A Reversible Semantics for Erlang[⋆]

Naoki Nishida[1], Adrián Palacios[2,⋆⋆], and Germán Vidal[2]

[1] Graduate School of Information Science, Nagoya University
Furo-cho, Chikusa-ku, 4648603 Nagoya, Japan,
`nishida@is.nagoya-u.ac.jp`
[2] MiST, DSIC, Universitat Politècnica de València
Camino de Vera, s/n, 46022 Valencia, Spain
{`apalacios`, `gvidal`}`@dsic.upv.es`

Abstract. In a reversible language, any forward computation can be undone by a finite sequence of backward steps. Reversible computing has been studied in the context of different programming languages and formalisms, where it has been used for debugging and for enforcing fault-tolerance, among others. In this paper, we consider a subset of Erlang, a concurrent language based on the actor model, and formally introduce a semantics for reversible computation. To the best of our knowledge, this is the first attempt to define a reversible semantics for Erlang.

1 Introduction

Let us consider that the operational semantics of a programming language is specified by a state transition relation R such that $R(s, s')$ holds if the state s' is reachable—in one step—from state s. As it is common practice, we let R^* denote the reflexive and transitive closure of R. Then, we say that a programming language (or formalism) is *reversible* if there exists a constructive algorithm that can be used to, given a computation from state s to state s', in symbols $R^*(s, s')$, obtain the state s from s'. In general, such a property does not hold for most programming languages and formalisms. We refer the interested reader to, e.g., [3, 10, 24, 25] for a high level account of the principles of reversible computation.

The notion of *reversible computation* was first introduced in Landauer's seminal work [14] and, then, further improved by Bennett [2] in order to avoid the generation of "garbage" data. The idea underlying these works is that any programming language or formalism can be made reversible by adding the *history* of

[⋆] This work has been partially supported by the EU (FEDER) and the Spanish *Ministerio de Economía y Competitividad* (MINECO) under grants TIN2013-44742-C4-1-R and TIN2016-76843-C4-1-R, by the *Generalitat Valenciana* under grant PROMETEO-II/2015/013 (SmartLogic), and by COST Action IC1405 on Reversible Computation - extending horizons of computing.

[⋆⋆] Partially supported by the EU (FEDER) and the Spanish programs *Ayudas para contratos predoctorales para la formación de doctores* and *Ayudas a la movilidad predoctoral para la realización de estancias breves en centros de I+D*, MINECO (SEIDI), under FPI grants BES-2014-069749 and EEBB-I-16-11469.

the computation to each state, which is usually called a *Landauer's embedding*. Although carrying the history of a computation might seem infeasible because of its size, there are several successful proposals that are based on this idea. In particular, one can restrict the original language or apply a number of analysis in order to restrict the required information in the history as much as possible, as in, e.g., [18, 19, 22] in the context of a functional language.

In this paper, we aim at introducing a form of reversibility in the context of a programming language that follows the actor model (concurrency based on message passing), and that can be considered as a subset of the concurrent and functional language Erlang [1]. Previous approaches have mainly considered reversibility in—mostly synchronous—concurrent calculi like CCS [7, 8], a general framework for reversibility of algebraic process calculi [20], or the recent approach to reversible *session-based* π-calculus [23]. However, we can only find a few approaches that considered the reversibility of *asynchronous* calculi, e.g., Cardelli and Laneve's reversible structures [5] or the approach based on a rollback construct of [11, 15, 16] (for a higher-order asynchronous π-calculus), [12] (for μKlaim) and [17] (for μOz).

To the best of our knowledge, our work is the first one that considers reversibility in the context of the functional, concurrent, and distributed language Erlang. Here, given a running Erlang system consisting of a pool of interacting processes, possibly distributed in several computers, we aim at allowing a *single* process to undo its actions in a stepwise manner, including the interactions with other processes, following a rollback fashion. In this context, we must ensure *causal consistency* [7], i.e., an action cannot be undone until all the actions that depend on it have already been undone. E.g., if a process spawns another process, we cannot undo this process spawning until all the actions performed by the new process are undone too. This is particularly challenging in our asynchronous and distributed setting since there is no *global* order for the language events. In this paper, we introduce a rollback operator that undoes the actions of a process until a given *checkpoint* is reached.

In this paper, we consider a simple Erlang-like language that can be considered a subset of *Core Erlang* [6]. We present the following contributions: First, we introduce an appropriate standard semantics for the language. In contrast to monolothic previous semantics like that in [4], our semantics is more modular, which simplifies the definition of a reversible extension. In contrast to [21], although we follow some of the ideas in this approach (e.g., the use of a global mailbox), we include the evaluation of expressions and, moreover, our treatment of messages is more deterministic.[3] We then introduce a reversible extension of the standard semantics (basically, a Landauer's embedding) where forward and backward computations are done stepwise. Here, we focus only on the concurrent actions (namely, process spawning, message sending and receiving) and, thus, do not consider the reversibilization of the functional component of the

[3] E.g., in the semantics of [21], at the expression level, the semantics of an expression containing a receive statement is, in principle, infinitely branching, since their formulation allows for an infinite number of possible queues and selected messages.

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

Fig. 1. Language syntax rules

language; rather, we assume that the state of the process—the current expression and its environment—is stored in the history after each execution step. This approach could be improved following, e.g., the approaches from [18, 19, 22]. Finally, we introduce a backward semantics that can be used to undo the actions of a given process, in a rollback fashion, until a checkpoint—introduced by the programmer—is reached. Here, ensuring causal consistency is essential and might propagate the rollback action to other, dependent processes.

2 Language Syntax

In this section, we present the syntax of a first-order concurrent and distributed functional language that follows the actor model. Our language is basically equivalent to a subset of Core Erlang [6].

The syntax of the language can be found in Figure 1. Here, a module is a sequence of function definitions, where each function name f/n (atom/arity) has an associated definition of the form $\mathsf{fun}\ (X_1, \ldots, X_n) \to e$. We consider that a program consists of a single module for simplicity. The body of a function is an *expression*, which can include variables, literals, function names, lists, tuples, calls to built-in functions—mainly arithmetic and relational operators—, function applications, case expressions, let bindings, and receive expressions; furthermore, we also consider the functions spawn, "!" (for sending a message), and $\mathsf{self}()$ that are usually considered built-ins in the Erlang language.

Despite the general syntax in Figure 1, as mentioned before, we only consider first-order expressions. Therefore, the first expression in calls, applications and spawns can only be function names (instead of arbitrary expressions or closures).

In this language, we distinguish expressions, patterns, and values. As mentioned before, expressions can include all constructs of the language. In contrast, *patterns* are built from variables, literals, lists, and tuples. Finally, *values* are built from literals, lists, and tuples, i.e., they are *ground*—without variables—patterns. Expressions are denoted by $e, e', e_1, e_2, \ldots$, patterns by $p, p', p_1, p_2, \ldots$

and values by $v, v', v_1, v_2, \ldots$ As it is common practice, a *substitution* θ is a mapping from variables to expressions such that $\mathcal{D}om(\theta) = \{X \in \mathit{Var} \mid X \neq \theta(X)\}$ is its domain. Substitutions are usually denoted by sets of mappings like, e.g., $\{X_1 \mapsto v_1, \ldots, X_n \mapsto v_n\}$. Substitutions are extended to morphisms from expressions to expressions in the natural way. The identity substitution is denoted by *id*. Composition of substitutions is denoted by juxtaposition, i.e., $\theta\theta'$ denotes a substitution θ'' such that $\theta''(X) = \theta'(\theta(X))$ for all $X \in \mathit{Var}$. Also, we denote by $\theta[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n]$ the *update* of θ with the mapping $X_1 \mapsto v_1, \ldots, X_n \mapsto v_n$, i.e., it denotes a new substitution θ' such that $\theta'(X) = v_i$ if $X = X_i$, for some $i \in \{1, \ldots, n\}$, and $\theta'(X) = \theta(X)$ otherwise.

In a case expression "$\mathsf{case}\ e\ \mathsf{of}\ p_1\ \mathsf{when}\ e_1 \to e'_1;\ \ldots;\ p_n\ \mathsf{when}\ e_n \to e'_n\ \ \mathsf{end}$", we first evaluate e to a value, say v; then, we should find (if any) the first clause $p_i\ \mathsf{when}\ e_i \to e'_i$ such that v matches p_i (i.e., there exists a substitution σ for the variables of p_i such that $v = p_i\sigma$) and $e_i\sigma$—the *guard*—reduces to *true*; then, the case expression reduces to $e'_i\sigma$. Note that guards can only contain calls to built-in functions (typically, arithmetic and relational operators).

As for the concurrent features of the language, we consider that a *system* is of a pool of processes that can only interact through message sending and receiving (i.e., there is no shared memory). Each process has an associated *pid* (process identifier), which is unique in a system. For clarity, we often denote pids with roman letters $\mathrm{p}, \mathrm{p}', \mathrm{p}_1, \ldots$, though they are considered values in our language (i.e., atoms). By abuse of notation, when no confusion can arise, we refer to a process with its pid.

An expression of the form $\mathsf{spawn}(f/n, [e_1, \ldots, e_n])$ has, as a *side effect*, the creation of a new process with a fresh pid p which is initialized with the expression $\mathsf{apply}\ f/n\ (e_1, \ldots, e_n)$; the expression $\mathsf{spawn}(f/n, [e_1, \ldots, e_n])$ itself evaluates to the new pid p. The function $\mathsf{self}()$ just returns the pid of the current process. An expression of the form $\mathrm{p} \,!\, v$ evaluates to the value v and, as a side effect, stores the value v—the *message*—in the queue or *mailbox* of process p.

Finally, an expression "$\mathsf{receive}\ p_1\ \mathsf{when}\ e_1 \to e'_1; \ldots; p_n\ \mathsf{when}\ e_n \to e'_n\ \ \mathsf{end}$" traverses the messages in the process' queue until one of them matches a branch in the receive statement; i.e., it should find the *first* message v in the process' queue (if any) such that $\mathsf{case}\ v\ \mathsf{of}\ p_1\ \mathsf{when}\ e_1 \to e'_1; \ldots; p_n\ \mathsf{when}\ e_n \to e'_n\ \mathsf{end}$ can be reduced; then, the receive expression evaluates to the same expression to which the above case expression would be evaluated, with the additional side effect of deleting the message v from the process' queue. If there is no matching message in the queue, the process suspends its execution until a matching message arrives.

Example 1. Consider the program shown in Figure 2, where the symbol "_" is used to denote an *anonymous* variable, i.e., a variable whose name is not relevant. The computation starts with "$\mathsf{apply}\ \mathit{main}/0\ ()$." Then, this process, say $P1$, spawns two new processes, say $P2$ and $P3$, and then sends the message "*world*" to process $P3$ and the message $\{P3, \mathit{hello}\}$ to process $P2$, which then resends "*hello*" to $P3$. In our language, there is no guarantee regarding which message arrives first to $P3$, i.e., "$\mathsf{apply}\ \mathit{main}/0\ ()$" can evaluate nondeterministically to

```
main/0 = fun () → let P2 = spawn(echo/0, [ ])
                  in let P3 = spawn(target/0, [ ])
                  in let _ = P3 ! world
                  in let P2 ! {P3, hello}

target/0 = fun () → receive
                        A → receive                echo/0 = fun () → receive
                               B → {A, B}                                {P, M} → P ! M
                            end                                        end
                    end
```

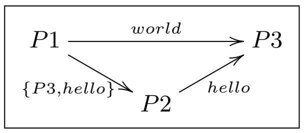

Fig. 2. A simple concurrent program

either $\{hello, world\}$ or $\{world, hello\}$. This is coherent with the semantics of Erlang, where the only guarantee is that if two messages are sent from process p to process p′ and both are delivered, then the order of these messages is kept.

3 The Language Semantics

In order to set precisely the framework for our proposal, in this section we formalize the semantics of the considered language.

Definition 1 (process). *A process is denoted by a tuple of the form* $\langle \mathrm{p}, (\theta, e), q\rangle$ *where* p *is the pid of the process,* (θ, e) *is the control of the state—which consists of an environment (a substitution) and an expression to be evaluated—, and* q *is the process' mailbox, a FIFO queue with the sequence of messages that have been sent to the process.*

A running *system* is then a pool of processes, which is formally defined as follows:

Definition 2 (system). *A system is denoted by* $\Gamma; \Pi$*, where* Γ *is a global mailbox of the system (see below) and* Π *is a pool of processes, denoted by an expression of the form* $\langle \mathrm{p}_1, (\theta_1, e_1), q_1\rangle \,\&\cdots\&\, \langle \mathrm{p}_n, (\theta_n, e_n), q_n\rangle$*, where "&" is an associative and commutative operator. We typically denote a system by an expression of the form* $\Gamma; \langle \mathrm{p}, (\theta, e), q\rangle \& \Pi$ *to point out that* $\langle \mathrm{p}, (\theta, e), q\rangle$ *is an arbitrary process of the pool (thanks to the fact that "&" is associative and commutative).*

The role of Γ (which is similar to the "ether" in [21]) will be clarified later, but it is essential to guarantee that all admissible interleavings can be modelled with the semantics. Here, we define Γ as a set of FIFO queues among all (non-necessarily different) pids, i.e., Γ is made of elements of the form $(\mathrm{p}, \mathrm{q}, [v_1, \ldots, v_n])$, where p, q are (not necessarily different) pids and $[v_1, \ldots, v_n]$ is a (possibly empty) ordered list of messages such that v_1 is the oldest message and v_n is the most recent one. For simplicity, we assume that Γ is initialized as follows: $\{(\mathrm{p}, \mathrm{q}, [\,]) \mid \mathrm{p}, \mathrm{q} \text{ are pids}\}$. Then, we use the following notation: $\Gamma \cup (\mathrm{p}, \mathrm{q}, v)$ denotes $\Gamma \setminus \{(\mathrm{p}, \mathrm{q}, vs)\} \cup \{(\mathrm{p}, \mathrm{q}, vs{+\!\!+}[v])\}$, while $\Gamma \setminus (\mathrm{p}, \mathrm{q}, v)$ denotes $\Gamma \setminus \{(\mathrm{p}, \mathrm{q}, v : vs)\} \cup \{(\mathrm{p}, \mathrm{q}, vs)\}$, where $+\!\!+$ is the list concatenation operator.

$$(\mathit{Var})\ \frac{}{\theta, X \xrightarrow{\tau} \theta, \theta(X)} \qquad (\mathit{Tuple})\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e'_i \quad i \in \{1,\ldots,n\}}{\theta, \{e_1,\ldots,e_n\} \xrightarrow{\ell} \theta', \{\overline{e_{1,i-1}}, e'_i, \overline{e_{i+1,n}}\}}$$

$$(\mathit{List1})\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e'_1}{\theta, [e_1|e_2] \xrightarrow{\ell} \theta', [e'_1|e_2]} \qquad (\mathit{List2})\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e'_2}{\theta, [e_1|e_2] \xrightarrow{\ell} \theta', [e_1|e'_2]}$$

$$(\mathit{Let1})\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e'_1}{\theta, \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \xrightarrow{\ell} \theta', \mathsf{let}\ X = e'_1\ \mathsf{in}\ e_2} \quad (\mathit{Let2})\ \frac{}{\theta, \mathsf{let}\ X = v\ \mathsf{in}\ e \xrightarrow{\tau} \theta[X \mapsto v], e}$$

$$(\mathit{Case1})\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\begin{array}{l}\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \\ \xrightarrow{\ell} \theta', \mathsf{case}\ e'\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end}\end{array}} \quad (\mathit{Case2})\ \frac{\mathsf{match}(v, cl_1, \ldots, cl_n) = \langle \theta_i, e_i \rangle}{\theta, \mathsf{case}\ v\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\tau} \theta\theta_i, e_i}$$

$$(\mathit{Call1})\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e'_i \quad i \in \{1,\ldots,n\}}{\theta, \mathsf{call}\ op\ (\overline{e_n}) \xrightarrow{\ell} \theta', \mathsf{call}\ op\ (\overline{e_{1,i-1}}, e'_i, \overline{e_{i+1,n}})} \quad (\mathit{Call2})\ \frac{\mathsf{eval}(op, v_1, \ldots, v_n) = v}{\theta, \mathsf{call}\ op\ (v_1, \ldots, v_n) \xrightarrow{\tau} \theta, v}$$

$$(\mathit{Apply1})\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e'_i \quad i \in \{1,\ldots,n\}}{\theta, \mathsf{apply}\ a/n\ (\overline{e_n}) \xrightarrow{\ell} \theta', \mathsf{apply}\ a/n\ (\overline{e_{1,i-1}}, e'_i, \overline{e_{i+1,n}})}$$

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

Fig. 3. Standard semantics: evaluation of sequential expressions

In the following, we denote by $\overline{o_n}$ the sequence of syntactic objects $o_1, \ldots, o_n$ for some n. We also write $\overline{o_{i,j}}$ for the sequence $o_i, \ldots, o_j$ when $i \leq j$ (and the empty sequence otherwise). We write $\overline{o}$ when the number of elements is not relevant.

The semantics is defined by means of two transition relations: $\longrightarrow$ for expressions and $\longmapsto$ for systems. Let us first consider the labelled transition relation

$$\longrightarrow\ : (\mathit{Env}, \mathit{Exp}) \times \mathit{Label} \times (\mathit{Env}, \mathit{Exp})$$

where Env and Exp are the domains of environments (i.e., substitutions) and expressions, respectively, and Label denotes an element of the set

$$\{\tau, \mathsf{send}(v_1, v_2), \mathsf{rec}(y, \overline{cl_n}), \mathsf{spawn}(y, a/n, [\overline{e_n}]), \mathsf{self}(y)\}$$

whose meaning will be explained below. For clarity, we divide the transition rules of the semantics for expressions in two sets, depicted in Figures 3 and 4 for sequential and concurrent expressions, respectively.

Most of the rules are self-explanatory. In the following, we only discuss some subtle or complex issues. In principle, the transitions are labelled either with τ (a sequential expression) or with a label that identifies a concurrent action. Labels are used in the system rules (Figure 5) to perform the associated side effects.

$$
\begin{array}{ll}
(\mathit{Send1}) & \dfrac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1' \,!\, e_2} \qquad \dfrac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1 \,!\, e_2'} \\[2ex]
(\mathit{Send2}) & \dfrac{}{\theta, v_1 \,!\, v_2 \xrightarrow{\mathsf{send}(v_1, v_2)} \theta, v_2} \\[2ex]
(\mathit{Receive}) & \dfrac{}{\theta, \mathsf{receive}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(y, \overline{cl_n})} \theta, y} \\[2ex]
(\mathit{Spawn}) & \dfrac{}{\theta, \mathsf{spawn}(a/n, [e_1, \ldots, e_n]) \xrightarrow{\mathsf{spawn}(y, a/n, [\overline{e_n}])} \theta, y} \\[2ex]
(\mathit{Self}) & \dfrac{}{\theta, \mathsf{self}() \xrightarrow{\mathsf{self}(y)} \theta, y}
\end{array}
$$

Fig. 4. Standard semantics: evaluation of concurrent expressions

In some of the rules (e.g., for evaluating tuples, lists, etc) we consider for simplicity that elements are evaluated in a non-deterministic way. In an actual programming language the order of evaluation of the arguments in the tuple or list is usually fixed. E.g., in Erlang, reduction takes place from left to right.

For case evaluation, we assume an auxiliary function match which selects the first clause, $cl_i = (p_i\ \mathsf{when}\ e_i' \to e_i)$, such that v matches p_i, i.e., $v = \theta_i(p_i)$, and the guard holds, i.e., $\theta\theta_i, e_i' \longrightarrow^* \theta', \mathit{true}$. Note that, for simplicity, we do not consider here the case in which the argument v matches no clause.

Function calls can either be defined in the program (apply) or be a built-in (call). In the latter case, they are evaluated using the auxiliary function eval. In rule *Apply2*, we consider that the mapping μ stores all function definitions, i.e., it maps every function name a/n to its definition $\mathsf{fun}\ (X_1, \ldots, X_n) \to e$ in the program. As for the applications, note that we only consider first-order functions. In order to extend our semantics to also consider higher-order functions, one should reduce the function name to a *closure* of the form $(\theta', \mathsf{fun}\ (X_1, \ldots, X_n) \to e)$ and, then, reduce e in the environment $\theta'[X_1 \mapsto v_1, \ldots, X_n \mapsto v_n]$. We skip this extension since it is orthogonal to our approach.

Let us now consider the evaluation of concurrent expressions that produce some side effect (Figure 4). Here, we can distinguish two kinds of rules. On the one hand, we have the rules for "!", *Send1* and *Send2*. In this case, we know *locally* what the expression should be reduced to (i.e., v_2 in rule *Send2*). For the remaining rules, this is not known locally and, thus, we return a fresh distinguished symbol, $y \notin \mathit{Var}$ (by abuse, y is dealt with as a variable), so that the system rules will eventually bind y to its correct value. This *trick* allows us to keep the rules for expressions and systems separated (i.e., the semantics shown in Figures 3 and 4 is mostly independent from the rules in Figure 5), in contrast to other calculi, e.g., [4], where they are combined into a single transition relation.

Let us finally consider the system rules, which are depicted in Figure 5. In most of the rules, we consider an arbitrary system of the form $\Gamma; \langle \mathrm{p}, (\theta, e), q\rangle \& \Pi$, where Γ is the global mailbox and $\langle \mathrm{p}, (\theta, e), q\rangle \& \Pi$ is a pool of process that contains at least one process $\langle \mathrm{p}, (\theta, e), q\rangle$.

$$(\mathit{Exp})\qquad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle \mathrm{p}, (\theta, e), q\rangle \& \Pi \longmapsto \Gamma; \langle \mathrm{p}, (\theta', e'), q\rangle \& \Pi}$$

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

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

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

$$(\mathit{Self})\qquad \frac{\theta, e \xrightarrow{\mathsf{self}(y)} \theta', e'}{\Gamma; \langle \mathrm{p}, (\theta, e), q\rangle \& \Pi \longmapsto \Gamma; \langle \mathrm{p}, (\theta', e'\{y \mapsto \mathrm{p}\}), q\rangle \& \Pi}$$

$$(\mathit{Sched})\qquad \frac{\alpha(\Gamma) = (\mathrm{p}', \mathrm{p}) \qquad \Pi = \langle \mathrm{p}, (\theta, e), q\rangle \& \Pi'}{\Gamma; \Pi \longmapsto \Gamma \setminus \{(\mathrm{p}', \mathrm{p}, v)\}; \langle \mathrm{p}, (\theta, e), v : q\rangle \& \Pi'}$$

Fig. 5. Standard semantics: system rules

Note that, in rule *Send*, we add the triple $(\mathrm{p}, \mathrm{p}'', v)$ to Γ instead of adding it to the queue of process p''. This is necessary to ensure that all possible non-deterministic results can be obtained (as discussed in Example 1). Observe that e' is usually different from v since e may have different nested operators. E.g., if e has the form "case $\mathrm{p}\,!\,v$ of $\{\ldots\}$," then e' will be "case v of $\{\ldots\}$" with label $\mathsf{send}(\mathrm{p}, v)$.

In rule *Receive*, we use the auxiliary function matchrec to evaluate a receive expression. The main difference with match is that matchrec also takes a queue q and returns the modified queue q'. Then, the distinguished variable y is bound to the expression in the selected clause, e_i, and the environment is extended with the matching substitution. If no message in the queue q matches any clause, then the rule is not applicable and the selected process cannot be reduced (i.e., it suspends).

With the rules presented so far, any system will soon reach a state in which no reduction can be performed, since messages are stored in the global mailbox, but they are not dispatched to the queues of the processes. This is precisely the task of the scheduler, which is modelled by rule *Sched*. The rule is non-deterministic, so any scheduling policy can be modelled by the semantics. A message is selected from the list of messages by the auxiliary function α, which can select any arbitrary pair of (non-necessarily different) pids $(\mathrm{p}', \mathrm{p})$. Note that we take the oldest message in the queue—the first one in the list—, which is necessary to ensure that "the messages sent—directly—between two given processes arrive in the same order they were sent", as mentioned in the previous section.

Example 2. Let us consider the program shown in Figure 6 and a possible execution trace. This trace is modelled by our semantics. For clarity, we only show

```
main/0   = fun () → let S = spawn(server/0, [ ])
                    in let _ = spawn(client/1, [S])
                    in apply client/1 (S)
server/0 = fun () → receive
                    {P, M} → let _ = P ! ack
                                 in apply server/0 ()
                    end
client/1 = fun (S) → let _ = S ! {self(), req}
                     in receive
                          ack → ok
                     end
```

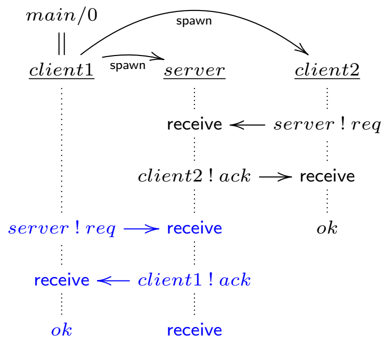

Fig. 6. A simple client-server

$$
\begin{array}{lrl}
 & [\,];& \langle \mathrm{c1}, (id, \underline{\mathsf{apply}\ main/0\ ()}), [\,]\rangle \\
\longmapsto & & \ldots \\
\longmapsto & [\,];& \langle \mathrm{c1}, (\sigma, \mathsf{let}\ \_ = \underline{S\ !\ v_2}\ \mathsf{in} \ldots), [\,]\rangle \ \&\ \langle \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [(\mathrm{c1}, \mathrm{s}, [v_2])];& \langle \mathrm{c1}, (\sigma, \underline{\mathsf{let}\ \_ = v_2\ \mathsf{in} \ldots}), [\,]\rangle \ \&\ \langle \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [\underline{(\mathrm{c1}, \mathrm{s}, [v_2])}];& \langle \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle \ \&\ \langle \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [\,];& \langle \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle \ \&\ \langle \mathrm{s}, (id, \underline{\mathsf{receive}\ \{P, M\} \to \ldots}), [v_2]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [\,];& \langle \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle \ \&\ \langle \mathrm{s}, (\theta_2, \mathsf{let}\ \_ = \underline{P\ !\ ack}\ \mathsf{in}\ \ \ldots), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [(\mathrm{s}, \mathrm{c1}, [ack])];& \langle \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle \ \&\ \langle \mathrm{s}, (\theta_2, \underline{\mathsf{let}\ \_ = ack\ \mathsf{in}\ \ \ldots}), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [(\mathrm{s}, \mathrm{c1}, [ack])];& \langle \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle \ \&\ \langle \mathrm{s}, (\theta_2, \underline{\mathsf{apply}\ server/0\ ()}), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [\underline{(\mathrm{s}, \mathrm{c1}, [ack])}];& \langle \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle \ \&\ \langle \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [\,];& \langle \mathrm{c1}, (\sigma, \underline{\mathsf{receive}\ ack \to \ldots}), [ack]\rangle \ \&\ \langle \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\longmapsto & [\,];& \langle \mathrm{c1}, (\sigma, ok), [\,]\rangle \ \&\ \langle \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \& \\
 & & \langle \mathrm{c2}, (\sigma, ok), [\,]\rangle
\end{array}
$$

Fig. 7. A trace with $\sigma = \{S \mapsto \mathrm{s}\}$, $\theta_2 = \{P \mapsto \mathrm{c1}, M \mapsto req\}$, and $v_2 = \{\mathrm{c1}, req\}$.

in Figure 7 the transition steps that correspond to the last two messages sent between *client1* and *server*.

4 Reversible Semantics

In this section, we introduce a reversible extension of the semantics defined so far. Moreover, thanks to our modular design, the semantics for the language expressions needs not be changed.

To be precise, in this section we introduce two transition relations: $\rightharpoonup$ and $\leftharpoondown$. The first relation, $\rightharpoonup$, is a conservative extension of the standard semantics $\longmapsto$ (Figure 5) to also include some additional information in the states, following a typical Landauer's embedding. We refer to $\rightharpoonup$ as the *forward* reversible semantics (or simply the forward semantics). In contrast, the second relation, $\leftharpoondown$, proceeds in the backwards direction, "undoing" the actions of a single process step by step (and, by causal consistency, possibly propagating it to other processes). We refer to $\leftharpoondown$ as the backward (reversible) semantics. We denote the union $\rightharpoonup \cup \leftharpoondown$ by $\rightleftharpoons$. Then, in a computation modelled with $\rightleftharpoons$ the system mainly evolves forwards, except for some processes that can run backwards in order to undo some particular actions (and, afterwards, will run forwards again).

Here, we will introduce a (non-deterministic) "undo" operation which has some similarities to, e.g., the rollback operator of [11]. In order to delimit the scope of this operation (i.e., to determine when to stop undoing the actions of a process), we allow the programmer to introduce *checkpoints* in a program. Syntactically, they are denoted with the built-in function check, which takes an identifier $\mathtt{t}$ as an argument, which is supposed to be unique in the program. Given an expression, *expr*, we can introduce a checkpoint by replacing *expr* with "$\mathsf{let}\ x = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ expr$". A call of the form $\mathsf{check}(\mathtt{t})$ just returns $\mathtt{t}$ (see below). In the following, we consider that the rules to evaluate the language expressions (Figures 3 and 4) are extended with the following rule:

$$(\mathit{Check})\quad \frac{}{\theta, \mathsf{check}(\mathtt{t}) \xrightarrow{\mathsf{check}(\mathtt{t})} \theta, \mathtt{t}}$$

In the next section, we will see that the only effect of a call to function check is to add a checkpoint to the trace of a given process.

4.1 Forward Semantics

First, we introduce the forward (reversible) semantics. Since the expression rules are the same (except for the additional rule for check mentioned above), we will only introduce the reversible system rules, which are shown in Figure 8. Processes now include a memory (or *trace*) π that records the intermediate states of a process. Here, we use terms headed by constructors τ, check, send, rec, spawn, self, and sched to record the steps given with the forward semantics. Note that we could optimize the information stored in these terms by following a strategy similar to that in [18, 19, 22] for the reversibility of functional expressions, but this is orthogonal to our purpose in this paper, so we focus only on the concurrent actions.

The rules are mostly self-explanatory. Checkpoints introduced by the programmer, of the form $\mathsf{check}(\theta, e, \mathtt{t})$, represent a *safe* point in the program execution. Rollback operations and checkpoints introduced by the programmer lay the ground for defining *safe* sessions whose actions can be undone if, e.g., an exception occurs before they are completed. Besides these checkpoints, we also consider checkpoints associated to receiving a message, denoted by $\mathsf{sched}(\mathrm{p}, \mathrm{p}', v)$, and spawning a process (empty trace). These checkpoints are internal and only used to ensure causal consistency.

Example 3. Consider again the program shown in Figure 6, where the function *client*/1 is now defined as follows:

$$\begin{array}{l} \mathit{client}/1 = \mathsf{fun}\ (S) \rightarrow \mathsf{let}\ \_ = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ \mathsf{let}\ \_ = S\,!\,\{\mathsf{self}(), \mathit{req}\} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathsf{in}\ \mathsf{receive}\ \mathsf{ack} \rightarrow \mathsf{ok}\ \mathsf{end} \end{array}$$

$$(\mathit{Internal}) \quad \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle \pi, p, (\theta, e), q\rangle \& \Pi \rightharpoonup \Gamma; \langle \tau(\theta, e) : \pi, p, (\theta', e'), q\rangle \& \Pi}$$

$$(\mathit{Check}) \quad \frac{\theta, e \xrightarrow{\mathsf{check}(\mathtt{t})} \theta', e'}{\Gamma; \langle \pi, p, (\theta, e), q\rangle \& \Pi \rightharpoonup \Gamma; \langle \mathsf{check}(\theta, e, \mathtt{t}) : \pi, p, (\theta', e'), q\rangle \& \Pi}$$

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

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

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

$$(\mathit{Self}) \quad \frac{\theta, e \xrightarrow{\mathsf{self}(y)} \theta', e'}{\Gamma; \langle \pi, p, (\theta, e), q\rangle \& \Pi \rightharpoonup \Gamma; \langle \mathsf{self}(\theta, e) : \pi, p, (\theta', e'\{y \mapsto p\}), q\rangle \& \Pi}$$

$$(\mathit{Sched}) \quad \frac{\alpha(\Gamma) = (p', p) \qquad \Pi = \langle \pi, p, (\theta, e), q\rangle \& \Pi'}{\Gamma; \Pi \rightharpoonup \Gamma \setminus (p', p, v); \langle \mathsf{sched}(p', p, v) : \pi, p, (\theta, e), v : q\rangle \& \Pi'}$$

Fig. 8. Reversible semantics: system rules

and the execution trace shown in Figure 7. The corresponding forward (reversible) computation is shown in Figure 9.

In the following, we let $s_1, s_2, \ldots$ denote systems of the standard semantics (Fig. 5) and $rs_1, rs_2, \ldots$ denote systems of the instrumented reversible semantics (Fig. 8). Here, we denote by $\overline{rs}$ the system that results from rs by removing the traces of the processes; formally, $\overline{\Gamma; \langle \pi_1, \mathrm{p}_1, (\theta_1, e_2), q_1\rangle \& \ldots \& \langle \pi_n, \mathrm{p_n}, (\theta_n, e_n), q_n\rangle}$ $= \Gamma; \langle \mathrm{p}_1, (\theta_1, e_2), q_1\rangle \& \ldots \& \langle \mathrm{p_n}, (\theta_n, e_n), q_n\rangle$. The following result states that $\rightharpoonup$ is indeed a conservative extension of the standard semantics $\longmapsto$:

Theorem 1. *Let $\mathcal{P}$ be a program without occurrences of "*check*". Let s_1 be a system of the standard semantics and rs_1 a system of the reversible semantics with $\overline{rs_1} = s_1$. Then, $s_1 \longmapsto^* s_2$ iff $rs_1 \rightharpoonup^* rs_2$ and $\overline{rs_2} = s_2$.*

4.2 Backward Semantics

In the following, we denote a process running backwards with $\lfloor proc \rfloor_{\#}$, where $\#$ is a rollback label that refers to the checkpoint that the backward computation of $proc$ has to go through before resuming its forward computation. For instance, a process of the form $\lfloor proc \rfloor_{\#_{\mathsf{ch}}^{\mathtt{t}}}$ should go backwards until a checkpoint $\mathsf{check}(\theta, e, \mathtt{t})$ is found in its trace, a process $\lfloor proc \rfloor_{\#_{\mathsf{sch}}^{\mathrm{p}, v}}$ should go backwards until an event of the form $\mathsf{sched}(\mathrm{p}, \mathrm{p}', v)$ is found in its trace, and a process $\lfloor proc \rfloor_{\#_{\mathsf{sp}}}$

$$
\begin{array}{lrl}
 & [\,]; & \langle [\,], \mathrm{c1}, (id, \underline{\mathsf{apply}\ main/0\ ()}), [\,]\rangle \\
\rightharpoonup & & \ldots \\
\rightharpoonup & [\,]; & \langle \pi_i, \mathrm{c1}, (\sigma, \mathsf{let}\ _ = \underline{\mathsf{check}(\mathtt{t})}\ \mathsf{in} \ldots), [\,]\rangle \ \&\ \langle \pi_i', \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \& \\
 & & \langle \pi_i'', \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\rightharpoonup & [\,]; & \langle \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \underline{\mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots}), [\,]\rangle\ \& \\
 & & \langle \pi_i', \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \&\ \langle \pi_i'', \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\rightharpoonup & [\,]; & \langle \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \\
 & & \quad \mathrm{c1}, (\sigma, \mathsf{let}\ _ = \underline{S\ !\ v_2}\ \mathsf{in} \ldots), [\,]\rangle\ \& \\
 & & \langle \pi_i', \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \&\ \langle \pi_i'', \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\rightharpoonup & [(\mathrm{c1}, \mathrm{s}, [v_2])]; & \langle \mathsf{send}(\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots, \mathrm{c}_1, \mathrm{s}, v_2) : \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
 & & \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \underline{\mathsf{let}\ _ = v_2\ \mathsf{in} \ldots}), [\,]\rangle\ \& \\
 & & \langle \pi_i', \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \&\ \langle \pi_i'', \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\rightharpoonup & [\underline{(\mathrm{c1}, \mathrm{s}, [v_2])}]; & \langle \tau(\sigma, \mathsf{let}\ _ = v_2\ \mathsf{in} \ldots) : \mathsf{send}(\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots, \mathrm{c}_1, \mathrm{s}, v_2) : \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
 & & \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle\ \& \\
 & & \langle \pi_i', \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \&\ \langle \pi_i'', \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\rightharpoonup & [\,]; & \langle \tau(\sigma, \mathsf{let}\ _ = v_2\ \mathsf{in} \ldots) : \mathsf{send}(\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots, \mathrm{c}_1, \mathrm{s}, v_2) : \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
 & & \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle\ \& \\
 & & \langle \mathsf{sched}(\mathrm{c1}, \mathrm{s}, v_2) : \pi_i', \mathrm{s}, (id, \underline{\mathsf{receive}\ \{P, M\} \to \ldots}), [v_2]\rangle\ \& \\
 & & \langle \pi_i'', \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\rightharpoonup & [\,]; & \langle \tau(\sigma, \mathsf{let}\ _ = v_2\ \mathsf{in} \ldots) : \mathsf{send}(\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots, \mathrm{c}_1, \mathrm{s}, v_2) : \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
 & & \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle\ \& \\
 & & \langle \mathsf{rec}(id, \mathsf{receive}\ \{P, M\} \to \ldots, [v_2]) : \mathsf{sched}(\mathrm{c1}, \mathrm{s}, v_2) : \pi_i', \\
 & & \quad \mathrm{s}, (\theta_2, \mathsf{let}\ _ = \underline{P\ !\ ack}\ \mathsf{in}\ \ldots), [\,]\rangle\ \&\ \langle \pi_i'', \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\rightharpoonup & & \ldots
\end{array}
$$

Fig. 9. A (partial) trace with the forward reversible semantics, where $\sigma = \{S \mapsto \mathrm{s}\}$, $\theta_2 = \{P \mapsto \mathrm{c1}, M \mapsto req\}$, and $v_2 = \{\mathrm{c1}, req\}$

should go backwards until its initial state is reached (i.e., it should be completely undone).

In order to introduce a rollback operator (e.g., when a process crashes or some undesired situation occurs), we fire the following rule:

$$(\mathit{Undo}) \qquad \Gamma; \langle \pi, p, (\theta, e), q\rangle \ \&\ \Pi \leftharpoondown \Gamma; \lfloor \langle \pi, p, (\theta, e), q\rangle \rfloor_{\#^{\mathtt{t}}_{\mathsf{ch}}} \ \&\ \Pi$$

for some checkpoint identifier $\mathtt{t}$.

Let us now discuss the rules for performing backward computations, which are shown in Figure 10, where $\#$ denotes an arbitrary rollback label. In general, all rules restore the control (and sometimes also the queue) of a process.

Rule $\overline{\mathit{Check1}}$ resumes the forward computation of a process rolling back with $\#^{\mathtt{t}}_{\mathsf{ch}}$ when we reach a term of the form $\mathsf{check}(\ldots, \mathtt{t})$ in the trace. When the label is different (i.e., $\#_{\mathsf{sp}}$ or $\#^{p,v}_{\mathsf{sch}}$) or it is a label $\#^{\mathtt{t}'}_{\mathsf{ch}}$ with $\mathtt{t} \neq \mathtt{t}'$, then rule $\overline{\mathit{Check2}}$ just removes the $\mathsf{check}(\ldots)$ from the trace.

In order to undo the sending of a message, rule $\overline{\mathit{Send1}}$ removes a message from Γ when the message has not been delivered yet (using rule *Sched*). Here, we use the operator "$\backslash\backslash$" to denote the removal of the last (newest) message between two given processes—in contrast to "$\backslash$", which always removes the oldest one. Otherwise, i.e., when the message has already been delivered, rule $\overline{\mathit{Send2}}$ *freezes* the process,[4] denoted with $\lceil \ldots \rceil_{\#}$, and applies a rollback operator to the receiver labelled with $\#^{p,v}_{\mathsf{sch}}$. This will cause the receiver process to undo all the

[4] Note that we use the notation $\lceil \ldots \rceil$ to explicitly denote that a process is frozen, though it is not really necessary since no transition rule would be applicable anyway.

$$(\overline{\mathit{Internal}}) \quad \Gamma; \lfloor\langle \tau(\theta,e)\!:\!\pi, p, (\theta',e'), q\rangle\rfloor_{\#} \;\&\; \Pi \leftharpoondown \Gamma; \lfloor\langle \pi, p, (\theta,e), q\rangle\rfloor_{\#} \;\&\; \Pi$$

$$(\overline{\mathit{Check1}}) \quad \Gamma; \lfloor\langle \mathsf{check}(\theta,e,\mathtt{t})\!:\!\pi, p, (\theta',e'), q\rangle\rfloor_{\#^{\mathtt{t}}_{\mathsf{ch}}} \;\&\; \Pi \leftharpoondown \Gamma; \langle \pi, p, (\theta,e), q\rangle \;\&\; \Pi$$

$$(\overline{\mathit{Check2}}) \quad \Gamma; \lfloor\langle \mathsf{check}(\theta,e,\mathtt{t})\!:\!\pi, p, (\theta',e'), q\rangle\rfloor_{\#} \;\&\; \Pi \leftharpoondown \Gamma; \lfloor\langle \pi, p, (\theta',e'), q\rangle\rfloor_{\#} \;\&\; \Pi \quad \text{if } \# \neq \#^{\mathtt{t}}_{\mathsf{ch}}$$

$$(\overline{\mathit{Send1}}) \quad \begin{array}{l} \Gamma; \lfloor\langle \mathsf{send}(\theta,e,p,p',v)\!:\!\pi, p, (\theta',e'), q\rangle\rfloor_{\#} \;\&\; \Pi \leftharpoondown \Gamma'; \lfloor\langle \pi, p, (\theta,e), q\rangle\rfloor_{\#} \;\&\; \Pi \\ \qquad \text{if } (p,p',v) \text{ occurs in } \Gamma, \text{ with } \Gamma' = \Gamma \setminus\!\!\setminus (p,p',v) \end{array}$$

$$(\overline{\mathit{Send2}}) \quad \begin{array}{l} \Gamma; \lfloor\langle \mathsf{send}(\theta,e,p,p'',v)\!:\!\pi, p, (\theta',e'), q\rangle\rfloor_{\#} \;\&\; \langle \pi'', p'', (\theta'',e''), q''\rangle \;\&\; \Pi \\ \quad \leftharpoondown \Gamma; \lceil\langle \mathsf{send}(\theta,e,p,p'',v)\!:\!\pi, p, (\theta,e), q\rangle\rceil_{\#} \;\&\; \lfloor\langle \pi'', p'', (\theta'',e''), q''\rangle\rfloor_{\#^{p,v}_{\mathsf{sch}}} \;\&\; \Pi \\ \qquad \text{if } (p,p'',v) \text{ does not occur in } \Gamma \end{array}$$

$$(\overline{\mathit{Send3}}) \quad \begin{array}{l} \Gamma; \lceil\langle \mathsf{send}(\theta,e,p,p'',v)\!:\!\pi, p, (\theta,e), q\rangle\rceil_{\#} \;\&\; \lfloor\langle \mathsf{sched}(p,p'',v)\!:\!\pi'', p'', (\theta'',e''), v:q''\rangle\rfloor_{\#^{p,v}_{\mathsf{sch}}} \;\&\; \Pi \\ \quad \leftharpoondown \Gamma; \lfloor\langle \pi, p, (\theta,e), q\rangle\rfloor_{\#} \;\&\; \langle \pi'', p'', (\theta'',e''), q''\rangle \;\&\; \Pi \end{array}$$

$$(\overline{\mathit{Receive}}) \quad \Gamma; \lfloor\langle \mathsf{rec}(\theta,e,q)\!:\!\pi, p, (\theta',e'), q'\rangle\rfloor_{\#} \;\&\; \Pi \leftharpoondown \Gamma; \lfloor\langle \pi, p, (\theta,e), q\rangle\rfloor_{\#} \;\&\; \Pi$$

$$(\overline{\mathit{Spawn1}}) \quad \begin{array}{l} \Gamma; \lfloor\langle \mathsf{spawn}(\theta,e,p'')\!:\!\pi, p, (\theta',e'), q\rangle\rfloor_{\#} \;\&\; \langle \pi'', p'', (\theta'',e''), q''\rangle \;\&\; \Pi \\ \quad \leftharpoondown \Gamma; \lceil\langle \mathsf{spawn}(\theta,e,p'')\!:\!\pi, p, (\theta,e), q\rangle\rceil_{\#} \;\&\; \lfloor\langle \pi'', p'', (\theta'',e''), q''\rangle\rfloor_{\#_{\mathsf{sp}}} \;\&\; \Pi \end{array}$$

$$(\overline{\mathit{Spawn2}}) \quad \begin{array}{l} \Gamma; \lceil\langle \mathsf{spawn}(\theta,e,p'')\!:\!\pi, p, (\theta,e), q\rangle\rceil_{\#} \;\&\; \lfloor\langle [\,], p'', (\theta'',e''), q''\rangle\rfloor_{\#_{\mathsf{sp}}} \;\&\; \Pi \\ \quad \leftharpoondown \Gamma; \lfloor\langle \pi, p, (\theta,e), q\rangle\rfloor_{\#} \;\&\; \Pi \end{array}$$

$$(\overline{\mathit{Self}}) \quad \Gamma; \lfloor\langle \mathsf{self}(\theta,e)\!:\!\pi, p, (\theta',e'), q\rangle\rfloor_{\#} \;\&\; \Pi \leftharpoondown \Gamma; \lfloor\langle \pi, p, (\theta,e), q\rangle\rfloor_{\#} \;\&\; \Pi$$

$$(\overline{\mathit{Sched}}) \quad \begin{array}{l} \Gamma; \lfloor\langle \mathsf{sched}(p'',p,v)\!:\!\pi, p, (\theta,e), v\!:\!q\rangle\rfloor_{\#} \;\&\; \Pi \\ \quad \leftharpoondown \Gamma \mathbin{\overline{\cup}} (p'',p,v); \lfloor\langle \pi, p, (\theta,e), q\rangle\rfloor_{\#} \;\&\; \Pi \quad \text{if } \# \neq \#^{p'',v}_{\mathsf{sch}} \end{array}$$

Fig. 10. Backward semantics: Rules for backward computation.

actions that it has performed since it received the message, thus ensuring *causal consistency*. Once all these actions have been undone, rule $\overline{\mathit{Send3}}$ applies, resuming the forward computation for the receiver and the backward computation for the sender.

Analogously, for undoing the creation of a process, rule $\overline{\mathit{Spawn1}}$ freezes the process and marks the child process with label $\#_{\mathsf{sp}}$. The child process will then run backwards until, eventually, its trace is empty and rule $\overline{\mathit{Spawn2}}$ removes it from the system, resuming the backward computation for the spawning process.

Observe that, at first glance, one may think rule $\overline{\mathit{Receive}}$ should also introduce some new rollback operation for causal consistency. However, if we take a closer look, we will realize that receiving a message in our context is just about processing the message, and not actually receiving it. In fact, the processed message could have been delivered to the process mailbox a long time ago, and triggering a backward computation on the sending process would be unnecessary.

Finally, rule $\overline{\mathit{Sched}}$ applies when we find a term of the form $\mathsf{sched}(p,p'',v)$ in a trace and the rollback is not labelled with $\#^{p'',v}_{\mathsf{sch}}$ (since, in this case, rule $\overline{\mathit{Send3}}$ should be applied). Here, we just move the message back to the global mailbox Γ and continue undoing the remaining actions. The operator $\overline{\cup}$ is used

$$
\begin{array}{ll}
& [\,];\ \lfloor\langle \tau(\sigma, \mathsf{let}\ _ = v_2\ \mathsf{in} \ldots) : \mathsf{send}(\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots, \mathrm{c}_1, \mathrm{s}, v_2) : \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
& \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{receive}\ ack \to \ldots), [\,]\rangle\rfloor_{\#^{\mathtt{t}}_{\mathsf{ch}}}\ \& \\
& \quad \langle \mathsf{rec}(id, \mathsf{receive}\ \{P, M\} \to \ldots, [v_2]) : \mathsf{sched}(\mathrm{c1}, \mathrm{s}, v_2) : \pi'_i, \\
& \quad \mathrm{s}, (\theta_2, \mathsf{let}\ _ = P\ !\ ack\ \mathsf{in}\ \ldots), [\,]\rangle\ \&\ \langle \pi''_i, \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\leftharpoondown & [\,];\ \lfloor\langle \mathsf{send}(\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots, \mathrm{c}_1, \mathrm{s}, v_2) : \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
& \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{let}\ _ = v_2\ \mathsf{in} \ldots), [\,]\rangle\rfloor_{\#^{\mathtt{t}}_{\mathsf{ch}}}\ \& \\
& \quad \langle \mathsf{rec}(id, \mathsf{receive}\ \{P, M\} \to \ldots, [v_2]) : \mathsf{sched}(\mathrm{c1}, \mathrm{s}, v_2) : \pi'_i, \\
& \quad \mathrm{s}, (\theta_2, \mathsf{let}\ _ = P\ !\ ack\ \mathsf{in}\ \ldots), [\,]\rangle\ \&\ \langle \pi''_i, \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\leftharpoondown & [\,];\ \lceil\langle \mathsf{send}(\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots, \mathrm{c}_1, \mathrm{s}, v_2) : \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
& \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots), [\,]\rangle\rceil_{\#^{\mathtt{t}}_{\mathsf{ch}}}\ \& \\
& \quad \lfloor\langle \mathsf{rec}(id, \mathsf{receive}\ \{P, M\} \to \ldots, [v_2]) : \mathsf{sched}(\mathrm{c1}, \mathrm{s}, v_2) : \pi'_i, \\
& \quad \mathrm{s}, (\theta_2, \mathsf{let}\ _ = P\ !\ ack\ \mathsf{in}\ \ldots), [\,]\rangle\rfloor_{\#_{\mathsf{sch}}}\ \&\ \langle \pi''_i, \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\leftharpoondown & [\,];\ \lceil\langle \mathsf{send}(\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots, \mathrm{c}_1, \mathrm{s}, v_2) : \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
& \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots), [\,]\rangle\rceil_{\#^{\mathtt{t}}_{\mathsf{ch}}}\ \& \\
& \quad \lfloor\langle \mathsf{sched}(\mathrm{c1}, \mathrm{s}, v_2) : \pi'_i, \\
& \quad \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [v_2]\rangle\rfloor_{\#_{\mathsf{sch}}}\ \&\ \langle \pi''_i, \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\leftharpoondown & [\,];\ \lfloor\langle \tau(\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots) \\
& \quad : \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{let}\ _ = S\ !\ v_2\ \mathsf{in} \ldots), [\,]\rangle\rfloor_{\#^{\mathtt{t}}_{\mathsf{ch}}}\ \& \\
& \quad \langle \pi'_i,\ \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \&\ \langle \pi''_i, \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\leftharpoondown & [\,];\ \lfloor\langle \mathsf{check}(\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}) : \pi_i, \mathrm{c1}, (\sigma, \mathsf{let}\ _ = \mathtt{t}\ \mathsf{in} \ldots), [\,]\rangle\rfloor_{\#^{\mathtt{t}}_{\mathsf{ch}}}\ \& \\
& \quad \langle \pi'_i,\ \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \&\ \langle \pi''_i, \mathrm{c2}, (\sigma, ok), [\,]\rangle \\
\leftharpoondown & [\,];\ \langle \pi_i, \mathrm{c1}, (\sigma, \mathsf{let}\ _ = \mathsf{check}(\mathtt{t})\ \mathsf{in} \ldots, \mathtt{t}), [\,]\rangle\ \& \\
& \quad \langle \pi'_i,\ \mathrm{s}, (id, \mathsf{receive}\ \{P, M\} \to \ldots), [\,]\rangle\ \&\ \langle \pi''_i, \mathrm{c2}, (\sigma, ok), [\,]\rangle
\end{array}
$$

Fig. 11. A (partial) trace with the backward reversible semantics, where $\sigma = \{S \mapsto \mathrm{s}\}$, $\theta_2 = \{P \mapsto \mathrm{c1}, M \mapsto req\}$, and $v_2 = \{\mathrm{c1}, req\}$.

to add messages to the head of the corresponding list instead of to its end, i.e., $\Gamma \mathbin{\bar{\cup}} (\mathrm{p}, \mathrm{q}, \{\mathtt{t}, v\})$ denotes $\Gamma \setminus \{(\mathrm{p}, \mathrm{q}, vs)\} \cup \{(\mathrm{p}, \mathrm{q}, \{\mathtt{t}, v\} : vs)\}$.

Example 4. Consider the forward execution trace shown in Figure 9. A corresponding backward computation is shown in Figure 11.

Correctness. The rules of the backward semantics in Figure 10 basically *sequentialize* the backward computations for a given process. When a rollback operator is applied to a process p, we start undoing the actions in its trace until we find a concurrent action that may affect to other processes, like spawning a process or sending a message. In these cases, we freeze the backward computation of process p and propagate the rollback operator to the spawned process or to the receiver of a message, respectively. In particular, for a term of the form $\mathsf{spawn}(\theta, e, \mathrm{p}'')$, we freeze the process p and put a rollback operator on the spawned process p$''$. Only when all the actions of process p$''$ are undone and its trace is empty, we remove process p$''$ and resume the backward computation of process p. A similar behavior occurs when we find a term of the form $\mathsf{send}(\theta, e, \mathrm{p}, \mathrm{p}'', v)$. Therefore, causal consistency is ensured since no action can be undone until all the consequences of such an action are undone first.

Note that for undoing the delivery of a message, we do not propagate the rollback to the sender but just move the message back to the global mailbox. This is enough to ensure the correctness of the approach while minimizing the effects of the backward computation. Moreover, it can help to avoid a (possibly cyclic) *cascade* of rollback operators.

The correctness of our rollback operator is now stated as follows. Here, we consider a limited scenario for simplicity. Extending the proof to a more general case is not difficult but would require a longer proof scheme.

Lemma 1. *Let $\mathcal{P}$ be a program, and consider the following forward derivation:*

$$\begin{array}{l} \Gamma; \langle \pi, \mathrm{p}, (\theta, \mathsf{let}\ \_ = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \& \Pi \\ \rightharpoonup \Gamma; \langle \mathsf{check}(\theta, \mathsf{let}\ \_ = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e, \mathtt{t}) : \pi, \mathrm{p}, (\theta, \mathsf{let}\ \_ = \mathtt{t}\ \mathsf{in}\ e), q\rangle \& \Pi \\ \rightharpoonup^* \Gamma'; \langle \pi', \mathrm{p}, (\theta', e'), q'\rangle \& \Pi' \end{array}$$

where the processes of Π do not send messages to process p. *Then, we have*

$$\Gamma'; \lfloor \langle \pi', \mathrm{p}, (\theta', e'), q'\rangle \rfloor_{\#^{\mathtt{t}}_{\mathsf{ch}}} \& \Pi' \leftharpoondown^* \Gamma; \langle \pi, \mathrm{p}, (\theta, \mathsf{let}\ \_ = \mathsf{check}(\mathtt{t})\ \mathsf{in}\ e), q\rangle \& \Pi$$

5 Discussion

We have defined a reversible semantics for a first-order subset of Erlang that undoes the actions of a process step by step in a sequential way. To the best of our knowledge, this is the first attempt to define a reversible semantics for Erlang. As mentioned in the introduction, the closest to our work is the debugging approach based on a rollback construct of [11, 12, 15–17], but it is defined in the context of a different language or formalism. Also, we share some similarities with the checkpointing technique for fault-tolerant distributed computing of [9, 13], although the aim is different (they aim at defining a new language rather than extending an existing one).

As future work, we consider the definition of mechanisms to control reversibility to avoid storing history information at any time. This could be essential to extend Erlang with a new construct for *safe sessions*, where all the actions in a session can be undone if the session aborts. Such a construct could have a great potential to automate the fault-tolerance capabilities of the language Erlang.

Acknowledgements. We would like to thank Ivan Lanese and the anonymous reviewers for many useful suggestions to improve this paper.

References

1. Armstrong, J., Virding, R., Williams, M.: Concurrent programming in Erlang (2nd edition). Prentice Hall (1996)
2. Bennett, C.: Logical reversibility of computation. IBM Journal of Research and Development 17, 525–532 (1973)
3. Bennett, C.: Notes on the history of reversible computation. IBM Journal of Research and Development 44(1), 270–278 (2000)
4. Caballero, R., Martín-Martín, E., Riesco, A., Tamarit, S.: A declarative debugger for concurrent Erlang programs (extended version). Tech. Rep. SIC-15/13, UCM (2013), `http://maude.sip.ucm.es/~adrian/files/conc_cal_tr.pdf`

5. Cardelli, L., Laneve, C.: Reversible structures. In Proc. of CMSB 2011, pp. 131–140. ACM (2011)
6. Carlsson, R., Gustavsson, B., Johansson, E. *et al*: Core Erlang 1.0.3. language specification (2004), available from `https://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`
7. Danos, V., Krivine, J.: Reversible communicating systems. In Proc. of CONCUR 2004. Springer LNCS 3170, pp. 292–307 (2004)
8. Danos, V., Krivine, J.: Transactions in RCCS. In Proc. of CONCUR 2005. Springer LNCS 3653, pp. 398–412 (2005)
9. Field, J., Varela, C.A.: Transactors: a programming model for maintaining globally consistent distributed state in unreliable environments. In Proc. of POPL 2005, pp. 195–208. ACM (2005)
10. Frank, M.P.: Introduction to reversible computing: motivation, progress, and challenges. In Proc. of 2nd Conf. on Computing Frontiers, pp. 385–390. ACM (2005)
11. Giachino, E., Lanese, I., Mezzina, C.A.: Causal-consistent reversible debugging. In Proc. of FASE 2014. Springer LNCS 8411, pp. 370–384 (2014)
12. Giachino, E., Lanese, I., Mezzina, C.A., Tiezzi, F.: Causal-consistent Reversibility in a Tuple Based Language. In Proc. of PDP 2015. IEEE Computer Society, pp. 467–475 (2015)
13. Kuang, P., Field, J., Varela, C.A.: Fault tolerant distributed computing using asynchronous local checkpointing. In Proc. of AGERE! 2014, pp. 81–93. ACM (2014)
14. Landauer, R.: Irreversibility and heat generation in the computing process. IBM Journal of Research and Development 5, 183–191 (1961)
15. Lanese, I., Mezzina, C.A., Schmitt, A., Stefani, J.: Controlling reversibility in higher-order pi. In Proc. CONCUR 2011. Springer LNCS 6901, pp. 297–311 (2011)
16. Lanese, I., Mezzina, C.A., Stefani, J.: Reversibility in the higher-order π-calculus. Theor. Comput. Sci. 625, 25–84 (2016)
17. Lienhardt, M., Lanese, I., Mezzina, C.A., Stefani, J.: A reversible abstract machine and its space overhead. In Proc. of the Joint FMOODS 2012 and FORTE 2012 Int'l Conference. Springer LNCS 7273, pp. 1–17 (2012)
18. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In Proc. of ICFP 2007, pp. 47–58. ACM (2007)
19. Nishida, N., Palacios, A., Vidal, G.: Reversible term rewriting. In Proc. of FSCD 2016. Leibniz International Proceedings in Informatics (2016)
20. Phillips, I., Ulidowski, I.: Reversing algebraic process calculi. J. Log. Algebr. Program. 73(1-2), 70–96 (2007)
21. Svensson, H., Fredlund, L.A., Earle, C.B.: A unified semantics for future Erlang. In Proc. of the 9th ACM SIGPLAN workshop on Erlang, pp. 23–32. ACM (2010)
22. Thomsen, M.K., Axelsen, H.B.: Interpretation and programming of the reversible functional language RFUN. In Proc. of IFL 2015, pp. 8:1–8:13, ACM (2016)
23. Tiezzi, F., Yoshida, N.: Reversible session-based pi-calculus. J. Log. Algebr. Meth. Program. 84(5), 684–707 (2015)
24. Yokoyama, T.: Reversible computation and reversible programming languages. Electronic Notes in Theoretical Computer Science 253(6), 71–81 (2010), Proc. of the Workshop on Reversible Computation (RC 2009)
25. Yokoyama, T., Axelsen, H.B., Glück, R.: Principles of a reversible programming language. In Proc. of the 5th Conference on Computing Frontiers, pp. 43–54. ACM (2008)