier, and the message is evaluated to a value, a send action is emitted containing the source (which is propagated from the inter-process semantics in the NSend rule) and target identifiers and the message value, while the send expression itself is reduced to the message value.
- Exit describes explicitly sending an exit signal to a process. If the two-parameter `'exit'` BIF is on the top of the frame stack with the target process

identifier, and the reason is evaluated to a value, an exit action is emitted with the source (which is propagated from the inter-process semantics in NSend), target identifiers, and the exit reason value, while the expression is reduced to `'true'`. Note that when sending an explicit exit signal, the link flag of the signal is false.

- Link, Unlink rules both reduce the evaluable expression to `'ok'`. In the first case, a link signal is emitted with the source and target identifier, and the target is appended to the links of the process. In the second case, an unlink signal is emitted with the source and target identifier, and the target is removed from the links of the process.
- Dead describes the communication of a terminated process. In this rule, the first item of the links of the dead process is removed while an exit signal is emitted to the target with the reason that is specified in this first item. Note that the link flag of this exit signal is *true*, because this exit is sent through a link.

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

$$(\texttt{call 'exit'}(\iota_2, \square) :: K, v, q, pl, b) \xrightarrow{send(\iota_1, \iota_2, exit(v, ff))} (K, \texttt{'true'}, q, pl, b) \quad \text{(Exit)}$$

$$(\texttt{call 'link'}(\square) :: K, \iota_2, q, pl, b) \xrightarrow{send(\iota_1, \iota_2, link)} (K, \texttt{'ok'}, q, \iota_2 :: pl, b) \quad \text{(Link)}$$

$$(\texttt{call 'unlink'}(\square) :: K, \iota_2, q, pl, b) \xrightarrow{send(\iota_1, \iota_2, unlink)} (K, \texttt{'ok'}, q, rem(\iota_2, pl), b) \quad \text{(Unlink)}$$

$$(\iota_2, v) :: pl \xrightarrow{send(\iota_1, \iota_2, exit(v, tt))} pl \quad \text{(Dead)}$$

Figure 3: Process-local semantics (part 2)

Finally, we detail the rest of the process-local rules (Figure 4):

- Self receives the identifier of the process from the inter-process semantics, and evaluates the `'self'` BIF call to this identifier.
- Spawn describes process creation. The spawned process receives its identifier from the inter-process semantics, and this identifier will be the result of this

rule. Note that it is necessary that the first parameter of the `'spawn'` is a function value, while the second is a correct, object-level parameter list (which is checked in the inter-process semantics).

- RECEIVE describes message processing. With pattern matching, the first (oldest) message is selected from the mailbox of the process that matches any clause of the `receive` expression (if more patterns are matching to the same message, the first matching clause is selected). The evaluation continues with the body expression of the selected clause, substituted by the result (pattern variable - value) bindings.
- FLAG describes when the process flag `'trap_exit'` changes. The result of this rule is the original value of the flag.
- TERM describes normal termination, i.e. there are no more continuations in the frame stack, and the evaluable expression has already been reduced to a value. The result is a dead process, which will send exit signals to its links with the reason `'normal'`.
- EXITSELF describes the call of the single-parameter `'exit'` BIF. It imme-

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

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

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

$$\frac{convert(v) = Some\ v' \qquad v'' = convert(b)}{(\texttt{call}\ \texttt{'process\_flag'}(\texttt{'trap\_exit'}, \Box) :: K, v, q, pl, b) \xrightarrow{flag} (K, v'', q, pl, v')} \qquad \text{(FLAG)}$$

$$(\mathcal{J}d, v, q, pl, b) \xrightarrow{\Downarrow} map\ (\lambda\iota \Rightarrow (\iota, \texttt{'normal'}))\ pl \qquad \text{(TERM)}$$

$$(\texttt{call}\ \texttt{'exit'}() :: K, v, q, pl, b) \xrightarrow{\Downarrow} map\ (\lambda\iota \Rightarrow (\iota, v))\ pl \qquad \text{(EXITSELF)}$$

Figure 4: Process-local semantics (part 3)

diately terminates the process, and exit signals will be sent to the linked processes with the parameter reason value. We note that when introducing exceptions in the future, this version of exit signals will be capable of being caught by exception handlers.

3.4 Inter-Process Semantics

In this section we discuss the inter-process reduction rules for the semantics. The advantage of this formalisation is that the dynamic semantics of the system is described by only 5 rules (by combining the rules from related work [15, 18, 23] with the same premises but different actions), which resulted in shorter proofs. First, we introduce the necessary concepts.

Definition 5 (Ether)**.** *An ether (denoted by Δ) is a mapping of source and target identifier pairs to lists of signals. We use $\emptyset$ to denote the empty ether, which maps everything to the empty list*[5]*.*

We use an ether to express non-atomic signal passing (unlike a number of related works [15, 18]); that is, the signals sent from one process do not arrive immediately, but they are transferred via the ether. This is also described in the reference manual [13]: "The amount of time that passes between the time a signal is sent and the arrival of the signal at the destination is unspecified but positive".

In addition, the ether is used to implement the signal ordering guarantee [13], that is "if an entity sends multiple signals to the same destination entity, the order is preserved". If a source sends multiple signals to the same target, these signals will be appended to the end of the list associated with the source and target in the ether. However, if multiple processes send signals to the same destination, the arrival order of these signals is not specified, thus they are included in separate lists in the ether based on their source.

Definition 6 (Node)**.** *A node is a pair ($(\Delta, \Pi) \in Node$) of an ether and a process pool. The process pool (denoted by Π) is a mapping that associates process identifiers with processes. We denote nodes with Σ and the empty process pool with $\emptyset$.*

On top of these concepts, we introduce notations and metatheoretical functions:

- $\iota : p \parallel \Pi$: Appends process p associated with the identifier ι to the process pool Π. In formalising this we used function update, so that the order of identifiers is irrelevant. Because of this, we are justified in abusing the notation somewhat when we write the rules using pattern matching: without loss of generality, we assume that the item of interest appears in the head position of the collection of processes given.
- $remFirst(\Delta, \iota, \iota')$: Removes the first element in the ether Δ from the list associated with ι source and ι' destination, and returns a pair of this removed

[5] In the implementation, we formalised the ether as a function which maps (source) process identifiers to a function mapping (target) process identifiers to a list of signals.

signal and the result ether inside *Some*. If the associated list was empty, it returns *None*.

- $\Delta[(\iota, \iota') \overset{+}{\mapsto} s]$: Creates an ether by appending the signal s to the end of the list associated with ι source and ι' destination in the ether Δ (while keeping other parts of Δ unchanged).
- $\Pi \setminus \iota$: Creates a process pool by removing the process associated with ι from process pool Π. This operation was also formalised by function updates.
- $\iota \notin \Pi$: Checks whether there is no process associated with ι in Π.
- *convert_list*(vs): Creates a metatheoretical list of expressions based on an object-level Core Erlang list (constructed with `[_|_]` and `[]`); if successful the result is wrapped with a *Some* constructor; if not, *None* is returned.

Next, we define the semantics in Figure 5. This one-step reduction is denoted by $\Sigma \xrightarrow{\iota:a} \Sigma'$ that means the node Σ is reduced to Σ' by taking a reduction step determined by the action a with the process identified by ι. The rules always include a "first" process $(\iota : p \parallel \Pi)$, nevertheless, any process from the pool can take this place, since any process in a $\parallel$ chain can be the outermost one, as mentioned before. We give a brief, informal description of the inter-process rules now:

- NSend describes signal sending. While a process with the identifier ι is reduced by emitting a send action, the contents of this action (target, source, and signal) are placed into the ether.
- NArrive describes how an element is (nondeterministically) removed from the ether. Any signal can be removed from the lists in the ether, if the signal is the first element of that list, and there is a live process with the destination identifier in the process pool.
- NTerm describes how a process identifier is freed. When a dead process has notified all of its links, its identifier is removed from the process pool.
- NSpawn describes the creation of a new process. The new process is assigned a non-used identifier, and it starts evaluating the function application described in the spawn action of the rule (note that conversion from object-level to meta-level lists is needed). The initial configuration of the new process is the empty frame stack (continuation), the given function application as the evaluable expression, empty mailbox, it has no links, and it does not trap exit signals.
- NOther describes the reduction in case of any other action, that is, this rule propagates these actions to the process-local level.

We note that in every rule of this semantics, exactly one process is reduced. Furthermore, all reduction rules (except NTerm) actually propagate the action to the process-local semantics, while modifying the ether or the process pool. We also introduce the following notations on top of the inter-process semantics:

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

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

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

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

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

Figure 5: Formal semantics of communication between processes

- $\Sigma \xrightarrow{l} {}^*\Sigma'$ denotes a special reflexive, transitive closure of the relation $\xrightarrow{\iota:a}$, which traces the actions in the list l in forms of (ι, a) pairs. We use $\Sigma \xrightarrow{\cdots} {}^*\Sigma'$ when the trace is not relevant. For example, if a node Σ can be reduced to Σ' in the three following steps: 1) the process identified by ι sends a message v to the process identified by ι', 2) this message arrives to the target, 3) the message is received by the target, we use

$$\Sigma \xrightarrow{[(\iota,send(\iota,\iota',msg(v))),(\iota',arr(\iota,\iota',msg(v))),(\iota',rec(v))]} {}^*\Sigma'.$$

- $\Sigma \longrightarrow^* \Sigma$ denotes a reduction sequence from node Σ to node Σ' that contains only sequential (τ) reduction steps.

Discussion. There are other approaches (e.g. [15]) which define fewer actions for the semantics by defining input, output, spawn, and silent actions. In these approaches, $rec(v), flag, \Downarrow, \tau$ could all be handled as silent actions, since they affect the state of a single process and do not communicate. Still, we formalised the more fine-grained version, because this allows us to group the rules of the semantics into

more categories. By coupling the aforementioned actions, the less detailed approach can also be simulated. Moreover, we proved theorems (specifically, Theorem 9) which would not be provable if other actions were also considered to be silent.

4 Semantics Validation

After defining a formal semantics, the next step is to validate it [5]. We use two approaches: 1) we evaluate simple parallel programs and compare the results to the results of the Erlang/OTP compiler, and 2) we prove properties of the semantics. We are also investigating ways in which the concurrent semantics can be executed efficiently, which is a necessary step to enable extensive validation against the reference implementation.

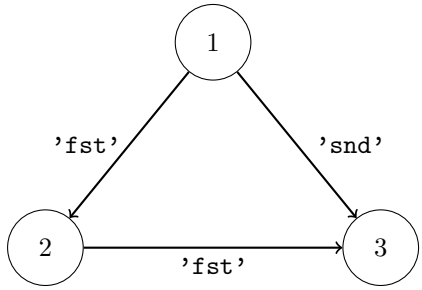

Figure 6: Actor diagram for Example 3

4.1 Example Program Evaluation

In this section we present some simple program evaluation case studies that demonstrate how the semantics operates.

Example 3 (Signal ordering)**.** The first example illustrates when the signal ordering guarantee cannot be applied. Let us consider three processes (with the identifiers 1, 2, 3), which evaluate the following expressions.

1. `let X = call '!'(2, 'fst') in call '!'(3, 'snd')`
2. `receive X -> call '!'(3, X) end`
3. `receive X -> X end`

Next, we construct a node with the empty ether from these processes, and start evaluating it. We use Π, to denote the process pool constructed from 2 and 3. For simplicity, we omit the list of linked processes and the trap flag, since they are not used during this evaluation. First, we reduce process 1, since the others are all

blocked by `receive` expressions. This evaluation puts the two messages into the ether.

$$(\emptyset, 1 : (\mathcal{I}d, \texttt{let X = call '!'(2, 'fst') in call '!'(3, 'snd')}, []) \parallel \Pi) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} msg(\texttt{'fst'})][(1,3) \overset{+}{\mapsto} msg(\texttt{'snd'})], 1 : (\mathcal{I}d, \texttt{'snd'}, []) \parallel \Pi) \qquad (\mathit{Init})$$

We denote the result process pool with Π_1 without process 3. Next, we can evaluate process 3, to which the message `'snd'` arrives. Then the `receive` expression removes it from the mailbox and processes it. Thus the final value upon termination in process 3 is `'snd'`.

$$(\emptyset[(1,2) \overset{+}{\mapsto} msg(\texttt{'fst'})][(1,3) \overset{+}{\mapsto} msg(\texttt{'snd'})],$$
$$3 : (\mathcal{I}d, \texttt{receive X -> X end}, []) \parallel \Pi_1) \xrightarrow{3:arr(1,3,msg(\texttt{'snd'}))}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} msg(\texttt{'fst'})], 3 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{'snd'}]) \parallel \Pi_1) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} msg(\texttt{'fst'})], 3 : (\mathcal{I}d, \texttt{'snd'}, []) \parallel \Pi_1)$$

Note that the last configuration we presented above could still progress, because process 2 can receive and forward the message `'fst'`.

However, this was not the only option to evaluate this simple program. Instead of evaluating process 3 in the previous reductions, we can progress with process 2 (from the state reached in *Init*). We denote the process pool containing the terminated process 1 and process 3 with Π_2. First, the message `'fst'` arrives to process 2 which removes it from the mailbox, and forwards it to process 3.

$$(\emptyset[(1,2) \overset{+}{\mapsto} msg(\texttt{'fst'})][(1,3) \overset{+}{\mapsto} msg(\texttt{'snd'})],$$
$$2 : (\mathcal{I}d, \texttt{receive X -> call '!'(3, X) end}, []) \parallel \Pi_2) \xrightarrow{2:arr(1,2,msg(\texttt{'fst'}))}$$
$$(\emptyset[(1,3) \overset{+}{\mapsto} msg(\texttt{'snd'})],$$
$$2 : (\mathcal{I}d, \texttt{receive X -> call '!'(3, X) end}, [\texttt{'fst'}]) \parallel \Pi_2) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,3) \overset{+}{\mapsto} msg(\texttt{'snd'})][(2,3) \overset{+}{\mapsto} msg(\texttt{'fst'})], 2 : (\mathcal{I}d, \texttt{'fst'}, []) \parallel \Pi_2)$$

After processes 1 and 2 are terminated (we denote the pool containing these with Π_3), we evaluate process 3. At this point, either of the messages in the ether could arrive first at process 3, which will be processed then by the `receive` expression,

since their source is different. We present the case when `'fst'` arrives first.

$$(\emptyset[(1,3) \overset{+}{\mapsto} msg(\texttt{'snd'})][(2,3) \overset{+}{\mapsto} msg(\texttt{'fst'})],$$
$$3 : (\mathcal{I}d, \texttt{receive X -> X end}, []) \parallel \Pi_3) \xrightarrow{3:arr(2,3,msg(\texttt{'fst'}))}$$
$$(\emptyset[(1,3) \overset{+}{\mapsto} msg(\texttt{'snd'})],$$
$$3 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{'fst'}]) \parallel \Pi_3) \xrightarrow{3:arr(1,3,msg(\texttt{'snd'}))}$$
$$(\emptyset, 3 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{'fst'}, \texttt{'snd'}]) \parallel \Pi_3) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} msg(\texttt{'snd'})], 3 : (\mathcal{I}d, \texttt{'fst'}, [\texttt{'snd'}]) \parallel \Pi_3)$$

However, with this reduction sequence, process 3 terminates with `'fst'`. The signal ordering guarantee was not applicable in this scenario, because the messages that process 3 received are from different sources. △

Example 4 (Exit signals)**.** Next, we present an example about sending exit signals, specifically we show the difference between the one- and two-parameter `'exit'` BIFs. Consider two processes:

1. `let X = call 'link'(2) in call 'exit'(1, 'kill')`
2. `receive X -> X end`

The second process is set to trap exit signals. Once again, we start the evaluation with the first process. In the first steps, process 1 creates the link between the two processes:

$$(\emptyset, 1 : (\mathcal{I}d, \texttt{let X = call 'link'(2) in call 'exit'(1, 'kill')}, [], [], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [], \mathit{tt}) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} \mathit{link}], 1 : (\mathcal{I}d, \texttt{call 'exit'(1, 'kill')}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [], \mathit{tt}) \parallel \emptyset) \xrightarrow{2:arr(1,2,\mathit{link})}$$
$$(\emptyset, 1 : (\mathcal{I}d, \texttt{call 'exit'(1, 'kill')}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], \mathit{tt}) \parallel \emptyset)$$

(*Link*)

Next, the first process terminates itself with the two-parameter `'exit'`. This involves multiple reduction steps, because the signal needs to be put into the ether, and then retrieved from it. Then the reason will be converted to `'killed'` because the two-parameter `'exit'` always sets the link flag of the exit signal to *ff*.

$$(\emptyset, 1 : (\mathcal{I}d, \texttt{call 'exit'(1, 'kill')}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], \mathit{tt}) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,1) \overset{+}{\mapsto} \mathit{exit}(\texttt{'kill'}, \mathit{ff})], 1 : (\mathcal{I}d, \texttt{'true'}, [], [2], \mathit{ff}) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], \mathit{tt}) \parallel \emptyset) \xrightarrow{1:arr(1,1,\mathit{exit}(\texttt{'kill'},\mathit{ff}))}$$
$$(\emptyset, 1 : [(2, \texttt{'killed'})] \parallel 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], \mathit{tt}) \parallel \emptyset)$$

Next, we propagate the exit signal through the link, and it will be converted to a message because of the trap flag in the execution of process 2.

$$(\emptyset, 1 : [(2, \texttt{'killed'})] \parallel 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} exit(\texttt{'killed'}, tt)],$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{2:arr(1,2,exit(\texttt{'killed'},tt))}$$
$$(\emptyset, 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{['EXIT', 1, 'killed']}], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset, 2 : (\mathcal{I}d, \texttt{['EXIT', 1, 'killed']}, [], [1], tt) \parallel \emptyset$$

However, if we use the single parameter `'exit'` BIF, the reduction would be carried out otherwise. We start the evaluation from the analogous state to the point *Link* above. It immediately terminates the process without sending signals into the ether. This also causes the reason `'kill'` not to be converted to `'killed'`. Next, this exit signal will be sent through a link (the link flag of the signal is tt), which enables the use of ExitConv in process 2.

$$(\emptyset, 1 : (\mathcal{I}d, \texttt{call 'exit'('kill')}, [], [2], ff) \parallel$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset, 1 : [(2, \texttt{'kill'})] \parallel 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset[(1,2) \overset{+}{\mapsto} exit(\texttt{'kill'}, tt)],$$
$$2 : (\mathcal{I}d, \texttt{receive X -> X end}, [], [1], tt) \parallel \emptyset) \xrightarrow{2:arr(1,2,exit(\texttt{'kill'},tt))}$$
$$(\emptyset, 2 : (\mathcal{I}d, \texttt{receive X -> X end}, [\texttt{['EXIT', 1, 'kill']}], [1], tt) \parallel \emptyset) \xrightarrow{\cdots}{}^{*}$$
$$(\emptyset, 2 : (\mathcal{I}d, \texttt{['EXIT', 1, 'kill']}, [], [1], tt) \parallel \emptyset$$

We should note that the `'kill'` reason is normally used to terminate a process regardless of its current state. Although, in this case (when the signal is sent through a link, i.e. its link flag is set) `'kill'` does *not* terminate the process in question. △

4.2 Properties of the Semantics

After formally evaluating simple programs, we proved some fundamental properties of the layers of the semantics, and formalised the proofs in the Coq theorem prover [9]. In this section we highlight the most important properties, and provide sketches of the proofs; for more insights, we refer to the formalisation. First, we show the determinism of the sequential and process-local levels.

Theorem 1 (Sequential and process-local evaluation is deterministic)**.** *For all frame stacks K, K', K'' and expressions e, e', e'', if $\langle K, e\rangle \longrightarrow \langle K', e'\rangle$ and $\langle K, e\rangle \longrightarrow \langle K'', e''\rangle$, then $K' = K''$ and $e' = e''$.*

Similarly, for all processes p, p', p'', and actions a, if $p \xrightarrow{a} p'$ and $p \xrightarrow{a} p''$, then $p' = p''$.

Proof. To prove determinism (in both semantics), we carried out case distinction based on the two reduction premises. If both use the same reduction rule, their result is equal, otherwise a contradiction is found between the premises of the different rules. □

The determinism of these layers of the semantics is a natural property; one process should handle an incoming action in the same way in the same inner state. However, we found that the conditions in the reference manual [13, Receiving Exit Signals] are ambiguous in the way that they describe how to handle exit signals. We checked with the reference implementation what the correct conditions are, and encoded them in the premises for reduction rules about exit signals: ExitConv, ExitDrop, and ExitTerm.

In the previous sections, we emphasised why the concept of the ether is necessary to ensure the signal ordering guarantee. We formally verified this property.

Theorem 2 (Signal ordering guarantee)**.** *For all nodes* $\Sigma_1, \Sigma_2, \Sigma_3$*, process identifiers* ι, ι'*, and unique signals*[6] $s_1 \neq s_2$*, 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:* $\Sigma_4 \xrightarrow{\iota':arr(\iota,\iota',s_2)} \Sigma_5$.

Proof. We proved this theorem by induction on the length of the reduction chain $\Sigma_3 \xrightarrow{l} {}^*\Sigma_4$. In the base case, the first removable element in the ether is either s_1 or another signal which is different from s_2. In the inductive case, we suppose that there is a reduction chain of length k which does not remove s_1 from the ether. Then there is the $(k+1)$th reduction step, which also cannot remove s_1 from the ether, based on the hypotheses. Thus once again, the first removable element from the ether is either s_1 or another signal which is different from s_2. □

This theorem informally states the following: if two signals have been sent from the same sender to the same target, after taking any number of reduction steps, which do not contain the arrival of the first signal, it is not possible that the second signal will arrive to the target.

We also proved a number of confluence properties, which are the basis of proving bisimulation-based program equivalence. Our goal is to prove that sequential evaluation ($\Sigma \longrightarrow^* \Sigma'$) produces equivalent nodes. The first theorem expresses that a sequential reduction can be carried out after another reduction step if this step does not terminate the process. Otherwise, the sequential reduction cannot be executed after the other action. This property holds for both process-local and inter-process semantics.

Theorem 3 (Confluence of sequential reductions in the same process)**.** *For all processes* p_1, p_2, p_2'*, and action* a*, supposing that* $p_1 \xrightarrow{\tau} p_2$ *and* $p_1 \xrightarrow{a} p_2'$*, then there exists a process* p_3 *that satisfies* $p_2 \xrightarrow{a} p_3$ *and* $(p_2' \xrightarrow{\tau} p_3 \vee p_2' = p_3)$.

[6] They are different from any other signal in the starting configuration.

Similarly, for all nodes $\Sigma_1, \Sigma_2, \Sigma_2'$*, process identifiers* ι*, and actions* a*, supposing that* $\Sigma_1 \xrightarrow{\iota:\tau} \Sigma_2$ *and* $\Sigma_1 \xrightarrow{\iota:a} \Sigma_2'$*, then there exists a node* Σ_3 *that satisfies* $\Sigma_2 \xrightarrow{\iota:a} \Sigma_3$ *and* $(\Sigma_2' \xrightarrow{\iota:\tau} \Sigma_3 \vee \Sigma_2' = \Sigma_3)$*.*

Proof. The proof for both semantics relies on case distinction in the derivation of $\Sigma_1 \xrightarrow{\iota:a} \Sigma_2'$. There are actually two separate cases:

- If the action a does not terminate the process (still, it potentially modifies either the mailbox, or the list of linked processes, or the `'trap_exit'` flag), then the sequential reduction step can be taken after this action too, since these steps are not influenced by the mentioned attributes of the process.
- If the action a terminates the process, then p_2' and p_3 denote the same terminated process, since sequential steps do not modify the list of linked processes, which is the only attribute of a live process that is kept when it terminates.

□

This theorem is used when two reductions for the same process need to be chained after each other. Actually, this theorem is a stepping stone towards proving Theorem 5.

The next theorem concerns different processes: possible reduction steps can be carried out after each other if they are not both *spawn* actions.

Theorem 4 (Action ordering)**.** *For all nodes* $\Sigma_1, \Sigma_2, \Sigma_2'$*, process identifiers* $\iota \neq \iota'$*, actions* a, a'*, which are not both spawn actions, if* $\Sigma_1 \xrightarrow{\iota:a} \Sigma_2$ *and* $\Sigma_1 \xrightarrow{\iota':a'} \Sigma_2'$ *then there exists a node* Σ_3*, which can be reached from* Σ_2 *with action* a'*:* $\Sigma_2 \xrightarrow{\iota':a'} \Sigma_3$*.*

Proof. This theorem is proved by case separation on the two reduction premises. There is no scheduling algorithm formalised in the semantics, thus any process can be reduced if it is not in a stuck configuration (i.e. if it is waiting for a message to evaluate a `receive` expression). We can define any order for the reductions of different processes (except if both reductions are labelled by *spawn* actions), because both of the reductions in the premise can always be carried out. The only action a in the first reduction that could prevent making the second reduction (with action a') is the arrival of an exit signal that terminates the process identified by ι', but the premise $\iota \neq \iota'$ rules this case out. □

The premise that restricts *spawn* actions is necessary because we cannot assure that these spawned processes obtain the same process identifiers if their spawn order is reversed (currently, the semantics assigns fresh process identifiers to spawned processes based on the list of process identifiers already in use). This theorem is also a stepping stone towards Theorem 6.

The following theorems contain any-step reduction chains. The first of these theorem expresses that if there are τ actions and an additional action that can be executed in a configuration, then either this additional action can be executed at the final node after executing the chain, or it was τ-reduction inside the chain.

Theorem 5 (Chaining a reduction to the end of an sequential sequence)**.** *For all nodes* $\Sigma_1, \Sigma_4, \Sigma_4'$*, process identifier* ι*, action* a*, and action traces* l*, which only include internal actions paired with any process identifiers, if* $\Sigma_1 \xrightarrow{l} {}^*\Sigma_4$ *and* $\Sigma_1 \xrightarrow{\iota:a} \Sigma_4'$*, then there are two potential scenarios:*

- *Either there is a node* Σ_5 *which can be reached by a reduction from* Σ_4*:* $\Sigma_4 \xrightarrow{\iota:a} \Sigma_5$*.*
- *Or* $a = \tau$ *and there are nodes* Σ_2, Σ_3 *and action traces* l_1, l_2*, which can be used to split the sequential reduction steps:* $\Sigma_1 \xrightarrow{l_1} {}^*\Sigma_2$*,* $\Sigma_2 \xrightarrow{\iota:a} \Sigma_3$ *and* $\Sigma_3 \xrightarrow{l_2} {}^*\Sigma_4$*, moreover* $l = l_1 +\!\!+ [(\iota, a)] +\!\!+ l_2$*.*

Proof. We proved this theorem by induction on the reduction chain $\Sigma_1 \xrightarrow{l} {}^*\Sigma_4$. The base case is solved by the premise $\Sigma_1 \xrightarrow{\iota:a} \Sigma_4'$ (by choosing $\Sigma_5 = \Sigma_4'$), since $\Sigma_1 \xrightarrow{l} {}^*\Sigma_4$ was a 0-step reduction, thus $\Sigma_1 = \Sigma_4$. In the inductive case, we did case distinction whether $a = \tau$ and (ι, a) is included in l. If this is not true, we can make the reduction determined by (ι, a) from the configuration Σ_4 based on the induction hypothesis and Theorem 3. Otherwise $(\iota, a) = (\iota, \tau)$ is included in the action trace l. □

This theorem expresses one of the fundamental properties needed to prove that $\longrightarrow^*$ is a weak bisimulation, (Theorem 9 below). The next theorem is the other fundamental property required. If there is an action that is executed at the end of the execution of a sequential reduction chain, and it can be executed in the starting configuration too, then from the result of the second derivation the result of the first one can be reached by only sequential steps.

Theorem 6 (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* $\Sigma_2' \longrightarrow^* \Sigma_3$*.*

Proof. The proof of this property is also carried out by induction on the reduction chain $\Sigma_1 \longrightarrow^* \Sigma_2$. In the base case $\Sigma_1 = \Sigma_2$ and by Theorem 1, $\Sigma_2' = \Sigma_3$, while the proof of the inductive case is based on Theorem 4 and the induction hypothesis. □

What if this potentially non-sequential action was the arrival of an exit signal? That will potentially terminate a process, which could have taken some internal steps. We note that with the $\longrightarrow^*$ reduction in the conclusion we do not say that the steps are preserved, thus the result node after the arrival of the exit signal can take fewer internal steps by leaving the steps for the terminated process out.

5 Program Equivalence

In this section, we investigate program equivalence using bisimulation. Bisimulations are relations between nodes that are preserved by the reduction steps.

Definition 7 (Bisimulation)**.** *A relation R is a bisimulation if it satisfies the following two properties:*

- *For all nodes $\Sigma_1, \Sigma_2, \Sigma_1'$, process identifiers ι, and actions a, if $(\Sigma_1, \Sigma_2) \in R$ and $\Sigma_1 \xrightarrow{\iota:a} \Sigma_1'$, then there is a node Σ_2', which is reducible from Σ_2 with the action a: $\Sigma_2 \xrightarrow{\iota:a} \Sigma_2'$, and $(\Sigma_1', \Sigma_2') \in R$.*
- *For all nodes $\Sigma_1, \Sigma_2, \Sigma_2'$, process identifiers ι, and actions a, if $(\Sigma_1, \Sigma_2) \in R$ and $\Sigma_2 \xrightarrow{\iota:a} \Sigma_2'$, then there is a node Σ_1', which is reducible from Σ_1 with the action a: $\Sigma_1 \xrightarrow{\iota:a} \Sigma_1'$, and $(\Sigma_1', \Sigma_2') \in R$.*

We can show that equality satisfies the above conditions of being a bisimulation.

Theorem 7. *The equality of nodes is a bisimulation.*

Proof. This property is just a simple consequence of the definition of bisimulation. □

We also defined a relaxed variant: weak bisimulations omit τ actions taken in the semantics, so only communication actions should preserve the relation.

Definition 8 (Weak bisimulation)**.** *A relation R is a weak bisimulation if it satisfies the following two properties:*

- *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 $(\Sigma_1', \Sigma_2') \in R$.*

Bisimulations satisfy the natural property of being weak bisimulations.

Theorem 8. *Bisimulations are weak bisimulations.*

Proof. This property is also a simple consequence of the definitions, since we can choose 0-step reductions for $\Sigma_2 \longrightarrow^* \Sigma_2^1$ and $\Sigma_2^2 \longrightarrow^* \Sigma_2'$ in Definition 8, while the middle step $\Sigma_2^1 \xrightarrow{\iota:a} \Sigma_2^2$ is obtained from Definition 7. □

We consider two programs (Σ, Σ') equivalent if there is a relation R that is a weak bisimulation and $(\Sigma, \Sigma') \in R$. Next, we prove that sequential evaluation is a weak bisimulation. For this proof we used the chaining properties (Theorem 5 and Theorem 6) of the semantics.

Theorem 9. $\longrightarrow^*$ *(between nodes) is a weak bisimulation.*

Proof. To avoid ambiguity, we use Λ to denote the available nodes in the proof, while we keep Σ for the definitions. To prove that a relation is a weak bisimulation, two properties need to be proved:

- For the first part of Definition 8 we have $\Lambda_1 \longrightarrow^* \Lambda_2$ and $\Lambda_1 \xrightarrow{\iota:a} \Lambda_1'$ as assumptions. We can chain the reduction determined by a to the end of the sequential reduction sequence by Theorem 5 ($\Lambda_2 \xrightarrow{\iota:a} \Lambda_3$ for some Λ_3). Actually, the second possible conclusion (i.e. the $a = \tau$) of this theorem can not occur here, because of the restriction $a \neq \tau$ in Definition 8. We need to prove that $\Lambda_2 \longrightarrow^* \Sigma_2^1$, $\Sigma_2^1 \xrightarrow{\iota:a} \Sigma_2^2$, $\Sigma_2^2 \longrightarrow^* \Sigma_2'$, and $\Lambda_1' \longrightarrow^* \Sigma_2'$ for suitable Σ nodes. We can choose $\Sigma_2^1 = \Lambda_2$, $\Sigma_2^2 = \Lambda_3$, and $\Sigma_2' = \Lambda_3$, thus the first and second $\longrightarrow^*$ reductions are 0-step reductions, while the single-step reduction is among the assumptions. The reduction $\Sigma_1' \longrightarrow^* \Sigma_2'$ remains, which can be proved by Theorem 6.
- To satisfy the second part of Definition 8 we have $\Lambda_1 \longrightarrow^* \Lambda_2$ and $\Lambda_2 \xrightarrow{\iota:a} \Lambda_2'$ as assumptions. We need to prove $\Lambda_1 \longrightarrow^* \Sigma_1^1$, $\Sigma_1^1 \xrightarrow{\iota:a} \Sigma_1^2$, and $\Sigma_1^2 \longrightarrow^* \Sigma_1'$, and $\Sigma_1' \longrightarrow^* \Lambda_2'$ for suitable nodes. We choose $\Sigma_1^1 = \Lambda_2$, $\Sigma_1^2 = \Lambda_2'$, and $\Sigma_1' = \Lambda_2'$, thus the first two reductions are among the assumptions, while the third and fourth ones are 0-step reductions.

□

With this proof, we can state that a node is equivalent to the nodes to which it reduces by using sequential steps only. For example, we can derive the following property:

Example 5. For all nodes Π, ethers Δ, frame stacks K, process identifiers ι, mailboxes q, list of process identifiers pl, and process flags *flag*, the following nodes are equivalent (where mm denotes the function expression inside `letrec` in Example 1, and f denotes the successor function from Example 2).

$$(\Delta, \iota : (K, \texttt{letrec 'mm'/2} = mm \texttt{ in apply 'mm'/2}(f, \texttt{[0,1,2]}), q, pl, \mathit{flag}) \parallel \Pi)$$

$$(\Delta, \iota : (K, \texttt{[1,2,3]}, q, pl, \mathit{flag}) \parallel \Pi)$$

Proof. We have already shown in Example 2 how the complex `letrec` expression can be reduced to a list of values. Using this fact together with Theorem 9 we can prove this equivalence (note that the sequential steps of the evaluation can be lifted to the inter-process level with rules Seq and NOther). □

There is a natural question, whether any evaluation sequence could be proved to be a bisimulation. Unfortunately, that is not the case.

Theorem 10. *For all l action traces, $\xrightarrow{l}$ is not a weak bisimulation.*

Proof. We can prove this theorem by providing a counterexample. Here, we just give the idea of it: consider the process (with identifier 0) that evaluates `let X =`

`0 in X`. This process terminates in two sequential steps according to the semantics. By the definition of the weak bisimulation, taking a reduction step determined by any action (specifically for $arr(0, 1, exit(\texttt{'kill'}, \texttt{'false'}))$), the result configurations should be reducible to each other (by two sequential steps). `let X = 0 in X` evaluates to a dead process with the action above, which naturally could not be reduced to anything by sequential steps. □

In this section we defined program equivalence based on bisimulations and proved that sequential evaluation is a bisimulation. With the help of these definitions and theorems, we can establish the equivalence of simple programs. As we noted at the start of the section, all of the definitions and theorems presented here are formalised in the Coq proof assistant [9]. We plan to further investigate bisimulations to enable reasoning about more complex programs.

6 Related Work

The results presented in this paper are extensions to our work on sequential Core Erlang [2–4, 19]. We mainly based the concurrent semantics on the work of Fredlund [15], Harrison [18], and Lanese et al. [23]. The general idea of an interaction semantics of actor languages is described in the work of Mason and Talcott [24].

The formalisation of Fredlund [15] is the most detailed regarding both the sequential and concurrent parts of Erlang, which also faithfully follows the documentation of Erlang [12]. However, it considers signal transfer as an atomic operation (i.e. when a signal is sent, it immediately arrives), while according to signal ordering guarantee [13], the order of the signals sent from an entity to the same entity is preserved, which means that two signals that are targeting different entities can arrive in arbitrary order. The semantics of Fredlund [15] differentiates active and passive termination signals, while we denote these by the link flag of the exit signal.

Moreover, the work of Fredlund [15] differentiates only three actions on the inter-process level semantics: input, output, and silent. This approach closely follows the general idea of the interaction semantics [24]. With our approach, we can simulate input and output actions: *send* actions can be considered as output actions, *arr* actions are the input actions, while every other action can be regarded as silent. The advantage of our semantics is that we can distinguish more classes of reduction sequences, moreover, we also exploit this property: the theorems discussed in Section 4.2 and Section 5 involving sequential reductions would not hold, if other actions (e.g. *flag*) were considered as silent.

Lanese et al. [23] describe their results on bisimulations, and prove a number of system equivalences (e.g. renaming, normalisation). They also use ethers to store messages, however, their approach (deliberately) ignores the guarantee for signal ordering [13]. Moreover, they do not formalise signals except messages, and used only a small subset of Core Erlang. Still, we incorporated some of their ideas in the formalisation of program equivalence, and plan to pursue this topic more in detail.

The work of Vidal et al. [21, 22, 27, 31] is also related to ours, they define multiple semantics (reduction semantics and small-step semantics) for Core Erlang

to express reversible computation. The language they formalised has a similar coverage to our formalisation, they also formalised concurrent semantics with an ether and action traces, moreover, they also proved similar theorems about the properties. However, their formalisations do not include signals except messages.

Harrison [18] presented a formalisation of a minimal subset of Core Erlang in his paper, which has also been formalised in Isabelle. His formalisation techniques aided us while creating a usable Coq definition of the concurrent semantics, however, his formalisation includes only a few of the language elements, and he too treated signal transfer as an atomic operation.

An important advantage of our formalisation compared to most of the existing ones is that it is also implemented in Coq in an open-source project. Most of the existing works are paper-based formalisations or the machine-checked version is no longer available to the public. Furthermore, our semantics implements the signal ordering guarantee [13] more faithfully than the other discussed approaches.

7 Conclusion and Future Work

In this paper, we described our three-level, modular formal semantics for concurrent Core Erlang. We discussed a number of theorems about the determinism and confluence properties of the semantics, defined bisimulations to be able to reason about program equivalence, and proved that side-effect-free evaluations of a program provide equivalent programs. Finally, we compared our approach with the results of other authors. The formalisation has also been implemented as an open-source project in the Coq proof assistant [9].

In the future we are planning to further extend this formalisation. Our future goals include the following points:

- Investigating bisimulations in more depth, potentially by following a similar path to Lanese *et al* [23] who defined barbed congruence, that enabled them to develop a proof technique to effectively reason about program equivalence.
- Proving the equivalence between more complex examples of concurrent programs equivalent, as well as investigating equivalence between sequential and concurrent algorithms.
- Extending the semantics to cover exceptions and other side effects (e.g. input-output) based on our previous results [3].
- Implementing a formalisation of the module system within this semantics.
- In the longer term, an extensive, usable formalisation of Erlang is our ultimate goal.

References

[1] Agha, G. and Hewitt, C. Concurrent programming using actors: exploiting large-scale parallelism. In Bond, A.H. and Gasser, L., editors, *Readings in Distributed Artificial Intelligence*, pages 398–407. Morgan Kaufmann, 1988. DOI: 10.1016/B978-0-934613-63-7.50042-5.

[2] Bereczky, P., Horpácsi, D., and Thompson, S. A proof assistant based formalisation of a subset of sequential Core Erlang. In Byrski, A. and Hughes, J., editors, *Trends in Functional Programming*, pages 139–158, Cham, 2020. Springer International Publishing. DOI: 10.1007/978-3-030-57761-2_7.

[3] Bereczky, P., Horpácsi, D., and Thompson, S.J. Machine-checked natural semantics for Core Erlang: exceptions and side effects. In *Proceedings of Erlang 2020*, page 1–13. ACM, 2020. DOI: 10.1145/3406085.3409008.

[4] Bereczky, P., Horpácsi, D., Kőszegi, J., Szeier, S., and Thompson, S. Validating formal semantics by property-based cross-testing. In *Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20)*, pages 150–161. ACM, New York, NY, USA, 2021. DOI: 10.1145/3462172.3462200.

[5] Blazy, S. and Leroy, X. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009. DOI: 10.1007/s10817-009-9148-3.

[6] Carlsson, R., Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.-O., Pettersson, M., and Virding, R. Core Erlang 1.0.3 language specification. Technical report, 2004. URL: https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf.

[7] Cesarini, F. and Thompson, S. *Erlang programming*. O'Reilly Media, Inc., 1st edition, 2009. URL: https://www.oreilly.com/library/view/erlang-programming/9780596803940/.

[8] Core Erlang formalization. URL: https://github.com/harp-project/Core-Erlang-Formalization, 2022. Accessed on 20th of September, 2022.

[9] Core Erlang mini. URL: https://github.com/harp-project/Core-Erlang-mini/releases/tag/v1.6, 2022. Accessed on 20th of September, 2022.

[10] de Bruijn, N.G. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972. DOI: 10.1016/1385-7258(72)90034-0.

[11] Erlang/OTP compiler, version 24.0. URL: https://www.erlang.org/patches/otp-24.0. Accessed on 30th of September 2022.

[12] Erlang documentation. URL: https://www.erlang.org/docs, 2022. Accessed on 20th of September, 2022.

[13] Erlang documentation, Processes. URL: https://www.erlang.org/doc/reference_manual/processes.html, 2022. Accessed on 20th of September, 2022.

[14] Felleisen, M. and Friedman, D.P. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts - III: Proceedings of the IFIP TC 2/WG 2.2 Working Conference on Formal Description of Programming Concepts - III*, pages 193–222, 1987.

[15] Fredlund, L.-Å. *A framework for reasoning about Erlang code*. PhD thesis, Mikroelektronik och informationsteknik, 2001. URL: https://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-3210.

[16] Gumbs, K. The core of Erlang. URL: https://8thlight.com/blog/kofi-gumbs/2017/05/02/core-erlang.html, 2017. Accessed on 17th of March, 2022.

[17] High-Assurance Refactoring Project. URL: https://github.com/harp-project, 2023. Accessed on 27th of March 27th, 2023.

[18] Harrison, J.R. Towards an Isabelle/HOL formalisation of Core Erlang. In *Proceedings of the 16th ACM SIGPLAN International Workshop on Erlang*, Erlang 2017, page 55–63, New York, NY, USA, 2017. Association for Computing Machinery. DOI: 10.1145/3123569.3123576.

[19] Horpácsi, D., Bereczky, P., and Thompson, S. Program equivalence in an untyped, call-by-value functional language with uncurried functions. *Journal of Logical and Algebraic Methods in Programming*, 132:100857, 2023. DOI: 10.1016/j.jlamp.2023.100857.

[20] Kőszegi, J. KErl: Executable semantics for Erlang. *CEUR Workshop Proceedings*, 2046:144–160, 2018. URL: http://ceur-ws.org/Vol-2046/koszegi.pdf.

[21] Lanese, I., Nishida, N., Palacios, A., and Vidal, G. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, 2018. DOI: 10.1016/j.jlamp.2018.06.004.

[22] Lanese, I., Palacios, A., and Vidal, G. Causal-consistent replay reversible semantics for message passing concurrent programs. *Fundamenta Informaticae*, 178(3):229–266, 2021. DOI: 10.3233/FI-2021-2005.

[23] Lanese, I., Sangiorgi, D., and Zavattaro, G. Playing with bisimulation in Erlang. In Boreale, M., Corradini, F., Loreti, M., and Pugliese, R., editors, *Models, Languages, and Tools for Concurrent and Distributed Programming*, pages 71–91. Springer, Cham, 2019. DOI: 10.1007/978-3-030-21485-2_6.

[24] Mason, I. and Talcott, C. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991. DOI: 10.1017/S0956796800000125.

[25] Mosses, P.D. Formal semantics of programming languages: — An overview —. *Electronic Notes in Theoretical Computer Science*, 148(1):41–73, 2006. DOI: 10.1016/j.entcs.2005.12.012, Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004).

[26] Neuhäußer, M. and Noll, T. Abstraction and model checking of Core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):147–163, 2007. DOI: 10.1016/j.entcs.2007.06.013, Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006).

[27] Nishida, N., Palacios, A., and Vidal, G. A reversible semantics for Erlang. In Hermenegildo, M.V. and Lopez-Garcia, P., editors, *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 259–274, Cham, 2017. Springer, Springer International Publishing. DOI: 10.1007/978-3-319-63139-4_15.

[28] Owens, S., Myreen, M.O., Kumar, R., and Tan, Y.K. Functional big-step semantics. In Thiemann, P., editor, *Programming Languages and Systems*, pages 589–615. Springer Berlin Heidelberg, 2016. DOI: 10.1007/978-3-662-49498-1_23.

[29] Pitts, A.M. and Stark, I.D.B. Operational reasoning for functions with local state. *Higher order operational techniques in semantics*, pages 227–273, 1998. DOI: 10.5555/309656.309671.

[30] Schäfer, S., Tebbi, T., and Smolka, G. Autosubst: Reasoning with de Bruijn terms and parallel substitutions. In Urban, Christian and Zhang, Xingyuan, editors, *Interactive Theorem Proving*, pages 359–374, Cham, 2015. Springer International Publishing. DOI: 10.1007/978-3-319-22102-1_24.

[31] Vidal, G. Towards symbolic execution in Erlang. In Voronkov, A. and Virbitskaite, I., editors, *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*, pages 351–360, Berlin, Heidelberg, 2015. Springer, Springer Berlin Heidelberg. DOI: 10.1007/978-3-662-46823-4_28.

[32] Wand, M., Culpepper, R., Giannakopoulos, T., and Cobb, A. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proceedings of the ACM on Programming Languages*, 2(ICFP), 2018. DOI: 10.1145/3236782.