Reversibility in Erlang: Imperative Constructs

Pietro Lami[1] Ivan Lanese[2] Jean-Bernard Stefani[1]
Claudio Sacerdoti Coen[3] Giovanni Fabbretti[1]

[1]Univ. Grenoble Alpes, INRIA, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

[2]Focus Team, Univ. of Bologna, INRIA, 40126 Bologna, Italy

[3]Univ. of Bologna, 40126 Bologna, Italy

RC22, 05/07/2022

Table of Contents

Reversibility

In a sequential system reversibility can be obtained by recursively undoing the last action. This definition is not suitable in concurrent systems, since the last action may not be well-defined.

Causal-consistent reversibility

Any action can be undone provided that all its effects (if any) have been undone before

The idea is that each process saves all the information necessary to restore past states.

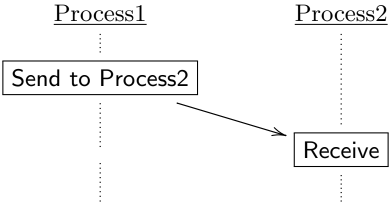
Causal dependencies

There is a dependency between two actions when:

- they cannot be performed in the opposite order
- executing them in the opposite order would change the result

Example (Dependency)

Receiving a message depends on its send.



CauDEr

CauDEr is a reversible causal-consistent debugger for the Erlang programming language.

Erlang is a functional, concurrent and distributed programming language based on the actor paradigm.

CauDEr implements:

- a forward semantics
- a backward semantics
- a rollback semantics

Our Contribution

We extend CauDEr and its underlying theory by adding the support for some imperative primitives that have not been considered before.

From the technical point of view, we show that the dependencies introduced by shared memory are not trivial.

Map

In our extension, atoms and pids are central:

- an atom is a literal constant;
- Pid is an abbreviation for process identifier

In Erlang, a pid can be associated to an atom.

Each Erlang node has a map associating atoms to pids

Both atoms and pids in the map must be **unique**.

Imperative primitives

We extend CauDEr with support for the following built-in Erlang functions (BIFs):
given an atom a and a pid p

- `register`(a, p): inserts the pair $\langle a, p \rangle$ in the map;
- `unregister`(a): removes the (unique) pair $\langle a, p \rangle$ from the map;
- `whereis`(a): returns the associated pid p;
- `registered`$()$: returns the list of all the atoms in the map.

Table of Contents

System definition

We define a reduction semantics with two types of reductions:

Forward
$\Gamma; \Pi; M \rightharpoonup \Gamma'; \Pi'; M'$
stores history information

Backward
$\Gamma; \Pi; M \leftharpoondown \Gamma'; \Pi'; M'$
exploits the stored history information

Definition (System)

A system is a tuple $\Gamma; \Pi; M$.

- Γ is the global mailbox,
- Π is the pool of processes,
- M is the map.

Process definition

Definition (Process)

A process is a tuple $\langle p, h, \theta, e, S \rangle \in \Pi$, where:

- p is the process pid,
- h is the process history,
- θ is the process environment,
- e is the expression under evaluation,
- S is a stack of process environments.

Extended Map

M is a quadruple $\langle a, p, t, s \rangle$ where a and p are the atom-pid registered, t is a unique identifier for the tuple and s can be either $\top$ or $\bot$.

Unique identifiers t are used to distinguish identical tuples existing at different times.

Tuples whose last field is $\top$ match the ones in the standard semantics, we call them *alive* tuples.

Those with $\bot$ are *ghost* tuples, namely alive tuples that have been removed from the map by a past forward action

Causal dependencies

The causal dependencies of the imperative primitives are:

1. between write actions that involve the same atom or pid;
2. between a write action and a read action that involve the same atom or pid;

Example

1. register(a, p) and unregister(a) are causally dependent
2. register(a, p) and whereis(a) are causally dependent

Extended forward and backward semantics

Reg

$$\frac{\theta, e, S \xrightarrow{\mathsf{register}(\kappa, a, p')} \theta', e', S' \qquad t \text{ fresh} \qquad \mathsf{M}_a = \emptyset \qquad \mathsf{M}_{p'} = \emptyset \qquad \mathsf{isAlive}(p', \Pi)}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{regS}(\theta, e, S, \{\langle a, p', t, \top\rangle\}){:}h, \theta', e'\{\kappa \to \mathsf{true}\}, S'\rangle \mid \Pi; \mathsf{M} \cup \{\langle a, p', t, \top\rangle\}}$$

$\overline{\text{Reg}}$

$$\frac{\mathsf{readop}(t, \Pi) = \emptyset}{\Gamma; \langle p, \mathsf{regS}(\theta, e, S, \{\langle a, p', t, \top\rangle\}){:}h, \theta', e', S'\rangle \mid \Pi; \mathsf{M} \cup \{\langle a, p', t, \top\rangle\} \leftharpoondown \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M}}$$

Extended forward and backward semantics

Reg

$$\frac{\theta, e, S \xrightarrow{\text{register}(\kappa, a, p')} \theta', e', S' \qquad t \text{ fresh} \qquad \mathsf{M}_a = \emptyset \qquad \mathsf{M}_{p'} = \emptyset \qquad \mathsf{isAlive}(p', \Pi)}{\Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M} \rightharpoonup \Gamma; \langle p, \mathsf{regS}(\theta, e, S, \{\langle a, p', t, \top\rangle\}){:}h, \theta', e'\{\kappa \to \mathsf{true}\}, S'\rangle \mid \Pi; \mathsf{M} \cup \{\langle a, p', t, \top\rangle\}}$$

$\overline{\text{Reg}}$

$$\frac{\mathsf{readop}(t, \Pi) = \emptyset}{\Gamma; \langle p, \mathsf{regS}(\theta, e, S, \{\langle a, p', t, \top\rangle\}){:}h, \theta', e', S'\rangle \mid \Pi; \boxed{\mathsf{M} \cup \{\langle a, p', t, \top\rangle\}} \leftharpoondown \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathsf{M}}$$

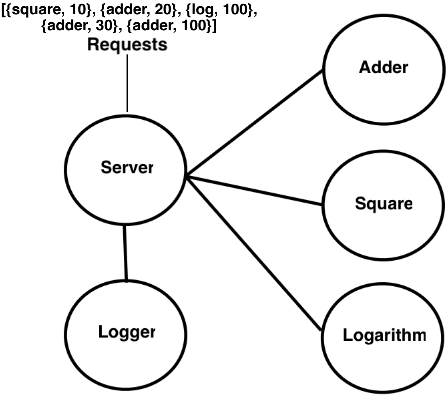
Table of Contents

Case study

We consider a simple server dispatching requests to various mathematical services, and logging the results of the evaluation on a logger.

Each service is a process, and they are created only when there is a first request for them.

The logger keeps track of the values it receives, and answers each request with the sequential number of the element in the log



Log of the server

```
SEND REQUEST:{square,10}
SEND REQUEST:{adder,20}
SEND REQUEST:{log,100}
SEND REQUEST:{adder,30}
LOGGED {square,100} TIME:0
SEND REQUEST:{adder,100}
LOGGED {adder,20} TIME:1
LOGGED {adder,50} TIME:2
```

Happens

```
LOGGED 100 TIME:3
LOGGED {adder,150} TIME:4
```

Expected

```
LOGGED {log,2.0} TIME:3
LOGGED {adder,150} TIME:4
```

We see that the there is an error in the behavior of the program.

History of the server in CauDER

In the server history we can see the send of the logarithm request but we do not have the answer from the service as in the other services (e.g. square). So we can do a rollback of message (11)

```
...
send with atom adder (30,13)
read of [{adder,4,3,top}]
receive({adder,30},3)
send with atom log (100,11)
read of [{log,2,1,top}]
receive({log,100},2)
receive({logged,{{square,100},0}},10)
send with atom log ({square,100},9)
receive({reply,{square,100}},6)
send with atom square (10,4)
receive({square,10},0)
...
```

Bad behavior

After the rollback we arrive at:

```
                ...
case whereis(Atom) of
        undefined ->
                register(Atom, spawn(?MODULE, Atom, [])),
                send(Atom,Val);
        _ ->
                send(Atom,Val)
end
```

We are in the branch where the atom is already registered. But this is the first time that we ask for the logarithm service.

Detect the error

So we can rollback the registration of atom log and we arrive at:

```
register(server, spawn(?MODULE, server, [])),
register(log, spawn(?MODULE, logger, [0, []])),
```

We understand that we use the same atom to identify two different processes.
Then the behavior is incorrect because we use the atom log for the logarithm service but also to send messages to the logger

$$\text{log (logarithm)} \neq \text{log (logger)}$$

Screenshot

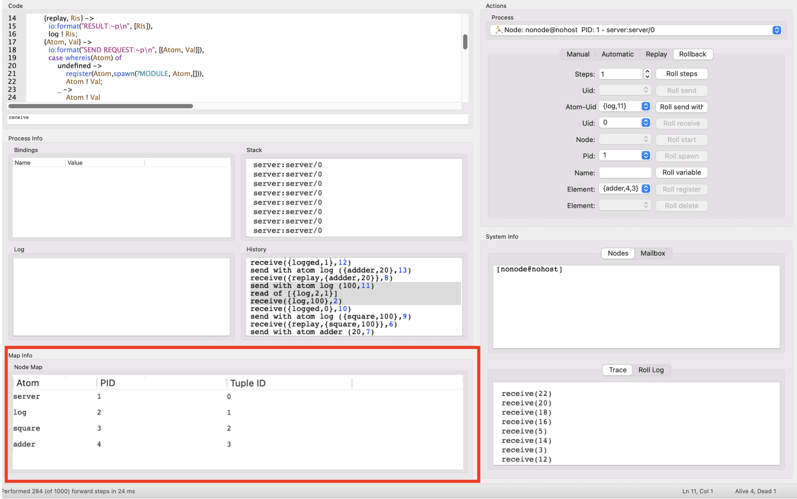

Code
{replay, Ris} ->
io:format("RESULT:~p\n", [Ris]),
log ! Ris;
{Atom, Val} ->
io:format("SEND REQUEST:~p\n", [{Atom, Val}]),
case whereis(Atom) of
undefined ->
register(Atom,spawn(?MODULE, Atom,[])),
Atom ! Val;
_ ->
Atom ! Val
receive
Process Info
Bindings
Name
Value
Stack
server:server/0
server:server/0
server:server/0
server:server/0
server:server/0
server:server/0
server:server/0
server:server/0
Log
History
receive({logged,1},12)
send with atom log ({addder,20},13)
receive({replay,{adder,20}},8)
send with atom log (100,11)
read of [{log,2,1}]
receive({log,100},2)
receive({logged,0},10)
send with atom log ({square,100},9)
receive({replay,{square,100}},6)
send with atom adder (20,7)
Map Info
Node Map
Atom
PID
Tuple ID
server
1
0
log
2
1
square
3
2
adder
4
3
Actions
Process
Node: nonode@nohost PID: 1 - server:server/0
Manual
Automatic
Replay
Rollback
Steps: 1
Roll steps
Uid:
Roll send
Atom-Uid {log,11}
Roll send with
Uid: 0
Roll receive
Node:
Roll start
Pid: 1
Roll spawn
Name:
Roll variable
Element: {adder,4,3}
Roll register
Element:
Roll delete
System Info
Nodes
Mailbox
[nonode@nohost]
Trace
Roll Log
receive(22)
receive(20)
receive(18)
receive(16)
receive(5)
receive(14)
receive(3)
receive(12)
Performed 284 (of 1000) forward steps in 24 ms
Ln 11, Col 1
Alive 4, Dead 1

Table of Contents

Future work

- in some cases sequences of transitions would commute, but their composing transitions do not;
- currently this creates dependencies since commuting is possible only between single transitions

Example

A `registered()` operation does not commute with `register(a,_)` followed by `unregister(a)`.

Thank you for the attention

Imperative primitives

We extend CauDEr with support for the following built-in Erlang functions (BIFs):

- `register`: given an atom a and a pid p, it inserts the pair $\langle a, p \rangle$ in the map and returns the atom **true**; if it is not possible an exception is raised;
- `unregister`: given an atom a, it removes the (unique) pair $\langle a, p \rangle$ from the map and returns **true** if the atom a is found, raises an exception otherwise;
- `whereis`: given an atom, it returns the associated pid if it exists, the atom **undefined** otherwise;
- `registered`: returns a list (possibly empty) of all the atoms in the map.

Example (Register followed by delete)

Consider a process P doing a `registered` operation and another process, Q, doing a (successful) `register` followed by a delete operation of the same tuple.

Executing P first and the Q or vice versa would lead to the same state.

We want to distinguish these two computations, since undoing the `registered` should not always be possible.

Ghost tuples are our solution to this problem.

Rollback semantics

$$(U - Satisfy)\frac{\mathcal{S} \leftharpoondown_{p,r,\Psi'} \mathcal{S}' \;\land\; \psi \in \Psi'}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\Psi}}$$

$$(U - Act)\;\frac{\mathcal{S} \leftharpoondown_{p,r,\Psi'} \mathcal{S}' \;\land\; \{p,r\} \notin \Psi'}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\{p,\psi\}:\Psi}}$$

$$(Request)\;\frac{\begin{array}{c}\mathcal{S} = \Gamma; \langle p, h, \theta, e, S\rangle \mid \Pi; \mathrm{M} \;\land\; \mathcal{S} \not\leftharpoondown_{p,r,\Psi'} \;\land \\ \{p',\psi'\} = dep(\langle p, h, \theta, e, S\rangle, \mathcal{S})\end{array}}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\{p',\psi'\}:\{p,\psi\}:\Psi}}$$

Dependencies operator

$$
\begin{aligned}
&\mathrm{dep}(<\_, \mathrm{send}(\_, \_, \_, \{v, \lambda\}){:}h, \_, \_, \_>, \Gamma \cup \{(p, p', \{v, \lambda\})\}; \_; \_) &&= \{p', \lambda^{\Downarrow}\}\\
&\mathrm{dep}(< p, \mathrm{sendS}(\_, \_, \_, \{v, \lambda\}, \_){:}h, \_, \_, \_>, \Gamma \cup \{(p, p', \{v, \lambda\})\}; \_; \_) &&= \{p', \lambda^{\Downarrow}\}\\
&\mathrm{dep}(<\_, \mathrm{sendS}(\_, \_, \_, \_, \{\langle a, p, t, \top\rangle\}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}') &&= \{p', \mathrm{del}(t)\}\\
&\mathrm{dep}(<\_, \mathrm{spawn}(\_, \_, \_, p'){:}h, \_, \_, \_>, \_; \Pi; \_; \_) &&= \{p', s\}\\
&\mathrm{dep}(<\_, \mathrm{readS}(\_, \_, \_, \mathrm{M} \cup \{\langle a, \_, t, \top\rangle\}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}') &&= \{p', \mathrm{del}(t)\}\\
&\mathrm{dep}(<\_, \mathrm{readS}(\_, \_, \_, \mathrm{M} \cup \{\langle \_, p, t, \top\rangle\}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}') &&= \{p', \mathrm{del}(t)\}\\
&\mathrm{dep}(<\_, \mathrm{readF}(\_, \_, \_, a, \mathrm{M}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}' \cup \{\langle a, \_, t, \_\rangle\}) &&= \{p', \mathrm{regS}(t)\}\\
&\mathrm{dep}(<\_, \mathrm{readM}(\_, \_, \_, \mathrm{M}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}' \cup \{\langle \_, \_, t, \top\rangle\}) &&= \{p', \mathrm{regS}(t)\}\\
&\mathrm{dep}(<\_, \mathrm{readM}(\_, \_, \_, \mathrm{M}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}' \cup \{\langle \_, \_, t, \bot\rangle\}) &&= \{p', \mathrm{del}(t)\}\\
&\mathrm{dep}(<\_, \mathrm{regS}(\_, \_, \_, \{\langle \_, \_, t, \top\rangle\}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}' \cup \{\langle \_, \_, t, \bot\rangle\}) &&= \{p', \mathrm{del}(t)\}\\
&\mathrm{dep}(<\_, \mathrm{regS}(\_, \_, \_, \{\langle \_, \_, t, \top\rangle\}){:}h, \_, \_, \_>, \_; \_; \_) &&= \{p', \mathrm{read}(t)\}\\
&\mathrm{dep}(<\_, \mathrm{del}(\_, \_, \_, \{\langle a, \_, t, \top\rangle\}, \mathrm{M}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}' \cup \{\langle a, \_, t_a, \_\rangle\}) &&= \{p', \mathrm{regS}(t_a)\}\\
&\mathrm{dep}(<\_, \mathrm{del}(\_, \_, \_, \{\langle \_, p, t, \top\rangle\}, \mathrm{M}){:}h, \_, \_, \_>, \_; \_; \mathrm{M}' \cup \{\langle \_, p, t_p, \_\rangle\}) &&= \{p', \mathrm{regS}(t_p)\}\\
&\mathrm{dep}(<\_, \mathrm{del}(\_, \_, \_, \{\langle a, p, t, \top\rangle\}, \_){:}h, \_, \_, \_>, \_; \_; \mathrm{M}') &&= \{p', \mathrm{readfail}(t)\}
\end{aligned}
$$