Indice

Capitolo 1

Introduzione

Erlang è un linguaggio di programmazione funzionale e concorrente basato sul paradigma ad attori (concorrenza basata sul *message-passing*).
BEAM è la macchina virtuale al centro di Erlang, fa parte dell'Erlang Run-Time System (ERTS), che compila il codice sorgente di Erlang in bytecode, che viene quindi eseguito su BEAM. Core Erlang [1] è uno dei linguaggi intermedi in cui Erlang viene compilato.

In questa tesi si mostra un estensione della semantica descritta nell'articolo "A theory of reversibility for Erlang" [2] con delle feature imperative.

Nella Sezione 2.1 viene modificato il linguaggio presentato nell'articolo [2] aggiungendo le operazione di registrazione di atomi e pid, deregistrazione di pid e di lookup, inoltre viene aggiunto un non terminale *end* per poter identificare quando un processo è arrivato in uno stato finale.

Nella Sezione 2.2 viene modificata la nozione di sistema data nell'articolo [2] in cui viene introdotta una mappa formata dalle coppie di atomi-pid che sono stati registrati. Inoltre sono state introdotte le regole per la semantica del linguaggio, in questo caso sono state aggiunte le regole per la valutazione delle nuove espressioni e inoltre sono state aggiunte le regole per descrivere il fallimento di un'espressione, che nell'articolo [2] non erano presenti. Inoltre sono state aggiunte le regole di sistema per le nuove espressioni e per i casi di fallimento.

Nella Sezione 2.3 viene descritta la semantica reversibile divisa in semantica *forward* e *backward*. La prima è un'estensione delle regole introdotte precedentemente in cui nell'articolo [2] viene aggiunta una storia dei processi mentre in questo lavoro è stata l'aggiunta di una storia della mappa. La semantica *backward* invece descrive come viene fatto un passo all'indietro. In questo caso viene introdotta una nozione di operazione in lettura e scrittura per generalizzare le *side-condition* delle regole inserite e per dare una nozione di operazione concorrente più generale.

Nella Sezione 2.4 è stato esteso l'operatore di reversibilità che può essere utilizzato per annullare le azioni di un dato processo fino a raggiungere un determinato punto di controllo, introdotto dal programmatore. Per garantire la coerenza causale, l'azione di rollback potrebbe essere propagata ad altri processi dipendenti.

1.1 Reversibilità

Normalmente la computazione avviene in una direzione, in un programma le istruzioni vengono eseguite in un determinato ordine, per computazione reversibile ci si riferisce alla possibilità di eseguire un programma sia in avanti (o anche computazione *forward*) che indietro (o anche computazione *backward*).

Dato un programma non è ovvio che sia possibile eseguire una computazione backward, infatti tutti i linguaggi di programmazione tradizionali consentono operazioni con perdita di informazione.
Durante l'esecuzione di un programma, a meno che non sia reversibile o senza perdita di informazioni, i dati intermedi vengono persi mentre viene calcolato l'output finale.

Esempio 1. *L'assegnamento* `x = 96` *elimina il vecchio valore di* `x` *che deve essere memorizzato se si vuole annullare questa assegnazione.*

Landauer [3] notò anche che ogni computazione irreversibile può essere trasformata in una computazione reversibile, includendola in una computazione più ampia nella quale nessuna informazione viene persa, salvando ogni volta gli stati intermedi della computazione così da non avere perdita di informazione.
L'idea alla base di questo lavoro [3] è che qualsiasi linguaggio di programmazione o formalismo può essere reso reversibile aggiungendo la cronologia del calcolo a ogni stato, questo metodo è di solito chiamato incorporamento di Landauer.

Questa idea fu ulteriormente migliorata da Bennett [4] al fine di evitare la generazione di dati "spazzatura", applicando una serie di analisi al fine di limitare il più possibile le informazioni richieste nella storia.

La reversibilità in un contesto sequenziale è facile da capire, per invertire l'esecuzione di un programma sequenziale è sufficiente annullare ricorsivamente l'ultima azione eseguita dal programma. La definizione di reversibilità in un contesto in cui vengono considerati anche i sistemi distribuiti è più complicata, poiché non esiste il concetto di "ultima azione", dato che, molte azioni vengono eseguite contemporaneamente.

Una definizione adeguata di reversibilità in uno scenario concorrente è stata proposta da Danos e Krivine nel loro articolo [5]. Intuitivamente, la definizione afferma che qualsiasi azione può essere annullata a condizione che tutte le sue eventuali conseguenze siano annullate preventivamente.

Il calcolo reversibile potrebbe essere applicato per migliorare il modo in cui alcune attività vengono risolte, per esempio nel debug.

1.2 Debugger reversibile

I Debugger non sono altro che programmi, che consento di analizzare se un programma è sintatticamente corretto, in modo tale da effettuare una ricerca del bug in modo più veloce e accurato possibile. Grazie ai Debugger si ha la possibilità di scovare errori o malfunzionamenti all'interno del programma sfruttando funzioni specifiche per il debugging: l'attività che consiste proprio nell'individuazione della porzione di software affetta da bug.

Generalmente il tipo di debugger più utilizzato è il debugger a runtime che consente la ricerca dei bug tramite funzionalità standard come *breackpoint*, *controlpoint* e *viste da watch*, che consentono al programmatore di analizzare più accuratamente determinate parti di codice per identificare gli errori con una scansione in avanti del codice.

Di contro però, la maggior parte dei debugger fornisce (infatti), un'assistenza limitata all'esecuzione in avanti del codice per la navigazione temporale, per cui i programmatori devono spesso ricorrere alla simulazione dell'esecuzione del programma mentalmente per cercare di immaginare i flussi di istruzioni che vengono eseguiti. Infatti, con un debugger a *runtime*, il programmatore può cercare il bug inserendo dei breackpoint all'interno dei thread e sperare di scovare il bug facendo dei tentativi, magari entrando in watch per osservare come vengono modificate le variabili durante l'esecuzione del programma. A tal proposito sono nati anche debug più complessi i cosidetti: debugger reversibili che a differenza dei debugger tradizionali, consentono agli sviluppatori di registrare le attività del programma in esecuzione, per poi riavvolgere e riprodurre tali istruzioni, compresi eventuali errori, per ispezionare lo stato del programma, questi debugger sono molto utili negli scenari concorrenti [7].
Riavvolgere le azioni di un particolare processo significa anche annullare ogni conseguenza dell'operazione (cioè tutte le azioni "collegate" a quell'operazione) che si vuole annullare. Questa nozione è chiamata *consistenza causale* [5].

Capitolo 2

Estensione con feature imperative

Nel seguito di questo capitolo, verrà presentata la sintassi del linguaggio esteso supportato, seguita dalla semantica (reversibile). In contrasto con la semantica descritta nell'articolo "A theory of reversibility for Erlang" [2] quella che verrà presentata qui includerà funzioni imperative integrate. Le modifiche effettuate all'articolo [2] sono evidenziate in giallo.

2.1 Sintassi del linguaggio

In questa sezione, verrà mostrata la sintassi di un linguaggio funzionale di prim'ordine, concorrente e distribuito basato sul paradigma ad attori utilizzato. Questo linguaggio è un sotto insieme di Core Erlang [1], che è uno dei linguaggi intermedi in cui un programma Erlang viene compilato.

Nella Figura 2.1 viene presentata la sintassi del linguaggio, si può notare che vengono considerate solo espressioni di prim'ordine quindi il primo argomento delle applicazioni di funzioni e della spawn è un nome di funzione (invece che un'espressione o chiusura arbitraria) e il primo argomento nelle chiamate è un'operazione built-in *Op*. Rispetto alla sintassi precedente sono state aggiunte le funzioni built-in:

- whereis che dato in input un atomo restituisce il pid associato, se non è registrato l'atomo **undefined**;
- registered che restituisce una lista di tutti gli atomi nella mappa, se non sono presenti atomi registrati restituisce una lista vuota;

sono anche state aggiunte le funzioni:

$$
\begin{array}{rcl}
module & ::= & \mathsf{module}\ Atom = fun_1\ \ldots\ fun_n \\
fun & ::= & fname = \mathsf{fun}\ (Var_1, \ldots, Var_n) \rightarrow expr \\
fname & ::= & Atom / Integer \\
lit & ::= & Atom \mid Integer \mid Float \mid Pid \mid [\,] \\
expr & ::= & Var \mid fname \mid [expr_1 | expr_2] \mid \{expr_1, \ldots, expr_n\} \\
 & \mid & \mathsf{call}\ Op\ (expr_1, \ldots, expr_n) \mid \mathsf{apply}\ fname\ (expr_1, \ldots, expr_n) \\
 & \mid & \mathsf{case}\ expr\ \mathsf{of}\ clause_1; \ldots; clause_m\ \mathsf{end} \\
 & \mid & \mathsf{let}\ Var = expr_1\ \mathsf{in}\ expr_2 \mid \mathsf{receive}\ clause_1; \ldots; clause_n\ \mathsf{end} \\
 & \mid & \mathsf{spawn}(fname, [expr_1, \ldots, expr_n]) \mid expr\,!\,expr \mid \mathsf{self}() \\
 & \mid & \mathsf{register}(expr, expr) \mid \mathsf{unregister}(expr) \mid end \\
clause & ::= & pat\ \mathsf{when}\ expr_1 \rightarrow expr_2 \\
pat & ::= & Var \mid lit \mid [pat_1 | pat_2] \mid \{pat_1, \ldots, pat_n\} \\
Op & ::= & \cdots \mid \mathsf{whereis} \mid \mathsf{registered} \\
end & ::= & lit \mid [end_1 | end_2] \mid \{end_1, \ldots, end_n\}
\end{array}
$$

Figura 2.1: Regole di sintassi del linguaggio

- register che dati in input un atomo e un pid inserisce la in una mappa la coppia atomo,pid e restituisce l'atomo **true**. Altrimenti se l'atomo o il pid sono già presenti nella mappa il processo fallisce;

- unregister che dato in input un atomo toglie dalla mappa la coppia atomo,pid e restituisce l'atomo **true**. Altrimenti se l'atomo non è presente nella mappa il processo fallisce.

Inoltre è stato creato il non terminale *end* per riuscire a determinare quando un processo è in uno stato finale, così è possibile determinare se un processo è fallito. In questo modo si è riusciti ad avere un comportamento il più fedele possibile a quello di Erlang nel caso delle funzioni aggiunte. Nelle regole che verranno descritte successivamente il simbolo ϵ rappresenterà un non terminale *end*.

2.2 Semantica del linguaggio

In questa sezione descriveremo formalmente la semantica del linguaggio presentato nella sezione 2.1.

Definizione 2 (Processo). Un processo è indicato da una tupla $\langle p, (\theta, e), q \rangle$ dove p è il pid del processo (ed è unico), (θ, e) è il controllo, che consiste di un ambiente (una sostituzione) e di un'espressione da valutare, e q è la casella

di posta del processo, una coda FIFO con la sequenza di messaggi che sono stati inviati al processo.

Consideriamo le seguenti operazioni sulle cassette postali locali. Dato un messaggio v e una casella di posta locale q, $v : q$ denota una nuova casella di posta con il messaggio v sopra (cioè v è il messaggio più recente). Indichiamo anche con $q \backslash\backslash v$ una nuova coda che risulta da q rimuovendo l'occorrenza più vecchia del messaggio v (che non è necessariamente il messaggio più vecchio nel coda). □

Un *sistema* in esecuzione può quindi essere visto come un insieme di processi, che definiamo formalmente come segue:

Definizione 3 (Sistema). Un sistema è indicato da $\Gamma; \Pi; \mathsf{M}$, dove Γ, la *casella di posta globale*, è un insieme di coppie nella forma (*destinazione*, *messaggio*) e Π è un insieme di processi, indicato da un'espressione della forma

$$\langle p_1, (\theta_1, e_1), q_1 \rangle \mid \cdots \mid \langle p_n, (\theta_n, e_n), q_n \rangle$$

dove " | " denota un operatore associativo e commutativo. Data una casella di posta globale Γ, $\Gamma \cup \{(p, v)\}$ denota una nuova casella di posta che include anche la coppia (p, v), usiamo "$\cup$" come unione multiset. Spesso denotiamo un sistema con un'espressione nella forma $\Gamma; \langle p, (\theta, e), q \rangle \mid \Pi; \mathsf{M}$ per far notare che $\langle p, (\theta, e), q \rangle$ è un processo arbitrario del pool (grazie al fatto che " | " è associativo e commutativo). Infine M rappresenta l'insieme delle coppie $(atomo, pid)$ registrate, indicheremo con "$\cup$" l'aggiunta di una coppia alla mappa e con "$\backslash\backslash$" la rimozione di una coppia dalla mappa. □

Intuitivamente, Γ memorizza i messaggi dopo che sono stati inviati e prima che vengano inseriti nella casella di destinazione, quindi rappresenta i messaggi che si trovano nella rete.

La semantica è definita tramite due relazione di transizione: $\longrightarrow$ per le espressioni e $\hookrightarrow$ per il sistema. Mostreremo prima la relazione di transizione etichettata

$$\longrightarrow : (Env, Exp) \times Label \times (Env, Exp)$$

dove Env e Exp rappresentano rispettivamente l'ambiente (cioè le sostituzioni) e le espressioni mentre $Label$ denota un elemento dell'insieme

$$\{\tau, \mathsf{send}(v_1, v_2), \mathsf{rec}(\kappa, \overline{cl_n}), \mathsf{spawn}(\kappa, a/n, [\overline{v_n}]), \mathsf{self}(\kappa), \mathsf{register}(\kappa, a, p), \mathsf{unregister}(\kappa, a), \tau\mathsf{MM}', \bot\}$$

Verrà utilizzato ℓ per indicare un'etichetta fra quelle appena indicate.
Per chiarezza le regole di transizione della semantica verranno divise in quattro insiemi: quelle per le espressioni sequenziali sono raffigurate nella Figura 2.2, quelle per i fallimenti delle espressioni sequenziali che si trovano nella

Figura 2.3, quelle per le espressioni concorrenti sono nella Figura 2.4 e infine quelle per i fallimenti delle espressioni concorrenti che sono nella Figura 2.5.

$$(Var)\ \frac{}{\theta, X \xrightarrow{\tau} \theta, \theta(X)} \qquad (Tuple)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i'}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \xrightarrow{\ell} \theta', \{\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}\}}$$

$$(List1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, [e_1|e_2] \xrightarrow{\ell} \theta', [e_1'|e_2]} \qquad (List2)\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, [v_1|e_2] \xrightarrow{\ell} \theta', [v_1|e_2']}$$

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

$$(Case1)\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\ell} \theta', \mathsf{case}\ e'\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end}}$$

$$(Case2)\ \frac{\mathsf{match}(\theta, v, cl_1, \ldots, cl_n) = \langle \theta_i, e_i \rangle}{\theta, \mathsf{case}\ v\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\tau} \theta\theta_i, e_i}$$

$$(Call1)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{call}\ op\ (\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\ell} \theta', \mathsf{call}\ op\ (\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}})}$$

$$(Call2)\ \frac{\mathsf{eval}(op, v_1, \ldots, v_n) = v}{\theta, \mathsf{call}\ op\ (v_1, \ldots, v_n) \xrightarrow{\tau} \theta, v}$$

$$(Call3)\ \frac{\mathsf{evalM}(\mathsf{M}, op, v_1, \ldots, v_n) = (v, \mathsf{M}')}{\theta, \mathsf{call}\ op\ (v_1, \ldots, v_n) \xrightarrow{\tau\mathsf{MM}'} \theta, v}$$

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

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

Figura 2.2: Semantica standard: valutazione espressioni sequenziali

Le transizioni sono etichettate con τ (una riduzione sequenziale senza side-effects), o con $\tau\mathsf{MM}'$ che indica una riduzione che accede alla mappa senza side-effects, o con l'etichetta $\perp$ che indica la propagazione di un fallimento, o con un'etichetta che identifica la riduzione di un'azione con alcuni side-effects. Le etichette sono usate nelle regole di sistema (Figure 2.6, 2.7) per determinare gli effetti collaterali associati e/o le informazioni da recuperare.
Nella Figura 2.2 sono presenti le regole di transizione per la valutazione delle

espressioni sequenziali.
Come in Erlang, consideriamo che l'ordine di valutazione degli argomenti in una tupla, lista, ecc. È fisso da sinistra a destra.
Per la valutazione dei case, si assume una funzione ausiliaria match che seleziona la prima clausola, $cl_i = (pat_i \text{ when } e'_i \rightarrow e_i)$, in modo tale che v corrisponda a pat_i, ovvero $v = \theta_i(pat_i)$, e che la guardia sia soddisfatta, cioè, $\theta\theta_i, e'_i \longrightarrow^* \theta', true$. Come in Core Erlang, assumiamo che i patterns possano solo contenere nuove variabili (ma le guardie potrebbero avere variabili legate, quindi passiamo l'ambiente corrente θ alla funzione match).
Le funzioni possono essere definite nel programma (in questo caso sono invocate da apply) o essere un built-in (invocate da call). In quest'ultimo caso vengono valutati utilizzando la funzione ausiliaria eval.
Nella regola *Apply2*, si considera che la mappatura μ memorizza tutte le definizioni di funzione nel programma, cioè, mappa ogni nome di funzione a/n in una copia della sua definizione fun $(X_1, \ldots, X_n) \rightarrow e$, dove $X_1, \ldots, X_n$ sono nuove variabili (distinte) e sono le uniche variabili che possono essere libere in e. Per quanto riguarda le applicazioni, si noti che consideriamo solo il primo ordine funzioni. Per estendere la nostra semantica a considerare anche funzioni di ordine superiore, si dovrebbe ridurre il nome della funzione a *chiusura* della forma $(\theta', \text{fun } (X_1, \ldots, X_n) \rightarrow e)$.
Alle regole della semantica standard delle espressi sequenziali è stata aggiunta la regola $Call3$ utilizzata per la valutazione delle funzioni built-in che devono accedere alla mappa. Le funzioni built-in come si può vedere dalla regola $Call2$ vengono valutate tramite la funzione ausiliaria eval, per questo motivo si è scelto di utilizzare la funzione ausiliaria evalM per valutare le funzioni built-in che utilizzano una mappa. Infatti per la valutazione evalM oltre a prendere in input i parametri prende in input pure la mappa, indicata con M. La funzione evalM e la funzione eval sono quindi sono due funzioni parziali visto che non è definito l'output per ogni input. M′ corrisponde alla parte della mappa acceduta dalla funzione evalM.

$$(TupleF)\ \frac{\theta, e_i \xrightarrow{\perp} fail}{\theta, \{\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}\} \xrightarrow{\perp} fail}$$

$$(List1F)\ \frac{\theta, e_1 \xrightarrow{\perp} fail}{\theta, [e_1|e_2] \xrightarrow{\perp} fail} \quad (List2F)\ \frac{\theta, e_2 \xrightarrow{\perp} fail}{\theta, [v_1|e_2] \xrightarrow{\perp} fail}$$

$$(Let1F)\ \frac{\theta, e_1 \xrightarrow{\perp} fail}{\theta, \mathsf{let}\ X = e_1\ \mathsf{in}\ e_2 \xrightarrow{\perp} fail}$$

$$(Case1F)\ \frac{\theta, e \xrightarrow{\perp} fail}{\theta, \mathsf{case}\ e\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\perp} fail}$$

$$(Case2F)\ \frac{\mathsf{match}(\theta, v, cl_1, \ldots, cl_n) = \emptyset}{\theta, \mathsf{case}\ v\ \mathsf{of}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\perp} caseFail}$$

$$(Call1F)\ \frac{\theta, e_i \xrightarrow{\perp} fail}{\theta, \mathsf{call}\ op\ (\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\perp} fail}$$

$$(Call2F)\ \frac{\mathsf{eval}(op, v_1, \ldots, v_n) = \emptyset\ \wedge\ \mathsf{evalM}(\mathsf{M}, op, v_1, \ldots, v_n) = \emptyset}{\theta, \mathsf{call}\ op(v_1, \ldots, v_n) \xrightarrow{\perp} callFail}$$

$$(Apply1F)\ \frac{\theta, e_i \xrightarrow{\perp} fail}{\theta, \mathsf{apply}\ a/n\ (\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}) \xrightarrow{\perp} fail}$$

$$(Apply2F)\ \frac{\mu(a/n) = \emptyset}{\theta, \mathsf{apply}\ a/n\ (v_1, \ldots, v_n) \xrightarrow{\perp} applyFail}$$

Figura 2.3: Semantica standard: valutazione del fallimento delle espressioni sequenziali

Nella Figura 2.3 sono presenti le regole di transizione che gestiscono i fallimenti. La propagazione dell'errore è indicata dall'etichetta $\perp$, quindi se una valutazione di un'espressione fallisce allora il fallimento viene propagato. Con il simbolo $\emptyset$ si intende:

- nel caso dei match, che l'argomento v non corrisponde a nessuna clausola[1];

[1]Più precisamente in un programma Erlang quando è tradotto nella rappresentazione intermedia Core Erlang, viene aggiunta una clausola catch-all quindi il simbolo $\emptyset$ corrisponde al match della clausola aggiunta.

- nei casi di eval, evalM e μ significa che le funzioni non sono definite sull'input dato.

$$(Send1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, e_1 \,!\, e_2 \xrightarrow{\ell} \theta', e_1' \,!\, e_2} \qquad (Send2)\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, v_1 \,!\, e_2 \xrightarrow{\ell} \theta', v_1 \,!\, e_2'}$$

$$(Send3)\ \frac{}{\theta, v_1 \,!\, v_2 \xrightarrow{\mathsf{send}(v_1, v_2)} \theta, v_2}$$

$$(Receive)\ \frac{}{\theta, \mathsf{receive}\ cl_1; \ldots; cl_n\ \mathsf{end} \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta, \kappa}$$

$$(Spawn1)\ \frac{\theta, e_i \xrightarrow{\ell} \theta', e_i' \quad i \in \{1, \ldots, n\}}{\theta, \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}]) \xrightarrow{\ell} \theta', \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i', \overline{e_{i+1,n}}])}$$

$$(Spawn2)\ \frac{}{\theta, \mathsf{spawn}(a/n, [\overline{v_n}]) \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta, \kappa}$$

$$(Self)\ \frac{}{\theta, \mathsf{self}() \xrightarrow{\mathsf{self}(\kappa)} \theta, \kappa}$$

$$(Register1)\ \frac{\theta, e_1 \xrightarrow{\ell} \theta', e_1'}{\theta, \mathsf{register}(e_1, e_2) \xrightarrow{\ell} \theta', \mathsf{register}(e_1', e_2)}$$

$$(Register2)\ \frac{\theta, e_2 \xrightarrow{\ell} \theta', e_2'}{\theta, \mathsf{register}(v_1, e_2) \xrightarrow{\ell} \theta', \mathsf{register}(v_1, e_2')}$$

$$(Register3)\ \frac{}{\theta, \mathsf{register}(v_1, v_2) \xrightarrow{\mathsf{register}(\kappa, v_1, v_2)} \theta, \kappa}$$

$$(Unregister1)\ \frac{\theta, e \xrightarrow{\ell} \theta', e'}{\theta, \mathsf{unregister}(e) \xrightarrow{\ell} \theta', \mathsf{unregister}(e')}$$

$$(Unregister2)\ \frac{}{\theta, \mathsf{unregister}(v) \xrightarrow{\mathsf{unregister}(\kappa, v)} \theta, \kappa}$$

Figura 2.4: Semantica standard: valutazione delle espressioni concorrenti

Consideriamo ora la valutazione di espressioni concorrenti che producono side-effects (Figura 2.4). Abbiamo le regole *Send1*, *Send2* e *Send3* per " ! ". In questo caso sappiamo *localmente* a cosa dovrebbe essere ridotta l'espressione (cioè v_2 nella regola *Send3*). Per le restanti regole, questo non è noto localmente e, quindi, restituiamo un nuovo simbolo distinto, κ, che viene trattato come una variabile, in modo che nelle regole di sistema nella

figure 2.6, 2.7 κ legherà al suo valore corretto:[2] l'espressione selezionata nella regola *Ricevive*, un pid nelle regole *Spawn* e *Self* e **true** o **false** nelle regole *Register* e *Unregister*. In questi casi, l'etichetta della transizione contiene tutte le informazioni necessarie alle regole di sistema per eseguire la valutazione a livello di sistema, incluso il simbolo κ. Questo *trucco* ci permette di mantenere separate le regole per espressioni e sistemi (cioè, la semantica mostrata nelle Figura 2.2 e 2.4 è per lo più indipendente dalle regole nelle Figure 2.6, 2.7), in contrasto con altre semantiche di Erlang, ad esempio [6], dove sono combinate in una singola relazione di transizione. Nelle regole di valutazione delle espressioni concorrenti della semantica standard sono state aggiunte le regole per la valutazione della register e dell'unregister.

$$(Send1F)\ \frac{\theta, e_1 \xrightarrow{\perp} fail}{\theta, e_1 \,!\, e_2 \xrightarrow{\perp} fail} \qquad (Send2F)\ \frac{\theta, e_2 \xrightarrow{\perp} fail}{\theta, v_1 \,!\, e_2 \xrightarrow{\perp} fail}$$

$$(Spawn1F)\ \frac{\theta, e_i \xrightarrow{\perp} fail}{\theta, \mathsf{spawn}(a/n, [\overline{v_{1,i-1}}, e_i, \overline{e_{i+1,n}}]) \xrightarrow{\perp} fail}$$

$$(Register1F)\ \frac{\theta, e_1 \xrightarrow{\perp} fail}{\theta, \mathsf{register}(e_1, e_2) \xrightarrow{\perp} fail} \qquad (Register2F)\ \frac{\theta, e_2 \xrightarrow{\perp} fail}{\theta, \mathsf{register}(v_1, e_2) \xrightarrow{\perp} fail}$$

$$(Unregister1F)\ \frac{\theta, e \xrightarrow{\perp} fail}{\theta, \mathsf{unregister}(e) \xrightarrow{\perp} fail}$$

Figura 2.5: Semantica standard: valutazione del fallimento delle espressioni concorrenti

Anche nelle regole della valutazione delle espressioni concorrenti sono state aggiunte le regole che gestiscono il fallimento di un processo e che propagano il fallimento della valutazione di un'espressione.
Ora si possono presentare le regole di sistema che sono rappresentate nelle figure 2.6 e 2.7, in tutte le regole viene considerato un sistema generale nella forma $\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M}$ dove Γ è la casella di posta globale, $\langle p, (\theta, e), q\rangle \mid \Pi$ è l'insieme dei processi che contiene almeno un processo e M è la mappa dei pid registrati. Rispetto all'articolo [2] in tutte le regole è stata aggiunta M visto che la definizione di sistema (Definizione 3) è cambiata.

[2] Nota che κ assume valori nel dominio $expr \cup Pid$, al contrario delle variabili ordinarie che possono essere associate solo a valori.

Ora verranno descritte brevemente tutte le regole di transizione del sistema.

$$(Seq)\ \frac{\theta, e \xrightarrow{\tau} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}}$$

$$(Call3)\ \frac{\theta, e \xrightarrow{\tau \mathsf{MM}'} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}}$$

$$(Send)\ \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \mathsf{isAtom}(p'') = false}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma \cup (p'', v); \langle p, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}}$$

$$(SendA)\ \frac{\theta, e \xrightarrow{\mathsf{send}(a, v)} \theta', e' \quad \mathsf{isAtom}(a) = true \quad \mathsf{matchPid}(\mathsf{M}, a) = p''}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma \cup (p'', v); \langle p, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}}$$

$$(SendF)\ \frac{\theta, e \xrightarrow{\mathsf{send}(a, v)} \theta', e' \quad \mathsf{isAtom}(a) = true \quad \mathsf{matchPid}(\mathsf{M}, a) = false}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta, sendFail), q\rangle \mid \Pi; \mathsf{M}}$$

$$(Receive)\ \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, v)}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \backslash\backslash v\rangle \mid \Pi; \mathsf{M}}$$

$$(Spawn)\ \frac{\theta, e \xrightarrow{\mathsf{spawn}(\kappa, a/n, [\overline{v_n}])} \theta', e' \quad p' \text{ is a fresh pid}}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p'\}), q\rangle \mid \langle p', (id, \mathsf{apply}\ a/n\ (\overline{v_n})), [\,]\rangle \mid \Pi; \mathsf{M}}$$

$$(Self)\ \frac{\theta, e \xrightarrow{\mathsf{self}(\kappa)} \theta', e'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta', e'\{\kappa \mapsto p\}), q\rangle \mid \Pi; \mathsf{M}}$$

$$(Sched)\ \frac{}{\Gamma \cup \{(p, v)\}; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta, e), v : q\rangle \mid \Pi; \mathsf{M}}$$

Figura 2.6: Regole standard della semantica del sistema

La regola *Seq* aggiorna semplicemente il controllo (θ, e) del processo considerato quando un'espressione sequenziale viene ridotta utilizzando le regole di espressione.

La regola *Call3* come *Seq* aggiorna semplicemente il controllo (θ, e) del processo considerato quando un'espressione sequenziale viene ridotta utilizzando le regole di espressione, però accede alla mappa, quindi utilizza l'etichetta $\tau\mathsf{MM}'$ invece di τ.

Le regole *Send* e *SendA* aggiungono la coppia (p'', v) alla casella di posta globale Γ invece di aggiungerla alla coda del processo p''. Ciò è necessario

per garantire che tutti i possibili intrecci di messaggi siano modellati correttamente. È stata introdotta la funzione ausiliaria isAtom che viene utilizzata per capire se la regola debba accedere alla mappa, l'unica differenza fra le due regole è che la *SendA* utilizza un atomo come destinatario. Si noti che e' è solitamente diverso da v poiché e può avere operatori annidati diversi. Ad esempio, se e ha la forma " case p$!V$ of $\{\ldots\}$," allora e' sarà "case v of $\{\ldots\}$" con etichetta send(p, v).

La regola *SendF* introduce un fallimento, infatti l'espressione da valutare successivamente nel processo sarà $sendFail$ un atomo che corrisponde ad un fallimento perché all'atomo che si utilizza per inviare un messaggio non è associato nessun pid che si capisce grazie alla funzione matchPid.

Nella regola *Receive*, usiamo la funzione ausiliaria matchrec per valutare un'espressione di ricezione. La differenza principale con match è che matchrec prende anche una coda q e restituisce il messaggio selezionato v. Più precisamente, la funzione matchrec esegue la scansione della coda q cercando il *primo* messaggio v che corrisponde a un modello dell'istruzione di ricezione. Quindi, κ viene associato all'espressione nella clausola selezionata, e_i, e l'ambiente viene esteso con la sostituzione corrispondente.

Se nessun messaggio nella coda q corrisponde ad alcuna clausola, la regola non è applicabile e il processo selezionato non può essere ridotto (cioè viene sospeso). Come nelle espressioni case, assumiamo che i pattern possano contenere solo nuove variabili.

Le regole presentate finora consentono di memorizzare i messaggi nella casella di posta globale, ma non di rimuoverne i messaggi. Questo è esattamente il compito dello scheduler, modellato dalla regola *Sched*. Questa regola sceglie in modo non deterministico una coppia (p, v) nella cassetta postale globale Γ e consegna il messaggio v al processo di destinazione p. Qui, ignoriamo deliberatamente la restrizione: " i messaggi inviati, direttamente, tra due processi dati arrivano nello stesso ordine in cui sono stati inviati ", poiché le attuali implementazioni lo garantiscono solo all'interno dello stesso nodo. In pratica, ignorare questa restrizione equivale a considerare che ogni processo è potenzialmente eseguito in un nodo diverso. Una definizione alternativa che garantisce questa restrizione può essere trovata in [8].

Nella figura 2.7 si può notare che sono state aggiunte anche le regole per la register e per l'unregister. Queste regole sono divise in 2 casi:

- il caso di successo;
- il caso di fallimento (che introduce come prossima espressione da valutare un atomo che descrive un fallimento).

$$(RegisterT)\ \frac{\theta, e \xrightarrow{\mathsf{register}(\mathsf{true},a,p')} \theta', e' \ \mathsf{matchMapReg}(\mathsf{M}, a, p') = true}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \mid \Pi; \mathsf{M} \cup (a, p')}$$

$$(RegisterF)\ \frac{\theta, e \xrightarrow{\mathsf{register}(\mathsf{false},a,p')} \theta', e' \ \mathsf{matchMapReg}(\mathsf{M}, a, p') = false}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta, regFail), q\rangle \mid \Pi; \mathsf{M}}$$

$$(UnregisterT)\ \frac{\theta, e \xrightarrow{\mathsf{unregister}(\mathsf{true},a)} \theta', e' \ \mathsf{matchMapUnreg}(\mathsf{M}, a) = p'}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta', e'), q\rangle \mid \Pi; \mathsf{M} \backslash\backslash (a, p')}$$

$$(UnregisterF)\ \frac{\theta, e \xrightarrow{\mathsf{unregister}(\mathsf{false},a)} \theta', e' \ \mathsf{matchMapUnreg}(\mathsf{M}, a) = false}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta, unregFail), q\rangle \mid \Pi; \mathsf{M}}$$

$$(Catch)\ \frac{\theta, e \xrightarrow{\perp} fail}{\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta, fail), q\rangle \mid \Pi; \mathsf{M}}$$

$$(End)\ \frac{\mathsf{matchEnd}(\mathsf{M}, p) = a'}{\Gamma; \langle p, (\theta, \epsilon), q\rangle \mid \Pi; \mathsf{M} \hookrightarrow \Gamma; \langle p, (\theta, \epsilon), q\rangle \mid \Pi; \mathsf{M} \backslash\backslash (a', p)}$$

Figura 2.7: Regole standard della semantica del sistema che utilizzano la mappa

Il caso di successo corrisponde alle regole *RegisterT* e *UnregisterT* che rispettivamente aggiungono la coppia (a, p') alla mappa e tolgono la coppia (a, p') dalla mappa. Mentre il caso di fallimento corrisponde alle regole *RegisterF* e *UnregisterF*.

La regola *Catch* serve per gestire il caso di fallimento di un processo. In questo caso il fallimento viene catturato al top-level.

La regola *End* serve per gestire il caso di fine di un processo. In caso di fallimento o terminazione di un processo se il processo è registrato viene deregistrato.

Questa scelta è stata fatta per essere il più possibili fedeli al comportamento di Erlang, visto che nel linguaggio utilizzato non sono presenti le eccezioni.

Concorrenza in Erlang

Per definire una semantica reversibile per Erlang abbiamo bisogno non solo di una semantica come quella appena presentata, ma anche di una nozione di concorrenza (o, equivalentemente, della nozione opposta di conflitto).

Dati i sistemi s_1, s_2, chiamiamo $s_1 \hookrightarrow^* s_2$ una *derivazione*. Le derivazioni in un passaggio sono chiamate semplicemente *transizioni*. Usiamo $d, d', d_1, \ldots$ per indicare le derivazioni e $t, t', t_1, \ldots$ per le transizioni.

Le transizioni vengono etichettate con: $s_1 \hookrightarrow_{p,r} s_2$ dove:

- p è il pid del processo selezionato nella transizione o del processo a cui viene consegnato un messaggio (se la regola applicata è *Sched*);
- r è l'etichetta della regola di transizione applicata.

Data una derivazione $d = (s_1 \hookrightarrow^* s_2)$, definiamo $\mathsf{init}(d) = s_1$ e $\mathsf{final}(d) = s_2$. Due derivazioni, d_1 e d_2, sono *componibili* se $\mathsf{final}(d_1) = \mathsf{init}(d_2)$. In questo caso, $d_1; d_2$ corrisponde alla composizione $d_1; d_2 = (s_1 \hookrightarrow s_2 \hookrightarrow \cdots \hookrightarrow s_n \hookrightarrow s_{n+1} \hookrightarrow \cdots \hookrightarrow s_m)$ se $d_1 = (s_1 \hookrightarrow s_2 \hookrightarrow \cdots \hookrightarrow s_n)$ e $d_2 = (s_n \hookrightarrow s_{n+1} \hookrightarrow \cdots \hookrightarrow s_m)$. Due derivazioni, d_1 e d_2, sono dette *coiniziali* se $\mathsf{init}(d_1) = \mathsf{init}(d_2)$ e *cofinali* se $\mathsf{final}(d_1) = \mathsf{final}(d_2)$.

Definizione 4 (Transizioni concorrenti). Date due transizioni coiniziali, $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ e $t_2 = (s \hookrightarrow_{p_2,r_2} s_2)$, si dice che sono *in conflitto* se:

- considerano lo stesso processo, i.e., $p_1 = p_2$, e anche $r_1 = r_2 = \mathit{Sched}$ o se una transizione applica la regola *Sched* e l'altra applica la regola *Receive* [2];
- entrambe sono operazioni sulla mappa e una delle due regole accede in scrittura sulla mappa (RegisterT, UnregisterT, End) e l'altra accede o in scrittura o in lettura (SendA, SendF, RegisterF, UnregisterF, Call3) sulla mappa, in modo che una delle funzioni (matchPid, matchMapReg, matchMapUnreg, matchEnd, evalM) utilizzate non abbia più lo stesso risultato. I casi in cui questo accade sono presenti nella tabella 2.1.

  Nella Tabella 2.1 sono presenti i casi in cui si verificano i conflitti, a_1 e p' rappresentano l'atomo e il pid coinvolti da r_1, mentre a_2 e p'' rappresentano l'atomo e il pid coinvolto da r_2. M′ rappresenta gli elementi letti nella mappa dalla regola Call3. Le celle vuote rappresentano transizioni coiniziali che non vanno in conflitto.
  Da notare che non tutte le operazioni in scrittura sulla mappa sono in conflitto fra di loro questo perché in quei casi le transizioni non possono essere coiniziali e in conflitto.

Due transizioni coiniziali sono *concorrenti* se non sono in conflitto. □

Esempio 5. *Nel caso* RegisterT, UnregisterT *non può essere presente un conflitto quando le due transizioni sono coiniziali perché si dovrebbe avere*

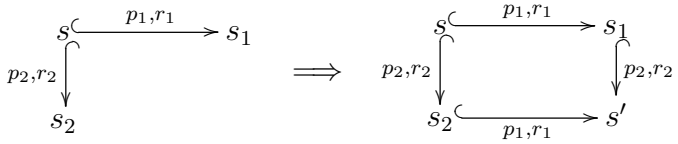
r_1 \ r_2	Call3	UnregisterF	RegisterF	SendF	SendA	End	UnregisterT	RegisterT
RegisterT	$a_1 \in \mathsf{M}' \vee p' \in \mathsf{M}'$	$a_1 = a_2$		$a_1 = a_2$				$a_1 = a_2 \vee p' = p''$
UnregisterT	$a_1 \in \mathsf{M}' \vee p' \in \mathsf{M}'$		$a_1 = a_2$		$a_1 = a_2$	$a_1 = a_2$	$a_1 = a_2$	
End	$a_1 \in \mathsf{M}' \vee p' \in \mathsf{M}'$		$a_1 = a_2$		$a_1 = a_2$			
SendA								
SendF								
RegisterF								
UnregisterF								
Call3								

Tabella 2.1: Tabella dei conflitti

che a_1 non dovrebbe appartenere alla mappa di atomi e pid per applicare RegisterT*, a_2 dovrebbe appartenere alla mappa di atomi e pid per poter applicare* UnregisterT *e per essere in conflitto a_1 dovrebbe essere uguale ad a_2 il che è impossibile.*

Questa definizione è utile per la dimostrazione del prossimo lemma.

Lemma 6 (**Square lemma**). *Date due transizioni coiniziali concorrentu $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ e $t_2 = (s \hookrightarrow_{p_2,r_2} s_2)$, allora esistono due transizioni cofinali $t_2/t_1 = (s_1 \hookrightarrow_{p_2,r_2} s')$ e $t_1/t_2 = (s_2 \hookrightarrow_{p_1,r_1} s')$. Graficamente,*



Dimostrazione. Abbiamo i seguenti casi:

- entrambe le transizioni non applicano regole sulla mappa:
  - due transizioni t_1 e t_2 dove $r_1 \neq \mathit{Sched}$ e $r_2 \neq \mathit{Sched}$. Banalmente, si applicano a processi diversi, ad esempio $p_1 \neq p_2$. Quindi, possiamo facilmente dimostrare che applicando la regola r_2 a p_1 in s_1 e la regola r_1 a p_2 in s_2 abbiamo due transizioni t_1/t_2 e t_2/t_1 che sono cofinali [2];
  - una transizione t_1 che applica la regola $r_1 = \mathit{Sched}$ per consegnare il messaggio v_1 all'elaborazione $p_1 = \mathrm{p}$, e un'altra transizione che

applica una regola r_2 diversa da *Sched*. Tutti i casi tranne $r_2 =$ *Ricevi* con $p_2 =$ p sono semplici. Quest'ultimo caso, tuttavia, non può accadere poiché le transizioni che utilizzano le regole *Sched* e *Ricevi* non sono simultanee [2];

- due transizioni t_1 e t_2 con le regole $r_1 = r_2 =$ *Sched* che consegnano rispettivamente i messaggi v_1 e v_2. Poiché le transizioni sono simultanee, dovrebbero consegnare i messaggi a processi differenti, cioè $p_1 \neq p_2$. Pertanto, possiamo vedere che consegnando v_2 da s_1 e v_1 da s_2 otteniamo due transizioni cofinali [2];

- due transizioni t_1 e t_2 dove $r_1 =$ operazione sulla mappa e $r_2 \neq$ operazione sulla mappa. Quindi, possiamo facilmente dimostrare che applicando la regola r_2 a p_1 in s_1 e la regola r_1 a p_2 in s_2 abbiamo due transizioni t_1/t_2 e t_2/t_1 che sono cofinali;

- vale anche se entrambe le operazioni r sono eseguite sulla mappa, questo perché se andassero a modificare gli stessi elementi della mappa non potrebbero essere concorrenti per la definizione data:

  - se $r_1 = r_2 =$ UnregisterT e deregistrano gli atomi a_1 e a_2 se le due operazioni sono concorrenti (quindi $a_1 \neq a_2$), deregistrare prima a_1 e poi a_2 o viceversa non cambia (in s' la mappa sarà la stessa);
  - questo ragionamento lo si può fare con tutte le regole sulla mappa.

□

Notiamo qui che sono possibili altre definizioni di transizioni concorrenti. La modifica del modello di concorrenza richiederebbe la modifica delle informazioni memorizzate nella semantica reversibile al fine di preservare la coerenza causale. Abbiamo scelto la nozione sopra poiché è ragionevolmente semplice da definire e perché è facile lavorarci.

2.3 Semantica reversibile

2.4 Semantica rollback

Capitolo 3

Conclusione

Bibliografia

[1] Richard Carlsson. An introduction to core Erlang. In *In Proceedings of the PLI'01 Erlang Workshop*, 2001.

[2] Ivan Lanese, Naoki Nishida, Adrián Palacios, and Germán Vidal. A theory of reversibility for Erlang. *Journal of Logical and Algebraic Methods in Programming*, 100:71–97, November 2018.

[3] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, July 1961.

[4] C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, November 1973.

[5] Vincent Danos and Jean Krivine. Reversible communicating systems. In *CONCUR 2004 - Concurrency Theory*, pages 292–307. Springer Berlin Heidelberg, 2004.

[6] R. Caballero, E. Martín-Martín, A. Riesco, and S. Tamarit. A declarative debugger for concurrent erlang programs (extended version). Technical Report SIC-15/13, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2013.

[7] Engblom, J. (2012, September). A review of reverse debugging. In System, Software, SoC and Silicon Debug Conference (S4D), 2012 (pp. 1-6). IEEE.

[8] N. Nishida, A. Palacios, and G. Vidal. A reversible semantics for Erlang. In M. Hermenegildo and P. López-García, editors, *Proc. of the 26th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2016*, volume 10184 of *LNCS*, pages 259–274. Springer, 2017. Preliminary version available from `https://arxiv.org/abs/1608.05521`.