Generation Of A Reversible Semantics For Erlang In Maude: Technical Report

Giovanni Fabbretti, Ivan Lanese

Abstract

In recent years, reversibility in concurrent settings has evoked quite some interests thanks to its many applications in different areas, like error-recovery, debugging, biological systems. Some of the formalism where it has been investigated are: petri-nets, process algebra and real programming languages. Nonetheless, all the attempts made so far suffer from the same limitation: they have been studied ad-hoc. To address this limit Lanese et al. recently proposed a new general method to derive a reversible semantics starting from a non-reversible one. The general method proposed though lacked an implementation that proves its feasibility not only in theory but in practice as well. The aim of the work presented here is to provide such missing implementation and apply it to the Erlang case study.

1 Introduction

Reversibility is the capability to execute a program both in a forward and backward manner. While reversibility is well-understood for sequential systems the same cannot be said for concurrent ones. The major difficulty while reversing the execution of a concurrent system is given from the lack of total order over the actions. To tackle this problem Danos and Krivine in [1] propose a notion of *Causal Consistency*, which aims at establishing a way to undo actions in a minimal and sound way. Causal Consistency states that an action can be undone iff all of its consequences, if any, have been undone beforehand.

Following their seminal work, reversibility has been studied in other formalisms as well, among which we find: the π-calculus [2], μKlaim [3], petri-nets [4], etc.

Another stem of research, investigating reversibility as a debugging technique, has risen after [5], where Giachino et al. proposed to equip a debugger with reversible primitives. Following the seminal work by Giachino et al. a great deal of effort has been placed in reversing the Erlang programming language [6, 7].

However, as anticipated, all the works cited so far suffer from the same limitation: reversibility has always been devised manually. Given a forward non-reversible formalism, the authors manually derived a forward reversible version and its symmetric backward counterpart. Clearly, deriving a reversible semantics manually presents some limitations: the process is

error-prone, does not scale well to other formalisms and lacks uniformity - i.e., the same properties must be proved every time.

To address this problem recently Lanese et al. in [8] proposed a new automatic general method to derive a reversible semantics starting from a non-reversible one. The advantages of having an automatic method are symmetrical to the disadvantages listed above, i.e., the method: is not error-prone; scales well; is uniform. The key idea behind the general method is to capture causal dependencies in terms of resources consumed and produced. The non-reversible semantics taken in input must a reduction semantics and the entities on the left-hand side of a rule are seen as the resources to be *consumed* to *produce* the resources on the right-hand side - i.e., the new entities. Then, keys and memories added to the semantics. Keys are used to uniquely identify processes while memories are used to recall past states so that eventually they can be restored.

However, in [8] the approach is only described theoretically and no implementation is provided. Our work aims at providing a concrete implementation of the general method, taking the Erlang programming language as case study and showing that the method proposed works not only in theory but in practice as well.

Notably, the main contributions of this work are: a novel formalization of Erlang using Maude; the implementation of a program that automatically derives a reversible semantics starting from a non reversible one.

The rest of the report will provide the reader with the required background in Section 2, then in Section 3 the main contributions will be analyzed in detail, while in Section 4 we discuss some ongoing work. Finally, in Section 5 we give some conclusions and we hints possible future directions.

All the code discussed in this report is publicly available at: `https://github.com/gfabbretti8/formalization-in-maude-of-erlang`.

2 Background

2.1 Erlang: Syntax And Semantics

Erlang is a functional, concurrent, programming language. First introduced in 1986 by Ericsson, has gained quite a lot of popularity since then. Today it is widely used and mostly appreciated because it is easy to learn, provides libraries to do concurrent and distributed programming, and because of its policies about handling failures. Erlang implements the actor model, a concurrency model based on message passing. In the actor model each process is identified as an actor that can interact with other actors only through the exchange of messages, no memory is shared. Indeed, centrals in Erlang are the send, receive and spawn operations.

The remaining of this section will provide the reader with a basic understanding of the language. We begin by illustrating its syntax, depicted in Fig. 1[1].

An Erlang program can be seen as a collection of modules, where a module is a sequence of function definitions. Each function is uniquely

[1]The syntax that we support is a subset of the real Erlang language.

$$
\begin{array}{rcl}
program & ::= & mod_1 \dots mod_n \\
mod & ::= & fun\_def_1 \dots fun\_def_n \\
fun\_def & ::= & Atom([patterns]) \rightarrow exprs. \\
pat & ::= & b\_value \mid Var \mid \ '\{'[patterns]'\}' \mid \ '['[patterns|patterns]']' \\
patterns & ::= & pat \ \{',' patterns\} \\
exprs & ::= & expr \ \{',' exprs\} \\
expr & ::= & b\_value \mid Var \mid \ '\{'[exprs]'\}' \mid \ '['[exprs|expr]']' \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clseq\ \mathsf{end} \mid \mathsf{receive}\ clseq\ \mathsf{end} \mid expr\ !\ expr \\
 & \mid & pat = expr \mid [Mod:]expr([exprs]) \\
b\_value & ::= & Atom \mid Char \mid Float \mid Integer \mid String \\
clseq & ::= & pat \rightarrow exprs \ \{';' pat \rightarrow exprs\}
\end{array}
$$

Figure 1: Language syntax

identified by its name and by the number of formal parameters, its body is represented by a sequence of expressions. In the following we denote an expression with e and sequences of expressions as $e_1, \dots, e_n$ - sequences of other syntactical elements are represented in the same manner.

Ground values in Erlang are: atoms (which are identifiers that either begin with a lowercase or are enclosed by quotes), integers and strings and any construction of these elements using tuples and lists. We denote atoms, strings and integers by v. Tuples are denoted as $\{v_1, \dots, v_n\}$ and lists are denoted as $[v_1|v_2]$, where v_1 is the head and v_2 the tail.

Spanning over ground values we have variables, which can be distinguished from other syntactical elements as they always have the first letter capital and are not enclosed in quotes, e.g., X, Y. Then, we have patterns, denoted by *pat*, which are like the ground-values but also admit the presence of variables. Patterns are used in branching statements, like $\mathsf{receive}\ pat_1 \rightarrow exprs_1; \dots; pat_n \rightarrow exprs_n\ \mathsf{end}$, to match the branch to evaluate, in the matching operation, i.e., $e_1 = e_2$, or in functions definition, to define the formal parameters.

Let us now talk about the semantics of some of most important Erlang constructs. One of the most distinguishing operation of Erlang is the *pattern matching*. Pattern matching occurs in the following scenarios: i) $e_1 = e_2$; ii) $\mathsf{case}\ e\ \mathsf{of}\ clseq\ \mathsf{end}$; iii) $\mathsf{receive}\ clseq\ \mathsf{end}$.

We begin by explaining the first case and then we move on to the others. The expression on the left-hand side, e_1, is evaluated until it becomes a pattern, or a ground value in case no free variables occur in it. Then, the expression on the right-hand side is evaluated until it becomes a ground value, eventual occurrences of free variables would raise an exception, and then the two elements are matched against each others. The free variables of the left-hand side are bound to the corresponding ground value of the right-hand side and ground values are compared together, if a mismatch occur an exception is raised. If no mismatch occurs then the operation evaluates to the ground values of the right-hand side and the environment is updated.

In the case scenario, the expression e must evaluate to a ground value,

then it is matched against the patterns, from top to down, until one that matches is found, when a match is found the environment is enriched with the new bindings and the sequence of expressions returned, if no match is found an exception is raised.

In the receive scenario things are similar to the case one, only the ground values are the messages in the queue of the process, which are checked one by one against the clauses and when a match is found the corresponding branch is selected. Conversely to the case, if a match is not found then the language does not raise an exception but suspends the process execution.

Despite being - mostly - functional Erlang admits some imperative operations that produce side-effects, like the receive above, or like spawning a new process, or sending a message.

Messages are sent with the syntax, $e_1!e_2$, where e_1 must evaluate to the pid of the receiver and e_2 must evaluate to the ground values that represents the payload of the message. The expression itself evaluates to the payload and as a side-effect the message is sent.

The spawn is the primitive to create a new process; it takes in input the function that the new process will execute together with the arguments to supply - if any. Once evaluated, the spawn will return the pid of the newly created process and as a side effect the process will be created.

Finally, the function self is used to get the process id of the process who invoked it.

Self, send, spawn and receive are the concurrent features, offered by Erlang, that we support in this work.

2.2 Maude

Maude [9] is a programming language that efficiently implements the rewriting logic [10]. Intuitively, a rewriting logic can be seen as a framework that unifies equational logic together with semantics rules.

Formally, a rewriting logic is a tuple (Σ, E, R), where Σ represents a collection of typed operators, E a set of equations among the operators and R is the set of semantics rules. Using a rewriting logic is quite convenient to formalize the semantics of a language as it brings together the benefits of using both an equational theory together with rewriting rules.

On one hand, the equational side of the rewriting logic is well-suited to define the deterministic part of the model, where we define class of equivalences over terms. More precisely we say that two terms v and u are equivalent if under a set of equation E we can prove $E \vdash v = u$. Equations can be conditional as well and can use as condition either the membership of the term to some kind or other equations.

On the other hand, the rewriting rules are well-suited to define the concurrent (non-deterministic) part of the programming language. The set of rules R specify how to rewrite a (parametrized) term t to another term t'. Rewriting rules, like the equations, can be conditional and as condition they can use membership, equations as well as other rules.

In other words the equational theory defines which terms defines the same states of a system, only using different syntactical elements, while

```
mod H is
   IL
   sorts SS .
   SSDS
   OPDS
   MAS
   EQS
   RLS
endm .
```

Figure 2: A generic maude module.

the rewriting rules define how the system can evolve and transit from one state to another.

A module in Maude has the shape depicted in Fig. 2, where: `H` is the module name; `IL` is the import list; `SS` is the set of sort declaration; `SSDS` is the set of sub-sort declaration; `OPDS` is the set of operators declaration; `MAS` is the set of membership declarations; `EQS` is the set of equations; `RLS` is the set of rewriting rules.

The transformation of the non-reversible semantics is defined in terms of a program that takes in input the modules of the non-reversible semantics and produces new modules, which define the reversible semantics.

2.3 A General Approach To Derive A Reversible Semantics

The following of this section summarizes the main ideas of [8] where Lanese et al. propose a methodology to automatically derive a reversible semantics starting from a non-reversible one, given that this last one is equipped with a reduction semantics that fits some criteria.

2.3.1 Format Of The Reduction Semantics

We proceed now to describe the shape that the reduction semantics taken as input must have.

First, the syntax must be divided in two levels: on the lower level there are no restrictions, while the productions of the upper level must be of the following form:

$$S ::= P \mid op_n(S_1, \ldots, S_n) \mid \mathbf{0}$$

where $\mathbf{0}$ is the empty entity, P an entity of the lower level and $op(S_1, \ldots, S_n)$ an operator composing entities. An entity of the lower-level could, for example, be a process of the system or a message traveling the network.

Second, the rules defining the operational semantics must fit the schema in Fig. 3. The schema allows rules to: i) allow entities to interact with each other; ii) exploit structural congruence; iii) allow single entities to execute at each step; iv) run the system in parallel.

$$(\textsc{Scm-Act})\ \frac{}{P_1 \mid \ldots \mid P_n \rightarrowtail T[Q_1, \ldots, Q_m]} \qquad (\textsc{Eqv\_})\ \frac{S \equiv_c S' \quad S \rightarrowtail S_1 \quad S_1 \equiv_c S'_1}{S' \rightarrowtail S'_1}$$

$$(\textsc{Scm-Opn})\ \frac{S_i \rightarrowtail S'_i}{op_n(S_0, \ldots, S_i, \ldots, S_n) \rightarrowtail op_n(S_0, \ldots, S'_i, \ldots, S_n)} \qquad (\textsc{Par})\ \frac{S \rightarrowtail S'}{S \mid S_1 \rightarrowtail S' \mid S_1}$$

Figure 3: Required structure of the semantics in input; Scm- rules are schemas

$$(\textsc{F-Scm-Act})\ \frac{j_1, \ldots, j_m \text{ are fresh keys}}{k_1 : P_1 \mid \ldots \mid k_n : P_n \twoheadrightarrow T[j_1 : Q_1, \ldots, j_m : Q_m] \mid [k_1 : P_1 \mid \ldots \mid k_n : P_n \,; T[j_1 : \bullet_1, \ldots, j_m : \bullet_m]]}$$

$$(\textsc{F-Scm-Opn})\ \frac{R_i \twoheadrightarrow R'_i \quad (\mathtt{keys}(R'_i) \setminus \mathtt{keys}(R_i)) \cap (\mathtt{keys}(R_0, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n) = \emptyset}{op_n(R_0, \ldots, R_i, \ldots, R_n) \twoheadrightarrow op_n(R_0, \ldots, R'_i, \ldots, R_n)}$$

$$(\textsc{F-Eqv})\ \frac{R \equiv_c R' \quad R \twoheadrightarrow R_1 \quad R_1 \equiv_c R'_1}{R' \twoheadrightarrow R'_1}$$

Figure 4: Forward rules of the uncontrolled reversible semantics

2.3.2 Methodology

To obtain a forward reversible semantics, first, the syntax of the reduction semantics is updated as follow:

$$\begin{aligned} R &::= k : P \mid op_n(R_1, \ldots, R_n) \mid \mathbf{0} \mid [R \,; C] \\ C &::= T[k_1 : \bullet_1, \ldots, k_m : \bullet_m] \end{aligned}$$

Two modifications have been done. First, each entity of the system is tagged with a key. Keys are used to distinguish identical processes with a different history. Second, the syntax is updated with another production: memories. Memories have the shape $\mu = [R; C]$, where R is the configuration of the system that gave rise to the forward step and C is a placeholder of the resulting one. C acts as the binder between R and the actual final configuration. In other words, memories bind different states of the entities. Moreover, they keep track of past states of the system so that eventually they can be restored.

Then, the rules of the non-reversible operational semantics are updated as depicted in Fig. 4. Now each time a forward step is performed each resulting entity is tagged with a fresh key and a memory, connecting the old configuration with the new one, is produced. Here, it is possible to manipulate the rules without actually knowing them because they fit the schema and they cannot have completely arbitrary forms.

The backward rules, depicted in Fig. 5, are symmetric to the forward ones: when a memory μ and the entities tagged with the keys in C are both available then a backward step can be performed and the old configuration R can be restored.

$$\text{(B-Scm-Act)}\ \frac{\mu = [k_1 : P_1 \mid \ldots \mid k_n : P_n \,; T[j_1 : \bullet_1, \ldots, j_m : \bullet_m]]}{T[j_1 : Q_1, \ldots, j_m : Q_m] \mid \mu \rightsquigarrow k_1 : P_1 \mid \ldots \mid k_n : P_n}$$

$$\text{(B-Scm-Opn)}\ \frac{R_i' \rightsquigarrow R_i}{op_n(R_0, \ldots, R_i', \ldots, R_n) \rightsquigarrow op_n(R_0, \ldots, R_i, \ldots, R_n)}$$

$$\text{(B-Eqv)}\ \frac{R \equiv_c R' \quad R \rightsquigarrow R_1 \quad R_1 \equiv_c R_1'}{R' \rightsquigarrow R_1'}$$

Figure 5: Backward rules of the uncontrolled reversible semantics

The reversible semantics produced by this approach captures causal dependencies in terms of resources produces and consumed, since, thanks to the memory, a causal link is created each time that some entities are rewritten. We refer to [8] for the formal demonstration of the causal-consistency of the reversible semantics. We also remark that the semantics here produced is uncontrolled, i.e., if several rules can be fired at the same time there is no policy on which one to choose.

3 Formalizing Erlang and Generating The Reversible Semantics

3.1 Formalizing Erlang

The work presented here has been strongly inspired by [12], where the authors formalized the semantics of Core-Erlang to do model-checking on it. While our final semantics is quite different from the one they presented (the most notable differences are that we formalize full Erlang and the use of labels) we were still able to re-use some of their modules and some of their ideas, like the internal representation of the ground-values, which greatly simplified the formalization task.

Our formalization of the semantics follows the style of the one in [11] with some differences, which we will soon discuss. As in [11], the semantics is a two layer semantics, one layer for the expressions level and another layer for the system level. This division is quite convenient for the formalization in Maude, as we can formalize the expression level as an equational theory and then using rewriting rules to describe the concurrent features, i.e., the system level.

The system level comprises a rewriting rule for each concurrent feature and a rewriting τ rule for sequential operations. While it would have been possible to define all the sequential operations as an equational theory we still decided to have a rewriting rule to perform single sequential steps to better simulate the behavior of a step-by-step debugger.

Before diving in discussing the rewriting logic let us first discuss the entities that compose our Erlang system. Processes are defined as the

following tuple:

$$\langle p, \theta, e, me \rangle$$

where p is the process unique id, θ is the store binding variables to values[2], e is the expression currently under evaluation and me is the module environment, which contains the functions that p can either invoke or spawn. Then we have messages, which are defined by the following tuple:

$$\langle p, p', v \rangle$$

where p is the pid of the sender, p' is the pid of the receiver and v is the payload. In the scope of this work processes and messages corresponds to the lower level of the syntactical productions, what we named P in 2.3.

A running system is denoted by messages and processes composed using the parallel operator.

Now, let us analyze in detail the shape of the rewriting logic by first analyzing the equational theory for the expressions. The theory is defined as a set of equations which have one of the following generic forms:

$$\begin{array}{l} \mathsf{eq}: \ [equation - name] \\ \quad \langle l, \theta, \ e \rangle = \langle l', \theta', \ e' \rangle \end{array}$$

$$\begin{array}{l} \mathsf{ceq}: \ [equation - name] \\ \quad \langle l, \theta, \ e \rangle = \langle l', \theta', \ e'' \rangle \\ \quad \mathsf{if} \ \langle l', \theta', e' \rangle = op(l, \theta, e) \ \wedge \ \langle l'', \theta'', \ e'' \rangle := \langle l', \theta', e' \rangle \end{array}$$

First, as we can see from above to evaluate an expression we do not only need the expression itself but we also need two additional items: a store θ and a label l. The store plays a single role, i.e., binds variables to their values, given that they have one. The label plays two roles: i) names the action performed by the process; ii) communicates data back and forth between the expression level and the system level. Examples of this mechanism will be soon introduced.

Second, we can observe that two kinds of rules exist, one kind which admit a if branch and another kind that does not. The equations defined without the if clause always define a reduction of the expression - i.e., a step taken by some process - and change the label to the appropriate one. On the other hand, the conditional equations can: either define a single step, that requires some side conditions (e.g., binding a variable to its value), or perform some intermediate operation (e.g., selecting the inner expression to evaluate) and then with the clause $t := t'$ use recursively other equations to reach a canonic form.

Let us now focus on the rewriting rules, which have the following general shape:

$$\begin{array}{l} \mathsf{crl}: \ [rule - name] \\ \quad \langle p, \theta, e, me \rangle \mid E \Rightarrow \langle p, \theta', \ e', me \rangle \mid op(l', E) \\ \quad \mathsf{if} \ \langle l', \theta', e' \rangle := \langle l, \theta, e \rangle \end{array}$$

In the schema above E defines other entities of the system - potentially even the empty one. As one can see, rules are always conditional, as we

[2]Actually θ is a stack of stores, later we will clarify why.

always rely on the expression semantics to understand which action the selected process is ready to perform. Finally, we express the side-effects through *op* according to the system shape and on the action that p wants to perform. In examples 3.1 and 3.2 we show possible concrete instances of E.

Example 3.1. Let us consider the example in Fig. 6, which is the rewriting conditional rule to send a message. In the conditional part of the rule we can see the equational theory in action, more precisely we are checking if, by using the equations defined on the expressions, it is possible to obtain a tuple as the one defined on the left-hand side of the operator `:=`. If it is possible to equate the two terms, it means that the process is ready to evaluate a send somewhere in the expression that is under evaluation. We know that the next operation to perform is a send thanks to the first element of the tuple. Moreover, from the resulting label, i.e., `DEST ! GVALUE`, we retrieve the pid of the receiver and the payload of the message. Here, E, on the left-hand side, will be the empty entity, then on the right-hand side E will be replaced by the message that has been sent. This is an example of how the label serves to communicate information from the expression level to the system one. Finally, once that all the information have been gathered, the system level defines the side-effects that must be performed, in this case the sending of a message.

To fully understand why we took some design choices, while we proceed in the explanation of the operational semantics, we discuss the differences w.r.t. the formalization in [11].

First, in [11] the expression semantics is not directly defined on expressions, but on an operator $C[\bullet]$, which is used to select the part of the expression that has to be evaluated, however no definition of such operator is given. For example, if the expression under evaluation is $\mathsf{case}\ foo(42)\ \mathsf{of} \ldots$ the context operator would put automatically the focus on $foo(42)$, i.e., $C[foo(42)]$. The convenience of using such operator is that in this way one can avoid the definition of all the rules to evaluate inner parts of an expression, like in the example just presented.

In Maude, since the semantics is runnable, we need to define every operator. So, we opted for an approach closer to the one in [6], where we have rules that recursively evaluate the expression until a computation is executed - as already discussed, we can use conditional equations to first perform operations and then use others equations to further reduce.

The second difference as well is due to the use of the context operator. In Erlang, we could face scenarios where the evaluation of some expressions could produce an illegal expression. Consider, for example, the case $\mathsf{case}\ foo(42)\ \mathsf{of}$ where $foo(N) \rightarrow \mathsf{spawn}(bar, N), N * N$. The evaluation of such expression would produce an illegal expression as the case construct expects a single expression. In some other cases, the expression produced would be legal but not having the intended effect, like $X = pow\_and\_sub(N, M)$ where $pow\_and\_sub(N, M) \rightarrow Z = N{*}N, Z{-}M.$, which would became $X = N * N, Z - M.$, where clearly the effect would not be the desired one.

To avoid these problems the solution adopted in [11] is to push the current environment together with the context operator, where the term

```
crl [sys-send] :
    < P | exp: EXSEQ, store-stack: STORE, ASET > =>
    < P | exp: EXSEQ', store-stack: STORE', ASET > ||
    < sender: P, receiver: DEST, payload: GVALUE >
    if < DEST ! GVALUE, STORE', EXSEQ' > :=
       < req-gen, STORE, EXSEQ > .
```

Figure 6: System rule send

under evaluation is replaced with a placeholder, in a stack and to start a new subcomputation - with the appropriate environment. Then, once the subcomputation is over the old contest and environment are retrieved from the stack and the final value of the subcomputation is replaced inside the expression which gave rise to it.

Again the context operators makes the formalization of this mechanism much easier. Indeed in reality, to implement such approach it would be necessary to define, for each supported construct, a rule to start the subcomputation and another rule to replace the value once the subcomputation is over. We preferred to adopt a different solution, which halves the number of rules required. Such solution consists of having a stack of stores inside each process, so θ in reality has the shape $\theta_1 : \ldots : \theta_n$. Then, each time an expression e that produces a sequence of expressions $e_1, \ldots, e_n$ is encountered we wrap the sequence with begin $e_1, \ldots, e_n$ end and we push on the stack of stores the appropriate store to evaluate $e_1, \ldots, e_n$. In case e is a function call the appropriate store is a new environment where eventual formal parameters of the function are bound to the actual ones, while if e is a case or a receive statement then the appropriate store is the previous one enriched with the eventual variables that got bounded during the pattern matching. Then, once the subcomputation is over and the expression has the shape begin v end we simply replace it with v and we pop one store from the stack. In such way we do not need rules to move around the expression under evaluation and we never create illegal or unwanted expressions. In Fig. 7 we can see an example of how the `receive` equation wraps the sequence of expressions returned and how the new store is stacked.

The final difference between our semantics and the one proposed in [11] concerns those rules of the expression semantics that need to rely on some information held by the system level. This is the case for instance for the self primitive or the receive statement, in the first case the information required is the process pid, held in the process tuple, and in the second case the information required is the message which has to be received.

To address the problem the solution proposed in [11], already used in [6], is to evaluate those rules that need some information available at the system level to a symbol κ, in other words a *future*. Then, the duty of replacing κ with the appropriate information is delegated to the system level.

The problem that would rise here is similar to the one discussed above about stacking expressions and environments to start subcomputations

```
crl [sys-receive] :
  < P | exp: EXSEQ, store-stack: STORE, ASET > ||
  < sender: SENDER, receiver: P, payload: GVALUE > =>
  < P | exp: EXSEQ', store-stack: STORE', ASET >
if < received, STORE', EXSEQ' > :=
   < req-receive(GVALUE), STORE, EXSEQ > .

ceq [receive] :
  < req-receive(PAYLOAD), STORE : STORESTACK, receive CLSEQ end, ME > =
  < received, STORE' : (STORE : STORESTACK), begin EXSEQ end, ME >
  if #entityMatchSuccess(EXSEQ | STORE') :=
     #entityMatch(CLSEQ ; #empty-clauselist | PAYLOAD | STORE ) .
```

Figure 7: System rule receive

and then replace the final value in the correct place: the number of rules to define the substitution.

The solution that we propose makes so that the expression semantics already has all the information necessary to reduce, and making sure that the expression semantics has the necessary information is the second role of the label l. The presence of the label in the tuple, over which the expression semantics is defined, makes possible to bubble up, from the system level, information so that is possible to reduce the expression without creating futures, since all the elements to reduce are already available locally.

Example 3.2. Let us consider the rule in Fig. 7. Rule sys − receive is the rule that we apply when a process receives a message. The preconditions for the successful firing of the rule are that the message targeting the selected process is floating in the soup and that the process is performing a receive with a clause aimed at receiving the message. What we do is bubbling up the payload of one of the (potential many) messages ready to be delivered to `P` with the label `req-receive(GVALUE)`, then, if there is a receive statement to be evaluated and one of its clauses matches the message, we already have all the information necessary to select the appropriate branch locally, as can be seen in the equation receive. Here, to fit the concrete instance of the rule in to the generic schema presented above we need to instantiate the E on the left-hand side as the message to be received. Then, the operator expressing the side effect corresponds to the removal of the message from the system.

3.2 Producing The Reversible Semantics

To produce the reversible semantics we decided to remain within Maude, mostly for two main reasons. First, Maude is well-suited to define program transformations thanks to its META-LEVEL module which contains facilities to meta-represent a module and to manipulate it. Second, since we defined Erlang's semantics inside Maude we do not need to define a

```
mod SEM_ENTITIES is
...
  sort Entities .
  subsort Entity < Entities .

  op #empty-entities : -> Entities [ctor] .
  op _||_ : Entities Entities -> Entities [ctor assoc comm .. ] .
...
endm
```

Figure 8: Example of the SEM_ENTITIES module for Erlang.

parser for it as it is already loaded and can be easily meta-represented by using the function *upModule* inside the module META-LEVEL.

3.2.1 Format Of The Non-Reversible Semantics

Let us now describe the format that the semantics to be transformed must have. First, it must have a module named SEM_ENTITIES, which must contain the definition of the top-level operators, where the operators defined correspond to the upper level of the syntactical productions, as discussed in 2.3 and the empty-entity. All the operators inside the module SEM_ENTITIES must be of sort 'Entities' and their inputs must of sort 'Entities' as well. The subsort relation 'Entity' <'Entities' must be declared as well, in fact it is necessary to use entities of the lower level inside the operators defined in the module. This imply that the sort of the lower level operators must extend the sort Entity.

In Fig. 3.2.1 it is depicted the entities module for the Erlang languages. We omitted the import of other modules, as well as the other elements since they are not interesting in this context.

The rewriting rules of the rewriting theory that defines the single steps of the reduction semantics must be defined under the module 'SEM_-SYSTEM'. Examples of the rules can be found in Fig. 6 and 7.

3.3 Transformation

The first transformation consists of adding a new *sort* to the module SEM_ENTITIES, i.e., the sort 'EntityWithKey'. The sort 'EntityWithKey' is built by composing an entity of the lower level of the semantics and a key. Keys play the role described in 2.3. Then the subsort relation 'EntityWithKey' <'Entities' is added to the module, so that now all the top-level operators can deal with tagged entities.

Also, a new sort 'Placeholder' is declared and a new operator that when given a key builds a placeholder is defined - this operator is what we called C in 2.3.2.

Then, memories are added, by adding the sort 'Memory' and by defining the operator that builds memories by combining a final configuration where

the entity have been replaced by their placeholders together with tagged entities, which represent the state that gave rise to the step.

The final transformation concerning the Entities module is to update the equations to meta-represents and to lower the entities representation as now they are entities tagged with a key.

3.4 Producing The Reversible Semantics

The transformation to be performed over the rewriting rules again is the same as the one described in 2.3.2. Rules now must deal with tagged entities and each time that a forward step is taken the resulting entities must be tagged with unique keys and the appropriate memory must be created.

The transformation is mostly forward, the only tricky part concerns the uniqueness of the keys and their generation. Indeed, we must have a 'distributed' way to compute them, as passing around a key generator would produce spurious dependencies. To solve the problem we resorted to the following idea. Keys are represented as lists of integers. Each time we need to produce new unique keys, to tag the (new) entities on the right-hand side of a rule, we first collect all the keys on the left-hand side of the rule, then we concatenate them together to a new list, say l, and finally we tag each of the new entities with l concatenated with a different number for each entity.

In practice, to transform a rule, we tag each entity of the left-hand side with a variable that represents the key and we keep track of the variables used. Then, to transform the right-hand side we tag all the created entities as described above, using the keys of which we kept track, and we also introduce the memory created by the rule application.

In Fig. 10 it is possible to observe rule `send` before and after being transformed. As one can see, on the left-hand side, of the reversible version, the process is initially tagged with some key, then the new entities on the right side are tagged with fresh keys, built from the one already available locally. Moreover, now each application of the rule will also produce a memory binding the two states.

The production of the backward semantics is even more straightforward of the forward one, in fact to produce it it is sufficient swap the left-hand side of each forward reversible rule with its right-hand side and get rid of the conditional branch.

The conditional branch is not required anymore because if the process has performed the forward step then it can always perform the backward one, given that all the consequences of the action have been already undone. This last condition is 'automatically' ensured by the presence of the memory together with the entities bound by the placeholder inside the memory itself.

4 Ongoing Work

The last step to conclude this work is to close the gap between the format of the rules expected by the general method and the actual format of

```
  crl [sys-send] :
      < P | exp: EXSEQ, store-stack: STORE, ASET > =>
      < P | exp: EXSEQ', store-stack: STORE', ASET > ||
      < sender: P, receiver: DEST, payload: GVALUE >
      if < DEST ! GVALUE, STORE', EXSEQ' > :=
         < req-gen, STORE, EXSEQ > .

crl [label sys-send]:
    < P | ASET, exp: EXSEQ, store-stack: STORE > * key(N)
    => < sender: P, receiver: DEST, payload: GVALUE > * key(0 N) ||
       < P | exp: EXSEQ', store-stack: STORE', ASET > * key(1 N) ||
       [< P | ASET, exp: EXSEQ,store-stack: STORE > * key(N) ;
        @: key(0 N) || @: key(1 N)]
     if < DEST ! GVALUE, STORE', EXSEQ' > := < req-gen,STORE, EXSEQ > .
```

Figure 9: Forward And Forward Reversible Rule Send

```
rl [label sys-send]:
   < sender: P, receiver: DEST,payload: GVALUE > * key(0 N) ||
   < P | exp: EXSEQ', store-stack: STORE', ASET > * key(1 N) ||
   [< P | ASET, exp: EXSEQ, store-stack: STORE > * key N ;
     @: key(0 N) || @: key(1 N)]
   => < P | ASET, exp: EXSEQ, store-stack: STORE > * key N
```

Figure 10: Reversible backward rule send

the rules provided in input. In fact, the schema of the general method allows for an arbitrary number of rules (potentially infinite) describing the internal steps of the system. Obviously, to efficiently describe a system we cannot produce infinite rules, actually quite the contrary, i.e., the less we have the better it is. So, in the formalization of the rules we resorted to the expression level semantics. In particular, we used predicates - i.e., the expression level semantics - to gather under the same meta-rule all infinite rules that would describe the same behavior.

Let us consider the following processes:

$$\langle p, \theta, 2\ !\ 'hello', me\rangle$$
$$\langle p', \theta', \mathsf{case}\ 2\ !\ 'hello'\ \mathsf{of}\ \ldots, me'\rangle$$
$$\langle p'', \theta'', X = 2\ !\ 'hello', me''\rangle$$

The three processes above are all ready to perform the same action, even though they have a different configuration, nonetheless thanks to the expression level semantics we are able to formalize their behavior in one meta-rule send.

Finally, what is required is a (set of) correctness theorem(s) to prove that all the instances of the meta-rules used to formalize the languages are correct rules in the sense required from [8] so that all the backward rules derived are also correctly modeling the backward semantics.

5 Conclusion

To conclude the technical report let us now briefly resume the work done so far and quickly discuss some future items.

First, we presented a new formalization of the Erlang language using Maude. Having a mechanized version of the Erlang semantics makes it much easier to debug it and to get convinced that the semantics correctly captures the behavior of the language. Indeed, to test the semantics, in our work, one can load an Erlang module and actually run an arbitrary complex program (as long as the program uses supported primitives) and the various states traversed can be compared with an actual execution to make sure that the first adheres to the latter.

Concretely formalizing a languages poses also some challenges that usually do not rise. For instance, it is often the case that 'theoretical semantics' are not directly implementable, in fact to keep them compact and elegant sometimes some operators are not fully defined or sometimes some parameters appears out of the blue. In section 3.1 we have discussed some examples.

Conversely, while defining a mechanized semantics it is not possible to leave operators undefined or make parameters appearing out of the blue. This constraint forced us to get creative during the formalization process, to contain the number of rules, e.g., using labels to communicate information back and forth between the two layers of the semantics.

Second, we implemented a program able to transform a non-reversible semantics in to a reversible one, providing an implementation of the general method described by [8]. One interesting detail of the implementation, where again we had to be creative, is the implementation of keys. In their work, due to the abstract setting, to generate new keys the authors resorted to the use of a predicate to generate fresh keys. In a concrete setting such predicate must be fully defined, to do so we implemented keys as list of integers so that it would be possible to generate new unique identifiers in a "distributed manner".

To conclude, let us discuss a couple of possible future directions for the presented work. First, one could investigate ways to optimize the implementation of the semantics, for example the way in which stores are managed is very naive, the main advantage of doing so would be the ability of simulate even computationally expensive erlang programs. Second, one could expand the set of supported primitives to widen the set of supported programs, in doing so of course it is important to make sure that the causal-dependencies captured by the producer-consumer model are appropriate - for example the model is not well-suited to capture read dependencies.

References

[1] Vincent Danos and Jean Krivine. Reversible communicating systems. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004 - Concurrency Theory, 15th International Conference, London, UK,*

August 31 - September 3, 2004, Proceedings, volume 3170 of *Lecture Notes in Computer Science*, pages 292–307. Springer, 2004.

[2] Ivan Lanese, Claudio Antares Mezzina, and Jean-Bernard Stefani. Reversibility in the higher-order π-calculus. *Theor. Comput. Sci.*, 625:25–84, 2016.

[3] Elena Giachino, Ivan Lanese, Claudio Antares Mezzina, and Francesco Tiezzi. Causal-consistent rollback in a tuple-based language. *J. Log. Algebraic Methods Program.*, 88:99–120, 2017.

[4] Anna Philippou and Kyriaki Psara. Reversible computation in petri nets. In Jarkko Kari and Irek Ulidowski, editors, *Reversible Computation*, pages 84–101, Cham, 2018. Springer International Publishing.

[5] Elena Giachino, Ivan Lanese, and Claudio Antares Mezzina. Causal-consistent reversible debugging. In Stefania Gnesi and Arend Rensink, editors, *Fundamental Approaches to Software Engineering - 17th International Conference, FASE 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014.

[6] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for erlang. *J. Log. Algebraic Methods Program.*, 100:71–97, 2018.

[7] Giovanni Fabbretti, Ivan Lanese, and Jean-Bernard Stefani. Causal-consistent debugging of distributed erlang programs. In Shigeru Yamashita and Tetsuo Yokoyama, editors, *Reversible Computation - 13th International Conference, RC 2021, Virtual Event, July 7-8, 2021, Proceedings*, volume 12805 of *Lecture Notes in Computer Science*, pages 79–95. Springer, 2021.

[8] Ivan Lanese and Doriana Medic. A general approach to derive uncontrolled reversible semantics. In Igor Konnov and Laura Kovács, editors, *31st International Conference on Concurrency Theory, CONCUR 2020, September 1-4, 2020, Vienna, Austria (Virtual Conference)*, volume 171 of *LIPIcs*, pages 33:1–33:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[9] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn L. Talcott, editors. *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[10] José Meseguer. Rewriting logic as a semantic framework for concurrency: a progress report. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory*, pages 331–372, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[11] Juan José González-Abril and Germán Vidal. *Causal-Consistent Reversible Debugging: Improving CauDEr*, volume 12548 of *Lecture Notes in Computer Science*, page 145–160. Springer International Publishing, 2021.

[12] Martin Neuhäußer and Thomas Noll. Abstraction and model checking of Core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):147–163, Jul 2007.