Causal-Consistent Debugging of Distributed Erlang Programs

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

[1]SPADES Team, INRIA

[2]University of Bologna/FOCUS Team, INRIA

Reversible Computation, 07-07-2021

Introduction

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

Distinctive features of CauDEr:

- Reversibility of systems composed by several processes
- Causal-consistent rollback

Our goal: extend CauDEr with support for distribution

Motivations

Concurrent and distributed systems are everywhere and both are well know for their intrinsic difficulties.

Hence we need effective tools while writing code.

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.

Erlang owes its success to three aspects: the support of concurrency and distribution, the facilities to do error-handling and the OTP libraries.

Reversibility

Causal Consistency: before undoing an action we must ensure that all of its consequences, if any, have been undone.

Each process embeds a memory that contains information about past states of its computation.

Our contribution

CauDEr, in its first version, addressed the functional and concurrent fragment of the Erlang language.

This work extends CauDEr and the theory behind it with the support for distributed programs.

Table of Contents

1 Introduction

2 Background

3 Distributed CauDEr

4 Future work

A deeper look inside CauDEr

CauDEr implements three semantics:

- a forward semantics that defines Erlang's behavior and stores information in the history
- a backward semantics that ensures that we undo only actions whose consequences have been already undone
- a rollback semantics which automatically undoes all the consequences of an action

Semantics structure

Definition (System)

A system is defined as $\Gamma; \Pi$ where

- Γ is the global mailbox
- Π is a pool of processes

Definition

A process is defined as: $\langle p, h, \theta, e, q \rangle$ where

- p is the process id
- h is the process history
- θ is the process environment
- e is the expression to evaluate
- q is the local mailbox

Causal dependencies

We say that there is a dependency between two actions in two cases:

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

(A simplification of) The dependencies that rose in the functional and concurrent fragment of the language are that a receive of a message depends on its send.

Table of Contents

1 Introduction

2 Background

3 Distributed CauDEr

4 Future work

The distributed primitives

The three primitives for distributed computing that we support are the following:

- slave:start/{1,2} starts a slave node
- erlang:node/0 returns the name of the local node
- erlang:nodes/0 returns the names of the connected nodes

With the introduction of nodes we also updated the spawn, which now takes an extra parameter that represents the node where it must be performed.

Causal dependencies

Informally, one could summarise the causal dependencies of the distributed primitives as follow:

1. every action of process p depends on the (successful) spawn of p;
2. a (successful) spawn on node *nid* depends on the start of *nid*;
3. a (successful) start of node *nid* depends on previous failed spawns on the same node, if any (if we swap the order, the spawn will succeed);
4. a failed start of node *nid* depends on its (successful) start;
5. a nodes reading a set Ω depends on the start of all nids in Ω, if any.

Semantics structure

Definition

A system is defined as $\Gamma; \Pi; \Omega$ where

- Ω is the set of connected nodes
- the other elements are as before

Definition

A process is defined as: $\langle nid, p, h, \theta, e, q \rangle$ where

- *nid* is the id of the node where the process is running
- the other elements are as before

Extended forward and backward semantics

$$(\mathit{StartS}) \quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, \mathit{nid}')} \theta', e' \quad \mathit{nid}' \notin \Omega}{\boxed{\Gamma; \langle \mathit{nid}, p, h, \theta, e\rangle \mid \Pi; \Omega} \rightharpoonup_{p, \mathsf{start}(\mathit{nid}'), \{s, st_{\mathit{nid}'}\}} \Gamma; \langle \mathit{nid}, p, \mathsf{start}(\theta, e, \mathsf{succ}, \mathit{nid}') : h, \theta', e'\{\kappa \mapsto \mathit{nid}'\}\rangle \mid \Pi; \{\mathit{nid}'\} \cup \Omega}$$

$$(\overline{\mathit{StartS}}) \quad \begin{array}{c} \Gamma; \langle \mathit{nid}, p, \mathsf{start}(\theta, e, \mathsf{succ}, \mathit{nid}') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{\mathit{nid}'\} \\ \leftharpoondown_{p, \mathsf{start}(\mathit{nid}'), \{s, st_{\mathit{nid}'}\}} \Gamma; \langle \mathit{nid}, p, h, \theta, e\rangle \mid \Pi; \Omega \\ \text{if } \mathit{spawns}(\mathit{nid}', \Pi) = [\,] \land \mathit{reads}(\mathit{nid}', \Pi) = [\,] \land \mathit{failed\_starts}(\mathit{nid}', \Pi) = [\,] \end{array}$$

Extended forward and backward semantics

$$(\mathit{StartS})\quad \frac{\boxed{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}}{\begin{array}{c}\Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \\ \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}\rangle \mid \Pi; \{nid'\} \cup \Omega\end{array}}$$

$$(\overline{\mathit{StartS}})\quad \begin{array}{c}\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{nid'\} \\ \leftharpoondown_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \\ \text{if } \mathit{spawns}(nid', \Pi) = [\,] \land \mathit{reads}(nid', \Pi) = [\,] \land \mathit{failed\_starts}(nid', \Pi) = [\,]\end{array}$$

Extended forward and backward semantics

$$(\mathit{StartS}) \quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}{\Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p, \mathsf{start}(nid'), \{s, st_{nid'}\}} \boxed{\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}\rangle \mid \Pi; \{nid'\} \cup \Omega}}$$

$$(\overline{\mathit{StartS}}) \quad \begin{array}{c} \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{nid'\} \\ \leftharpoondown_{p, \mathsf{start}(nid'), \{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \\ \text{if } \mathit{spawns}(nid', \Pi) = [\,] \wedge \mathit{reads}(nid', \Pi) = [\,] \wedge \mathit{failed\_starts}(nid', \Pi) = [\,] \end{array}$$

Extended forward and backward semantics

$$(StartS)\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, nid')} \theta', e' \quad nid' \notin \Omega}{\Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\{\kappa \mapsto nid'\}\rangle \mid \Pi; \{nid'\} \cup \Omega}$$

$$(\overline{StartS})\quad \boxed{\Gamma; \langle nid, p, \mathsf{start}(\theta, e, \mathsf{succ}, nid') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{nid'\}}$$
$$\leftharpoondown_{p,\mathsf{start}(nid'),\{s, st_{nid'}\}} \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega$$
$$\text{if } \mathit{spawns}(nid', \Pi) = [\,] \wedge \mathit{reads}(nid', \Pi) = [\,] \wedge \mathit{failed\_starts}(nid', \Pi) = [\,]$$

Extended forward and backward semantics

$$(\mathit{StartS})\quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, \mathit{nid}')} \theta', e' \quad \mathit{nid}' \notin \Omega}{\begin{array}{c}\Gamma; \langle \mathit{nid}, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p, \mathsf{start}(\mathit{nid}'), \{s, st_{\mathit{nid}'}\}} \\ \Gamma; \langle \mathit{nid}, p, \mathsf{start}(\theta, e, \mathsf{succ}, \mathit{nid}') : h, \theta', e'\{\kappa \mapsto \mathit{nid}'\}\rangle \mid \Pi; \{\mathit{nid}'\} \cup \Omega\end{array}}$$

$$(\overline{\mathit{StartS}})\quad \begin{array}{c}\Gamma; \langle \mathit{nid}, p, \mathsf{start}(\theta, e, \mathsf{succ}, \mathit{nid}') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{\mathit{nid}'\} \\ \leftharpoondown_{p, \mathsf{start}(\mathit{nid}'), \{s, st_{\mathit{nid}'}\}} \Gamma; \langle \mathit{nid}, p, h, \theta, e\rangle \mid \Pi; \Omega \\ \text{if } \boxed{\mathit{spawns}(\mathit{nid}', \Pi) = [\,] \land \mathit{reads}(\mathit{nid}', \Pi) = [\,] \land \mathit{failed\_starts}(\mathit{nid}', \Pi) = [\,]}\end{array}$$

Extended forward and backward semantics

$$(\mathit{StartS}) \quad \frac{\theta, e \xrightarrow{\mathsf{start}(\kappa, \mathit{nid}')} \theta', e' \quad \mathit{nid}' \notin \Omega}{\Gamma; \langle \mathit{nid}, p, h, \theta, e\rangle \mid \Pi; \Omega \rightharpoonup_{p,\mathsf{start}(\mathit{nid}'),\{s, st_{\mathit{nid}'}\}} \Gamma; \langle \mathit{nid}, p, \mathsf{start}(\theta, e, \mathsf{succ}, \mathit{nid}') : h, \theta', e'\{\kappa \mapsto \mathit{nid}'\}\rangle \mid \Pi; \{\mathit{nid}'\} \cup \Omega}$$

$$(\overline{\mathit{StartS}}) \quad \begin{array}{c} \Gamma; \langle \mathit{nid}, p, \mathsf{start}(\theta, e, \mathsf{succ}, \mathit{nid}') : h, \theta', e'\rangle \mid \Pi; \Omega \cup \{\mathit{nid}'\} \\ \leftharpoondown_{p,\mathsf{start}(\mathit{nid}'),\{s, st_{\mathit{nid}'}\}} \boxed{\Gamma; \langle \mathit{nid}, p, h, \theta, e\rangle \mid \Pi; \Omega} \\ \text{if } \mathit{spawns}(\mathit{nid}', \Pi) = [\,] \wedge \mathit{reads}(\mathit{nid}', \Pi) = [\,] \wedge \mathit{failed\_starts}(\mathit{nid}', \Pi) = [\,] \end{array}$$

Rollback Semantics

The rollback semantics allows us to reach a past state of the computation of the system, such past state is specified as an action performed by a process.

Some of the considered requests are:

- $\{p, \lambda^{\Downarrow}\}$: the receive of the message uniquely identified by λ;
- $\{p, st_{nid}\}$: the successful start of node nid;
- $\{p, sp_{p'}\}$: the spawn of process p'.

A system in rollback mode is denoted as $\llbracket \mathcal{S} \rrbracket_{\Psi}$

Rollback Semantics

$$\frac{\mathcal{S} \leftharpoondown_{p,r,\Psi'} \mathcal{S}' \;\wedge\; \psi \in \Psi'}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\Psi}} \qquad \frac{\mathcal{S} \leftharpoondown_{p,r,\Psi'} \mathcal{S}' \;\wedge\; \psi \notin \Psi'}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\{p,\psi\}:\Psi}}$$

$$\frac{\mathcal{S} = \Gamma; \langle nid, p, h, \theta, e\rangle \mid \Pi; \Omega \;\wedge\; \mathcal{S} \not\leftharpoondown_{p,r,\Psi'} \;\wedge\; \{p', \psi'\} = \mathsf{bwd\_dep}(\langle nid, p, h, \theta, e\rangle, \mathcal{S})}{[\![\mathcal{S}]\!]_{\{p,\psi\}:\Psi} \rightsquigarrow [\![\mathcal{S}']\!]_{\{p',\psi'\}:\{p,\psi\}:\Psi}}$$

Rollback semantics: dependencies operator

The dependencies operator does pattern matching on the history item and given the system computes the request to undo the consequences.

$$\text{bwd\_dep}(< \_, \_, \text{nodes}(\_, \_, \Omega') : h, \_, \_ >, \_; \Pi; \{nid'\} \cup \Omega) \quad = \{parent(nid', \Pi), st_{nid'}\}$$
where $nid' \notin \Omega'$

Table of Contents

1 Introduction

2 Background

3 Distributed CauDEr

4 Future work

Future work

Still many features of Erlang that are to be covered (monitors, links, errors, etc.) to have a debugger that can be used on real programs.

Another line of research worth being investigated is the study of automatic ways to derive a reversible semantics starting from a non-reversible one.

The end

Thank you for the attention!