Automatic Generation of a Reversible Semantics for Erlang in Maude

Giovanni Fabbretti [1], Ivan Lanese [2], Jean-Bernard Stefani[1]

[1]SPADES Team, INRIA and [2]FOCUS Team, University of Bologna/INRIA

ICFEM, 25-10-2022

Sequential Reversibility and Debugging

Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

Sequential Reversibility and Debugging

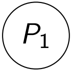
Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

Useful in debugging: one can chase bugs to their roots rather than executing the same program several times

Sequential Reversibility and Debugging

Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

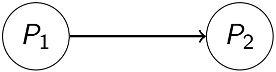
Useful in debugging: one can chase bugs to their roots rather than executing the same program several times



Sequential Reversibility and Debugging

Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

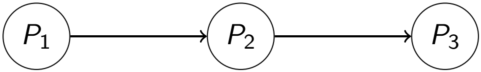
Useful in debugging: one can chase bugs to their roots rather than executing the same program several times



Sequential Reversibility and Debugging

Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

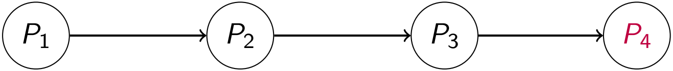
Useful in debugging: one can chase bugs to their roots rather than executing the same program several times



Sequential Reversibility and Debugging

Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

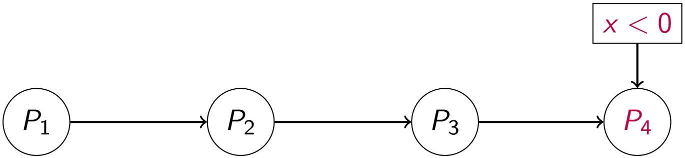
Useful in debugging: one can chase bugs to their roots rather than executing the same program several times



Sequential Reversibility and Debugging

Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

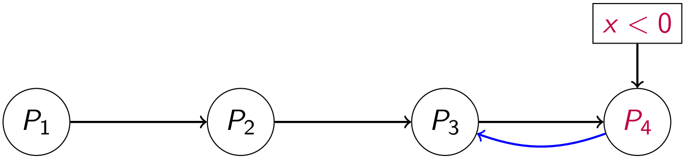
Useful in debugging: one can chase bugs to their roots rather than executing the same program several times



Sequential Reversibility and Debugging

Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

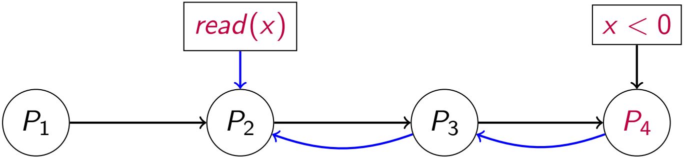
Useful in debugging: one can chase bugs to their roots rather than executing the same program several times



Sequential Reversibility and Debugging

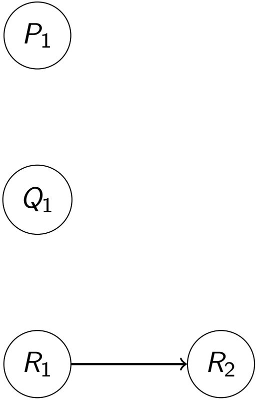
Reversibility is the ability to execute a program not only in the canonical forward direction but in a backward manner as well

Useful in debugging: one can chase bugs to their roots rather than executing the same program several times

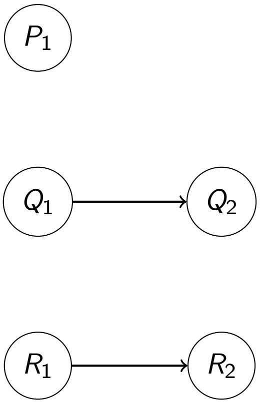


Concurrent Reversibility

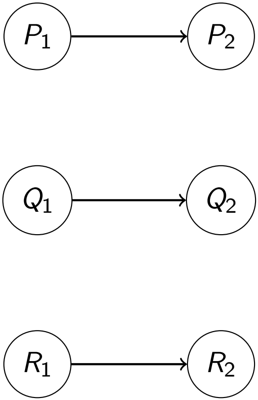
Concurrent Reversibility



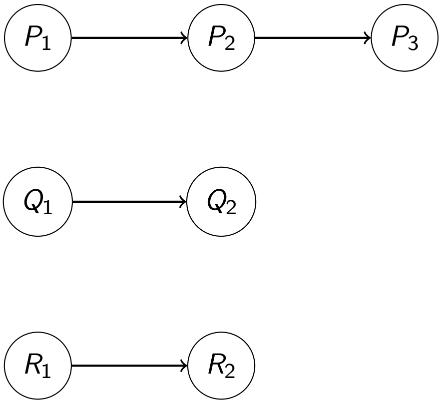
Concurrent Reversibility



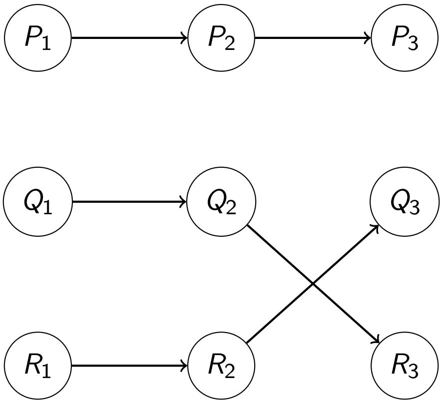
Concurrent Reversibility



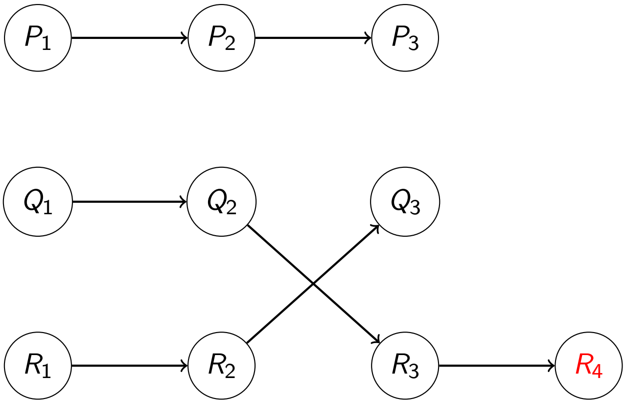
Concurrent Reversibility



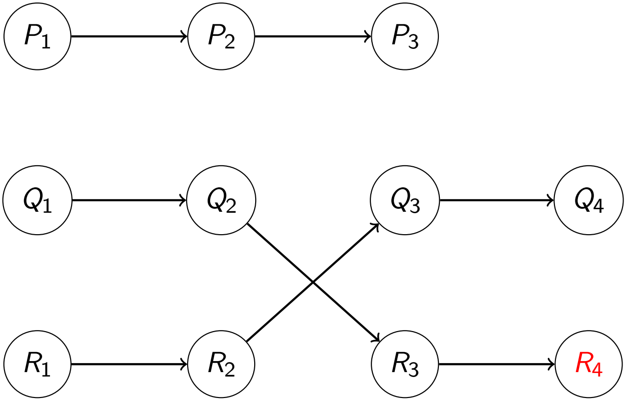
Concurrent Reversibility



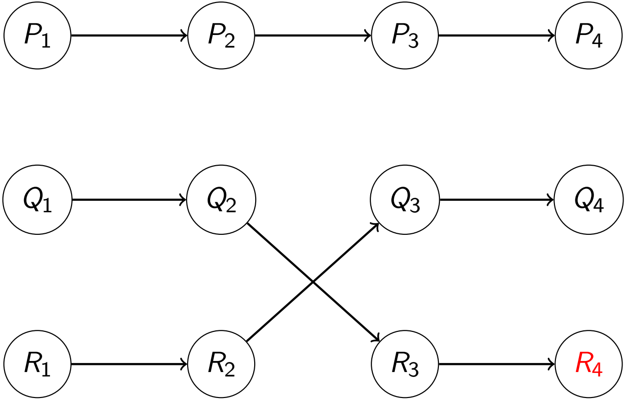
Concurrent Reversibility



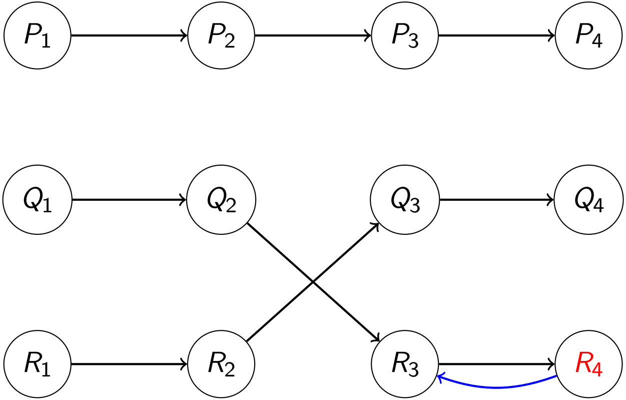
Concurrent Reversibility



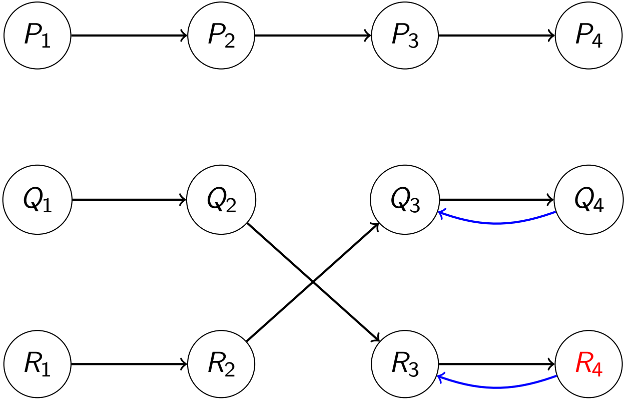
Concurrent Reversibility



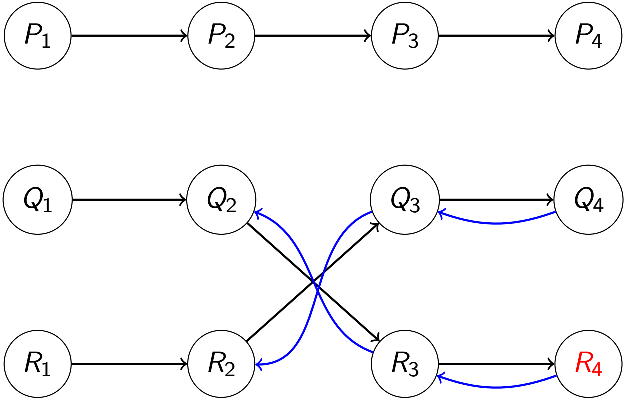
Concurrent Reversibility



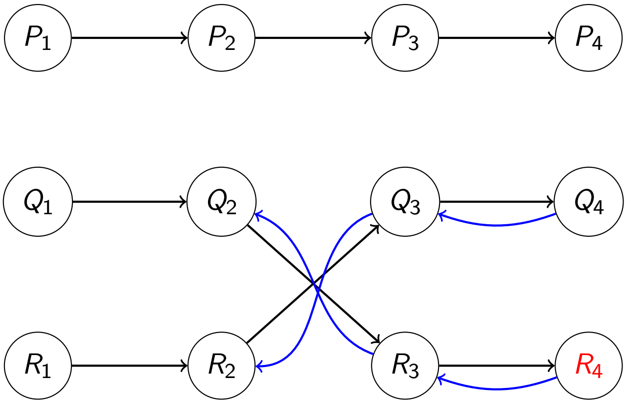
Concurrent Reversibility



Concurrent Reversibility



Concurrent Reversibility



Causal consistency: an action can be undone provided that all of its consequences have been already undone.

Limitations

Reversibility has been investigated in various settings, like ccs, π-calculus, Petri-nets, Erlang, μ-klaim, etc.

Limitations

Reversibility has been investigated in various settings, like ccs, π-calculus, Petri-nets, Erlang, μ-klaim, etc.

The majority of the reversible semantics have always been devised ad-hoc. A process that is error-prone, time-consuming and not scalable.

Limitations

Reversibility has been investigated in various settings, like ccs, π-calculus, Petri-nets, Erlang, μ-klaim, etc.

The majority of the reversible semantics have always been devised ad-hoc. A process that is error-prone, time-consuming and not scalable.

Lanese et al. recently proposed a general method to produce a reversible semantics given a non-reversible one. The pros are symmetric to the cons listed above.

Contributions

The general method proposed by Lanese et al. lacked an implementation which we propose here, by using the Maude programming language.

Contributions

The general method proposed by Lanese et al. lacked an implementation which we propose here, by using the Maude programming language.

Then, we tested it on a novel formalization of Erlang in Maude.

Contributions

The general method proposed by Lanese et al. lacked an implementation which we propose here, by using the Maude programming language.

Then, we tested it on a novel formalization of Erlang in Maude.

Finally, we developed a novel causal-consistent rollback operator on top of the reversible semantics.

Contributions

The general method proposed by Lanese et al. lacked an implementation which we propose here, by using the Maude programming language.

Then, we tested it on a novel formalization of Erlang in Maude.

Finally, we developed a novel causal-consistent rollback operator on top of the reversible semantics.

Hence our three main contributions are:

- A new mechanized formalization of Erlang in Maude
- A concrete implementation in Maude that generates reversible semantics
- A novel generalized rollback operator

Table of Contents

Ingredients

Before diving into the details of our contribution let us discuss the various ingredients required:

- Erlang
- Maude
- The general method

Erlang

The Erlang language

Erlang, developed in 1986 by Ericsson, is a concurrent, distributed, functional programming language, based on message passing.

It is probably the most popular programming language that implements the actor model.

Here we are mostly interested in the main concurrent primitives:

- spawn
- send
- receive

Maude

Maude

Maude is a programming language that efficiently implements conditional rewriting logic.

A rewriting logic theory is a tuple (Σ, E, R) where:

- Σ is a collection of typed operators
- E is a set of equations
- R is a set of rewriting rules

Maude: an example

```
fmod BOOL is
  sort Bool .

  op true : -> Bool .
  op false : -> Bool .
  op _and_ : Bool Bool -> Bool .

  var A : Bool

  eq true and A = A .
  eq false and A = false .
  eq A and A = A .

endfm
```

General Method

Input

The general method takes in input a formalism equipped with a syntax and a reduction semantics

Input

The general method takes in input a formalism equipped with a syntax and a reduction semantics

Then, causal dependencies are captured in terms of resources produced and consumed.

Input

The general method takes in input a formalism equipped with a syntax and a reduction semantics

Then, causal dependencies are captured in terms of resources produced and consumed.

$$\overline{t \to t'}$$

Above t is *consumed* to *produce* t'.

Syntax

The formalism must have a two level syntax. On the lower level there are no constraints, the upper level must be of the following shape.

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

Rules

The rules of the reduction semantics must fit the following schemas.

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

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

Keys and Memories

To make the semantics reversible we resort to the use of keys and memories.

Keys are attached to each entity of the lower level and are used to uniquely identify them.

Memories are produced each time a step forward is taken, they are used to bind two states of the system and to store configurations so that they can be restored later on.

Reversible Syntax

The reversible syntax has the following shape.

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

Forward Rules

The forward reversible rules of the reduction semantics have the following shape.

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

$$\text{(F-SCM-OPN)}\ \frac{R_i \twoheadrightarrow R_i' \quad (\mathtt{keys}(R_i') \setminus \mathtt{keys}(R_i)) \cap (\mathtt{keys}(R_0,\ldots,R_{i-1},R_{i+1},\ldots,R_n) = \emptyset}{op_n(R_0,\ldots,R_i,\ldots,R_n) \twoheadrightarrow op_n(R_0,\ldots,R_i',\ldots,R_n)}$$

$$\text{(F-EQV)}\ \frac{R \equiv_c R' \quad R \twoheadrightarrow R_1 \quad R_1 \equiv_c R_1'}{R' \twoheadrightarrow R_1'}$$

Backward Rules

The backward reversible rules of the reduction semantics have the following shape.

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

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

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

A Concrete Example

(non-reversible rule)
$\langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello \rangle \rightarrow$
$\langle pid : p_1, env : \theta, exp : hello \rangle \mid \langle sender : p_1, receiver : p_2, payload : hello \rangle$

A Concrete Example

(non-reversible rule)
$\langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello \rangle \rightarrow$
$\langle pid : p_1, env : \theta, exp : hello \rangle \mid \langle sender : p_1, receiver : p_2, payload : hello \rangle$

(forward reversible rule)
$k : \langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello \rangle \twoheadrightarrow$
$k_1 : \langle pid : p_1, env : \theta, exp : hello \rangle \mid k_2 : \langle sender : p_1, receiver : p_2, payload : hello \rangle \mid$
$[k : \langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello \rangle \; ; k_1 : \bullet_1 \mid k_2 : \bullet_2]$

A Concrete Example

(non-reversible rule)
$\langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello\rangle \rightarrow$
$\langle pid : p_1, env : \theta, exp : hello\rangle \mid \langle sender : p_1, receiver : p_2, payload : hello\rangle$

(forward reversible rule)
$k : \langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello\rangle \twoheadrightarrow$
$k_1 : \langle pid : p_1, env : \theta, exp : hello\rangle \mid k_2 : \langle sender : p_1, receiver : p_2, payload : hello\rangle \mid$
$[k : \langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello\rangle \; ; k_1 : \bullet_1 \mid k_2 : \bullet_2]$

(backward rule)
$k_1 : \langle pid : p_1, env : \theta, exp : hello\rangle \mid k_2 : \langle sender : p_1, receiver : p_2, exp : hello\rangle \mid$
$[k : \langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello\rangle \; ; k_1 : \bullet_1 \mid k_2 : \bullet_2]$
$\rightsquigarrow k : \langle pid : p_1, env : \theta, exp : p_2 \; ! \; hello\rangle$

Table of Contents

Automatic Generation of the Reversible Semantics

Input Format: Entities

```
mod SYSTEM is
...
sort Sys .
subsort Entity < Sys .

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

Reversible Entities

```
mod SYSTEM is
...
sorts Memory Context Sys .

subsort EntityWithKey Memory Context < Sys .

op @:_ : Key -> Context [ctor] .
op [_;_] : Sys Context -> Memory [ctor frozen .. ] .

op #empty-system : -> Sys [ctor] .
op _||_ : Sys Sys -> Sys [ctor assoc comm .. ] .

...
endm
```

Rewriting Rules: Send

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

Rewriting Rules: Forward and Backward Send

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

Rewriting Rules: Forward and Backward Send

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

crl [bwd sys-send]:
  < sender: P, receiver: DEST, payload: GVALUE > * key(0 L) ||
  < P | exp: EXSEQ', env-stack: ENV', ASET > * key(1 L) ||
  [< P | ASET, exp: EXSEQ, env-stack: ENV > * key(L) ;
  @: key(0 L) || @: key(1 L)] =>
  < P | ASET, exp: EXSEQ, env-stack: ENV > * key(L)
```

Correctness

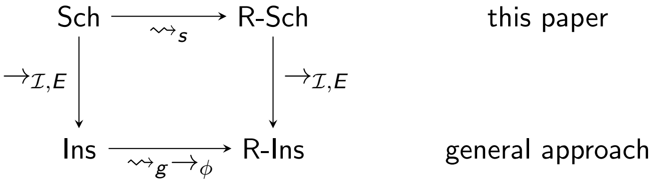


Figure: Schema of the proof of correctness.

Rollback

Causal-consistent rollback semantics allows to undo a past action by undoing only actions that have a causal dependency with it.

Thanks to the uniformity of the reversible semantics produced we were able to build a rollback operator which works on all of them.

Rollback: Idea

1. We pinpoint the action we wish to undo by using the key of one of the entity of such state.

Rollback: Idea

1. We pinpoint the action we wish to undo by using the key of one of the entity of such state.
2. - If such key is contained in the left-hand side of another memory $[R; C]$ then we have found a dependency and we recursively call the procedure on the keys of the C

Rollback: Idea

1. We pinpoint the action we wish to undo by using the key of one of the entity of such state.

2.
   - If such key is contained in the left-hand side of another memory $[R; C]$ then we have found a dependency and we recursively call the procedure on the keys of the C
   - if we cannot find other occurrences of the key then it means that there are no dependencies.

Rollback: Idea

1. We pinpoint the action we wish to undo by using the key of one of the entity of such state.
2. - If such key is contained in the left-hand side of another memory $[R; C]$ then we have found a dependency and we recursively call the procedure on the keys of the C
   - if we cannot find other occurrences of the key then it means that there are no dependencies.
3. Once computed the dependencies it suffices to undo them in a causal-consistent order.

Table of Contents

Future Work

Still many directions to be explored, one could for instance:

- Optimize and improve the implementation of the Erlang semantics
- Introduce support for read dependencies in the general method and in the tool as well to extend the set of semantics *correctly captured*.

The end

Thank you for the attention!

Erlang semantics

A Two Layer Semantics

We implemented the Erlang semantics as a two layer semantics:

- A set of equations for the expression semantics, defined over the tuples `<LABEL, ENV, EXPR>`
- A set of rewriting rules for the system semantics defined over system configurations, i.e., processes running in parallel with messages

Example of Equations

```
eq [match] :
   < REQLABEL, ENVSTACK, GVALUE = GVALUE > =
   < tau, ENVSTACK, GVALUE > .

ceq [receive] :
  < req-receive(PAYLOAD), ENV : ENVSTACK, receive CLSEQ end> =
  < received, ENV' : (ENV : ENVSTACK), begin EXSEQ end>
  if #entityMatchSuccess(EXSEQ | ENV') :=
     #entityMatch(CLSEQ | PAYLOAD | ENV ) .
```

Expression Handling

While managing expressions we need to be careful as a naive handling could cause unwanted effects.

$$pow\_and\_sub(N, M) \rightarrow Z = N * N, Z - M.$$

$$X = pow\_and\_sub(N, M) \Rightarrow_{\text{wrong}} X = Z = N * N, Z - M.$$

$$X = pow\_and\_sub(N, M) \Rightarrow X = \text{begin } Z = N * N, Z - M \text{ end.}$$

Rewriting Rules

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

crl [sys-self] :
      < P | exp: EXSEQ, env-stack: ENV, ASET > =>
      < P | exp: EXSEQ', env-stack: ENV', ASET >
      if < tau, ENV', EXSEQ' > :=
         < self(P), ENV, EXSEQ > .
```

Bug General Approach

Let us consider a configuration:

$$R = \nu b(k_1 : a\langle b\langle P\rangle\rangle \mid k_2 : a(X) \triangleright X)$$

R can reduce to:

$$R_1 = \nu b(j_1 : b\langle P\rangle \mid [k_1 : a\langle b\langle P\rangle\rangle \mid k_2 : a(X) \triangleright X; j_1 : \bullet_1])$$

We can now use naïf projection to α-convert b into c to obtain:

$$R_1 \equiv_n R_2 = \nu c(j_1 : c\langle P\rangle \mid [k_1 : a\langle b\langle P\rangle\rangle \mid k_2 : a(X) \triangleright X; j_1 : \bullet_1])$$

One can notice that occurrences of b inside memories have not been affected, since they were not part of the term to which α-conversion has been applied.