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. In questo articolo viene presentata una semantica reversibile per un sotto insieme di Core Erlang.
Nella Sezione 2.1 viene modificato il linguaggio presentato nell'articolo [2] aggiungendo le operazione di registrazione di atomi e pid, di deregistrazione di pid e di lookup e un meccanismo 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] introducendo una mappa formata dalle coppie di atomi-pid che sono stati registrati. Rispetto all'articolo [2] sono state estese le regole per: la semantica del linguaggio, la valutazione delle nuove espressioni e per descrivere il fallimento di un'espressione. 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 *forward* e *backward*. La prima è un'estensione delle regole introdotte precedentemente, a cui nell'articolo [2] viene aggiunta una storia dei processi; in questo lavoro è stata aggiunta la 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 con le istruzioni che vengono eseguite in un determinato ordine; per computazione reversibile invece 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, in quanto 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 così da non avere perdita di informazione.
L'idea alla base di questo lavoro [3], chiamato incorporamento di Landauer, è che qualsiasi linguaggio di programmazione o formalismo può essere reso reversibile aggiungendo la cronologia del calcolo a ogni stato.
Questa idea fu ulteriormente migliorata da Bennett [4] allo scopo 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 facilmente ottenibile annullando ricorsivamente l'ultima azione eseguita dal programma. La definizione di reversibilità in un contesto in cui vengono considerati anche i sistemi concorrenti è più complicata, poiché non esiste il concetto di "ultima azione", essendo molte azioni eseguite contemporaneamente.
Una definizione adeguata di reversibilità in uno scenario concorrente è stata proposta da Danos e Krivine nel loro articolo [5]. Intuitivamente, questa 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 svolte, per esempio nel debug.

1.2 Debugger reversibile

I Debugger non sono altro che programmi che consentono di analizzare se un programma è sintatticamente corretto, in modo da effettuare una ricerca del bug in modo più veloce e accurato possibile.
Generalmente il tipo più utilizzato di debugger è quello che consente a runtime la ricerca dei bug tramite funzionalità standard come *breakpoint*, *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 programma.
Di contro però, la maggior parte dei debugger fornisce un'assistenza limitata, 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 breakpoint 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. Per far fronte a questo problema sono nati anche debugger più complessi, definiti reversibili, che risultano più utili in scenari concorrenti. Questi, 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 [7].
Riavvolgere le azioni di un particolare processo significa anche elimare ogni conseguenza dell'operazione, cioè tutte le azioni "collegate" a quell'operazione, che si vuole annullare. Questa nozione è chiamata *consistenza causale* [5].

1.3 Background

In questa tesi, viene utilizzato "A theory of reversibility for Erlang" [2] come articolo di riferimento. In esso viene utilizzato come linguaggio un sottoinsieme di *Core Erlang* [1].
Inizialmente, al contrario di quelle utilizzate in altri lavori [6], viene introdotta una semantica modulare per il linguaggio, ciò permette di semplificare l'aggiornamento di quest'ultima ad una semantica reversibile.

Sucessivamente, utilizzando l'incorporamento di Landauer, è stata definita una semantica reversibile per il linguaggio scelto che può andare avanti e indietro in modo non deterministico, chiamata semantica reversibile *non controllata*, per la quale è stato dimostrato che essa gode delle proprietà usuali (loop lemma, square lemma e consistenza causale).
Infine, è stata introdotta una versione controllata della semantica a ritroso grazie all'utilizzo dell'operatore di rollback che annulla le azioni di un processo fino a un determinato punto di controllo; per garantire la coerenza causale, l'azione di rollback potrebbe essere propagata ad altri processi dipendenti.
L'implementazione di CauDEr mostra che l'approccio utilizzato è effettivamente attuabile nella pratica.

1.4 CauDEr

Un debugger casual-consistent è uno strumento che consente di eseguire il debug di un programma concorrente, garantendo la coerenza causale. Non è semplice individuare un comportamento erroneo in un sistema concorrente, quindi è utile disporre di strumenti che possono aiutare lo sviluppatore. In [8] viene introdotto CauDEr, ovverosia un debugger casual-consistent per un sottoinsieme funzionale e concorrente del linguaggio Core Erlang [1].
Il programma funziona come segue: l'utente carica un file sorgente Erlang che viene tradotto in Core Erlang. Successivamente egli specifica quale funzione è il punto di ingresso e, eventualmente, anche gli argomenti di input della funzione. Dopo aver premuto il pulsante di avvio il sistema verrà azionato e l'utente potrà eseguire il programma in modo automatico in avanti. Si potranno quindi specificare quanti passi avanti debbano essere eseguiti, o in alternativa si potrà selezionare un processo ed eseguire passi avanti su di esso. L'utente può anche selezionare un processo e se possibile potrà eseguire un calcolo a ritroso. Inoltre, l'utente ha l'opportunità di eseguire il rollback del processo nel momento in cui viene eseguita un'azione osservabile con la garanzia che è stata assicurata la coerenza causale.
Un debugger che permette all'utente di concentrarsi solo sul processo desiderato, ignorando quindi gli altri, riduce la quantità di informazioni che egli debba percepirare in un solo istante, riducendo così il carico cognitivo e consentendo al soggetto di doversi concentrare esclusivamente su ciò di cui ha più bisogno.
Contrariamente un debugger che tiene traccia dell'intero sistema e che può essere spostato avanti e indietro risulterebbe più problematico, infatti un tale strumento non consentirebbe di concentrarsi su un particolare processo ma costringerebbe l'utente a eseguire il rollback e riprodurre l'intero sistema,

anche i processi che non sono legati a quello a cui siamo interessati. Un tale approccio sommergerebbe quindi l'utente di informazioni non utili, aumentando così le probabilità che il bug che causa il comportamento scorretto non venga trovato.

Lo sviluppo di CauDEr è presente in due versioni [9, 10] che entrambe implementano la semantica reversibile descritta nell'articolo [2]. Le principali differenze fra le due versioni sono che nella seconda versione [10] non è presente la mailbox locale dei processi ma solo quella globale. Di conseguenza non è presente nemmeno lo scheduler che sposta i messaggi dalla mailbox globale a quella locale del destinatario. Nella seconda versione è stata aggiunta anche la definizione dei tipi delle strutture dati utilizzate e della sintassi, in questo modo viene aggiunto alle definizione delle funzioni il tipo dell'input e dell' output. Inoltre il linguaggio non è più tradotto in Core Erlang ma nella sintassi definita.
Infine un'altra differenza è che nella prima versione il codice visibile è il programma tradotto in Core Erlang mentre nella seconda versione è il codice originale in Erlang.

Capitolo 2

Estensione con feature imperative

Si vuole estendere [2] con alcuni meccanismi imperativi presenti in Core Erlang e ingorati in [2]: la possibilità di associare atomi a pid, tipicamente utilizzata per rendere pubblici i pid di server per servizi noti.
Nel seguito di questo capitolo, verrà presentata la sintassi del linguaggio esteso supportato, seguita dalla sua 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. 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, concorrente, distribuito e basato sul paradigma ad attori. 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 conseguenza 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 dell'articolo [2] sono state aggiunte le funzioni built-in:

- whereis che dato in input un atomo restituisce il pid associato, oppure **undefined**;

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

Figura 2.1: Regole di sintassi del linguaggio

- 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:

- register che dati in input un atomo e un pid inserisce in una mappa la coppia (atomo,pid) e restituisce l'atomo **true**. 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**. Se l'atomo non è presente nella mappa il processo fallisce.

Inoltre è stato creato il non terminale *end* che indica gli stati finali di un processo, in modo da 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 mailbox del processo, una coda FIFO con la sequenza di messaggi che sono stati inviati al processo.
Si considera le seguenti operazioni sulle mailbox locali. Dato un messaggio v e una mailbox locale q, $v : q$ denota una nuova mailbox con il messaggio v sopra (cioè v è il messaggio più recente). Con $q \backslash\backslash v$ verrà indicata una nuova coda che risulta da q rimuovendo l'occorrenza più vecchia del messaggio v (che non è necessariamente il messaggio più vecchio nella coda). □

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

Definizione 3 (Sistema). Un sistema è indicato da $\Gamma; \Pi; \mathsf{M}$, dove Γ, la *mailbox 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 " $\mid$ " denota un operatore associativo e commutativo. Data una mailbox globale Γ, $\Gamma \cup \{(p, v)\}$ denota una nuova mailbox che include anche la coppia (p, v); si usa "$\cup$" come unione multiset. Spesso un sistema viene denotato 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 " $\mid$ " è associativo e commutativo). Infine, M rappresenta l'insieme delle coppie $(atomo, pid)$ registrate; si indica 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 mailbox 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. Verrà prima mostrata 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 gruppi: quelle per le espressioni sequenziali sono raffigurate nella Figura 2.2, quelle per i fallimenti delle espressioni sequenziali nella Figura 2.3, quelle per le espressioni concorrenti nella Figura 2.4 e infine quelle per i fallimenti delle espressioni concorrenti sono mostrate 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

Ogni transizione è contrassegnata da un etichetta che può essere: τ che indica una riduzione sequenziale senza side-effects, $\tau\mathsf{MM}'$ una riduzione che accede alla mappa senza side-effects, $\bot$ la propagazione di un fallimento, o 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, si considera 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, $cl_i = (pat_i \text{ when } e'_i \to e_i)$, 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, si assume che i patterns possano solo contenere nuove variabili (ma le guardie potrebbero avere variabili legate, quindi verrà passato 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 valutate utilizzando la funzione ausiliaria eval.
Nella regola *Apply2*, si considera che la mappa μ memorizzi tutte le definizioni di funzione nel programma, cioè, mappi ogni nome di funzione a/n in una copia della sua definizione fun $(X_1, \ldots, X_n) \to e$, dove $X_1, \ldots, X_n$ sono nuove variabili distinte e sono le uniche che possono essere libere in e. Per estendere la semantica utilizzata a considerare anche funzioni di ordine superiore, si dovrebbe ridurre il nome della funzione a una *chiusura* della forma $(\theta', \text{fun } (X_1, \ldots, X_n) \to e)$.
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 non prende in input solo i parametri ma anche la mappa, indicata con M. La funzione evalM e la funzione eval sono quindi 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, che le funzioni non sono definite sull'input dato.

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

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

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

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

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

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

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

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

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

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

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

Figura 2.4: Semantica standard: valutazione delle espressioni concorrenti

Si considera ora la valutazione di espressioni concorrenti che producono side-effects (Figura 2.4). Si hanno le regole *Send1*, *Send2* e *Send3* per "!". In questo caso si sa *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, si restituisce un nuovo simbolo distinto, κ, che viene trattato come una variabile, in modo che nelle regole di sistema nelle Figure 2.6,

2.7 κ legherà al suo valore corretto:[2] l'espressione selezionata nella regola *Receive*, 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* 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 di 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 queste viene considerato un sistema generale nella forma $\Gamma; \langle p, (\theta, e), q\rangle \mid \Pi; \mathsf{M}$ dove Γ è la mailbox 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.
Ora verranno descritte brevemente tutte le regole di transizione del sistema.

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

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

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

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

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

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

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

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

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

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

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; accedendo però alla mappa, utilizza l'etichetta $\tau\mathsf{MM}'$ invece di τ.
Le regole *Send* e *SendA* aggiungono la coppia (p'', v) alla mailbox 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) per un qualche v.
La regola *SendF* introduce un fallimento, visto che l'espressione da valutare successivamente nel processo sarà l'atromo $sendFail$ che corrisponde ad un fallimento. Questo avviene perchè l'atomo che si utilizza per inviare un messaggio non è associato nessun pid nella mappa, come comprensibile dalla funzione matchPid.
Nella regola *Receive* si usa 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, si assume che i pattern possano contenere solo nuove variabili.
Le regole presentate finora consentono di memorizzare i messaggi nella mailbox globale, ma non di rimuoverli. Questo è esattamente il compito dello scheduler, modellato dalla regola *Sched*. Questa regola sceglie in modo non deterministico una coppia (p, v) nella mailbox globale Γ e consegna il messaggio v al processo di destinazione p. Qui si ignora 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 [11]. 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).

Il caso di successo corrisponde alle regole *RegisterT* e *UnregisterT* che rispettivamente aggiunge la coppia (a, p') alla mappa e tolgie la coppia (a, p'),

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

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

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

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

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

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

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

mentre il caso di fallimento corrisponde alle regole *RegisterF* e *UnregisterF*. La regola *Catch* serve per gestire il caso di fallimento di un processo che viene catturato al top-level.
La regola *End* serve per gestire il caso di terminazione di un processo. In caso di fallimento o terminazione con successo se è registrato allora viene deregistrato. Questa scelta è stata fatta per essere il più possibili fedeli al comportamento di Erlang.

Concorrenza in Erlang

Per definire una semantica reversibile per Erlang si ha 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, si chiama $s_1 \hookrightarrow^* s_2$ una *derivazione*.
Le derivazioni in un passaggio sono chiamate semplicemente *transizioni*. Si usa $d, d', d_1, \ldots$ 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 $d = (s_1 \hookrightarrow^* s_2)$, si definisce $\mathsf{init}(d) = s_1$ 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 vi accede o in scrittura o in lettura (SendA, SendF, RegisterF, UnregisterF, Call3), 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.

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

Tabella 2.1: Tabella dei conflitti per le regole forward-forward

Nella Tabella 2.1 sono presenti i casi in cui si verificano i conflitti; a' e p' rappresentano l'atomo e il pid coinvolti da r_1, mentre a'' e p'' rappresentano l'atomo e il pid coinvolto da r_2. M' rappresenta gli elementi

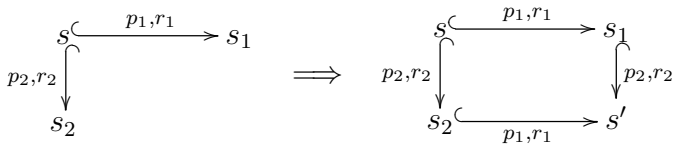
letti nella mappa dalla regola Call3. Le celle vuote rappresentano transizioni coiniziali che non vanno in conflitto.
Si nota che esistono coppie di operazini in scrittura che non sono in conflitto visto che nei casi in cui sarebbe presente il conflitto queste operazioni non possono essere coiniziali.

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

Questa definizione è utile per la dimostrazione del prossimo lemma.

Lemma 6 (**Square lemma**). *Date due transizioni coiniziali concorrenti $t_1 = (s \hookrightarrow_{p_1,r_1} s_1)$ 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. Si hanno 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, si può 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, si hanno 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 $r_2 = \mathit{Receive}$ con $p_2 = \mathrm{p}$ sono semplici. Quest'ultimo caso, tuttavia, non può accadere poiché le transizioni che utilizzano le regole *Sched* e *Receive* non sono simultanee [2];

  - due transizioni t_1 e t_2 con le regole $r_1 = r_2 = \mathit{Sched}$ 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, si può vedere che consegnando v_2 da s_1 e v_1 da s_2 si ottengono 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, si può 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 si hanno due transizioni t_1/t_2 e t_2/t_1 che sono cofinali;

- 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 $r_1 = r_2 = \mathsf{UnregisterT}$ deregistrano gli atomi a_1 e a_2 e 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 può essere applicato a tutte le regole sulla mappa. □

Si noti 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. È stata scelta la nozione sopra poiché è ragionevolmente semplice da definire e perché è facile lavorarci.

2.3 Semantica reversibile

In questa sezione, viene introdotta una semantica reversibile (non controllata) per il linguaggio considerato. Grazie al design modulare della semantica concreta, non è necessario modificare le regole di transizione per le espressioni del linguaggio per definire la semantica reversibile.
Per essere precisi, in questa sezione verranno introdotte due relazioni di transizione: $\rightharpoonup$ e $\leftharpoondown$. La prima relazione, $\rightharpoonup$, è un'estensione conservativa della semantica standard $\hookrightarrow$ (Figure 2.6 e 2.7) per includere anche alcune informazioni aggiuntive negli stati, seguendo un tipico incorporamento di Landauer. Verrà indicata con $\rightharpoonup$ la semantica reversibile *forward* (o semplicemente alla semantica forward). Al contrario, la seconda relazione, $\leftharpoondown$, procede nella direzione all'indietro, "annullando" le azioni passo dopo passo. Ci si riferisce a $\leftharpoondown$ come semantica *backward* (reversibile). Verrà indicata l'unione $\rightharpoonup \cup \leftharpoondown$

con $\rightleftharpoons$.
Per evitare di annullare tutte le azioni fino all'inizio del processo, lasceremo anche che il programmatore introduca *checkpoints*. Sintatticamente, sono indicati con la funzione incorporata check, che accetta un identificatore t come argomento. Tali identificatori dovrebbero essere univoci nel programma. Data un'espressione, *expr*, si può introdurre un checkpoint sostituendo *expr* con " let X = check(t) in *expr*". Una chiamata nella forma check(t) restituisce semplicemente t (vedi sotto). Di seguito, si considera che le regole per valutare le espressioni del linguaggio (Figure 2.2 e 2.4) sono estese con la seguente regola:

$$(Check) \quad \frac{}{\theta, \mathsf{check}(\mathtt{t}) \xrightarrow{\mathsf{check}(\mathtt{t})} \theta, \mathtt{t}}$$

Le principali modifiche effettuate sono state l'aggiunta di una storia al processo, di un identificatore per i messaggi (un timestamp λ) [2] e di una storia della mappa. L'aggiunta della storia del processo e della mappa è un tipico incorporamento di Laundered in cui viene aggiunta la cronologia delle operazioni eseguite dal processo (nel primo caso) e delle operazioni eseguite sulla mappa. Il sistema ora include una memoria (storia) che registra gli stati intermedi della mappa. Questo è stato fatto perché in questo modo è possibile tenere traccia delle operazioni che sono state eseguite sulla mappa e del loro ordine di esecuzione.
Al messaggio è stato aggiunto un timestamp perché, se si considera un processo p1 che invia due messaggi identici ad un altro processo p2 (il che non è insolito, ad esempio un "ack" dopo aver ricevuto una richiesta), per annullare la prima azione p2 ! v del processo p1 è necessario annullare tutte le azioni del processo p2 fino alla ricezione del primo messaggio. Tuttavia, non si può distinguere il primo messaggio dal secondo a meno che non si tenga conto di alcune informazioni aggiuntive. Pertanto, è necessario introdurre un identificatore univoco per distinguere con precisione questo caso.

Le regole di transizioni della semantica forward si possono trovare nelle Figure 2.8 e 2.9. Ai processi è stata aggiunta una *storia* (o memoria) h che registra gli stati intermedi del processo e dei messaggi con l'identificatore (univoco) a loro associato.

Nella storia per identificare le operazioni eseguite si utilizzano dei costruttori τ, check, send, rec, spawn, self, τMM′, sendA, sendF, regT, regF, unregT, unregF, fail e end, che identificano le regole applicate nella semantica forward. Nella regola *Receive* si può notare che la funzione ausiliare matchrec prende in input un messaggio del tipo $\{v, \lambda\}$, che è l'estensione originale che ignora λ quando elabora il match col primo messaggio.

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

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

$$(Check)\ \frac{\theta, e \xrightarrow{\mathsf{check}(\mathtt{t})} \theta', e'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M};\mathsf{S} \rightharpoonup \Gamma; \langle p, \mathsf{check}(\theta, e, \mathtt{t}) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M};\mathsf{S}}$$

$$(Send)\ \frac{\theta, e \xrightarrow{\mathsf{send}(p'', v)} \theta', e' \quad \lambda \text{ is a fresh identifier} \quad \mathsf{isAtom}(p'') = false}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M};\mathsf{S} \rightharpoonup \Gamma \cup (p'', \{v, \lambda\}); \langle p, \mathsf{send}(\theta, e, p'', \{v, \lambda\}) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M};\mathsf{S}}$$

$$(SendA)\ \frac{\theta, e \xrightarrow{\mathsf{send}(a, v)} \theta', e' \quad \lambda \text{ is a fresh identifier} \quad \mathsf{isAtom}(a) = true \quad \mathsf{matchPid}(\mathsf{M}, a) = p''}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M};\mathsf{S} \rightharpoonup \Gamma \cup (p'', \{v, \lambda\}); \langle p, \mathsf{sendA}(\theta, e, p'', \{v, \lambda\}, a, p) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}; \mathsf{sendA}(\theta, e, p'', \{v, \lambda\}, a, p) : \mathsf{S}}$$

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

$$(Receive)\ \frac{\theta, e \xrightarrow{\mathsf{rec}(\kappa, \overline{cl_n})} \theta', e' \quad \mathsf{matchrec}(\theta, \overline{cl_n}, q) = (\theta_i, e_i, \{v, \lambda\})}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M};\mathsf{S} \rightharpoonup \Gamma; \langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta'\theta_i, e'\{\kappa \mapsto e_i\}), q \setminus\!\setminus \{v, \lambda\}\rangle \mid \Pi; \mathsf{M};\mathsf{S}}$$

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

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

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

Figura 2.8: Semantica reversibile forward

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

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

$$(UnregisterT)\ \frac{\theta, e \xrightarrow{\mathsf{unregister(true},a)} \theta', e' \ \mathsf{matchMapUnreg}(\mathsf{M}, a, p) = p'}{\Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \rightharpoonup \Gamma; \langle p, \mathsf{unregT}(\theta, e, a, p', p) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M} \setminus\!\setminus (a, p'); \mathsf{unregT}(\theta, e, a, p', p) : \mathsf{S}}$$

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

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

$$(End)\ \frac{\mathsf{matchEnd}(\mathsf{M}, p) = a'}{\Gamma; \langle p, h, (\theta, \epsilon), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \rightharpoonup \Gamma; \langle p, \mathsf{end}(\theta, \epsilon, a', p) : h, (\theta, \epsilon), q\rangle \mid \Pi, \mathsf{M} \setminus\!\setminus (a', p); \mathsf{end}(\theta, \epsilon, a', p) : \mathsf{S}}$$

$$(End)\ \frac{\mathsf{matchEnd}(\mathsf{M}, p) = \emptyset}{\Gamma; \langle p, h, (\theta, \epsilon), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \rightharpoonup \Gamma; \langle p, \mathsf{end}(\theta, \epsilon, \mathsf{undefined}, p) : h, (\theta, \epsilon), q\rangle \mid \Pi, \mathsf{M}; \mathsf{end}(\theta, \epsilon, \mathsf{undefined}, p) : \mathsf{S}}$$

Figura 2.9: Semantica reversibile forward

Per dimostrare che la semantica forward $\rightharpoonup$ estende la semantica standard $\hookrightarrow$ bisogna introdurre $\mathsf{del}(s)$. Si può vedere che $\mathsf{del}(s)$ denota il sistema risultante da s rimuovendo le cronologie dei processi; formalmente, $\mathsf{del}(\Gamma; \Pi; \mathsf{M}; \mathsf{S}) = \Gamma; \mathsf{del}'(\Pi); \mathsf{M}$, dove:

$$\begin{array}{lcl} \mathsf{del}'(\langle p, h, (\theta, e), q\rangle) & = & \langle p, (\theta, e), q\rangle \\ \mathsf{del}'(\langle p, h, (\theta, e), q\rangle \mid \Pi) & = & \langle p, (\theta, e), q\rangle \mid \mathsf{del}'(\Pi) \end{array}$$

Si assume che Π non sia vuoto.

Teorema 7. *Sia s_1 un sistema nella semantica forward senza occorrenze di "*check*" e $s_1' = \mathsf{del}(s_1)$ un sistema nella semantica standard. Allora, $s_1' \hookrightarrow^* s_2'$ sse $s_1 \rightharpoonup^* s_2$ e $\mathsf{del}(s_2) = s_2'$.*

Dimostrazione. La dimostrazione avviene mostrando che le regole della semantica standard nelle Figure 2.8 e 2.9 sono la versione con la storia delle regole corrispondenti presenti nelle Figure 2.6 e 2.7. L'unico punto complicato è capire che l'introduzione dell'identificatore univoco nel messaggio non cambia il comportamento della regola *Receive* poiché la funzione $\mathsf{matchrec}$ ha come risultato sempre l'occorrenza più vecchia (in termini di posizione nella coda) del messaggio selezionato [2]. □

In alcune delle regole di transizione della semantica backward, nelle side condition si richiederà che non siano presenti alcune delle regole di lettura o scrittura.

Definizione 8 (ε e $\notin$). Per le operazioni sulla mappa si dice che x *ε op* se l'operazione coinvolge l'elemento x, cioè se l'operazione aggiunge, rimuove o legge x dalla mappa. Si indica con $x \notin op$ se l'operazione non coinvolge l'elemento x. Di seguito verrà mostrato l'elenco di come nella storia della mappa le regole vengono rappresentate e di quali sono gli elementi coinvolti.

- se $op = \mathsf{regT}(\theta, e, a, p, p')$ allora a *ε* regT, p *ε* regT;
- se $op = \mathsf{unregT}(\theta, e, a, p, p')$ allora a *ε* unregT, p *ε* unregT;
- se $op = \mathsf{regF}(\theta, e, a, p, p')$ allora a *ε* regF, p *ε* regF;
- se $op = \mathsf{unregF}(\theta, e, a, p')$ allora a *ε* unregF;
- se $op = \mathsf{sendA}(\theta, e, p, \{v, \lambda\}, a, p')$ allora a *ε* sendA, p *ε* sendA;
- se $op = \mathsf{sendF}(\theta, e, a, p')$ allora a *ε* sendF;

- se $op = \mathsf{end}(\theta, e, p')$ allora $a \ \varepsilon \ \mathsf{end}$, $p' \ \varepsilon \ \mathsf{end}$;
- se $op = \tau\mathsf{MM}'(\theta, e, p')$ allora $\forall \ p, a \in \mathsf{M}'; \ p \ \varepsilon \ \tau\mathsf{MM}' \wedge a \ \varepsilon \ \tau\mathsf{MM}'$. □

Definizione 9 (Operazioni lettura e scrittura). Per la semantica standard si era detto che:

- per operazioni in scrittura si intendono le operazioni che modificano la mappa (cioè quelle regole che aggiungono o tolgono coppie alla mappa) e sono RegisterT, UnregisterT, End.
- per operazioni in lettura si intendono le operazioni che accedono alla mappa senza modificarla, e sono SendA, SendF, RegisterF, UnregisterF, Call3.

Questo vale anche per la semantica forward. Per la semantica backward invece:

- le operazioni in scrittura sulla mappa sono $\overline{\mathsf{RegisterT}}$, $\overline{\mathsf{UnregisterT}}$ e $\overline{\mathsf{End}}$. Da notare che anche queste operazioni vanno a modificare la mappa nel modo opposto alla regola forward corrispondente;
- le operazioni in lettura sulla mappa nel caso della semantica backward sono $\overline{\mathsf{SendA}}$, $\overline{\mathsf{SendF}}$, $\overline{\mathsf{RegisterF}}$, $\overline{\mathsf{UnregisterF}}$ e $\overline{\mathsf{Call3}}$. Queste vengono considerate come operazioni che leggono gli stessi elementi dell'operazione forward corrispondenti. □

Scegliere di dire che un'operazione all'indietro sia un'operazione in lettura è una scelta insolita, ma è dovuta al fatto che si vuole evitare un "annullamento" di un'operazione di lettura sulla mappa quando questa è stata modificata.

Esempio 10. *Se si considera il caso in cui la storia ha una forma:*

$$\mathsf{RegT}(a, p); \mathsf{Call3}(a); \mathsf{unregT}(a)$$

se fosse possibile fare un'operazione di annullamento di call3 ($\overline{\mathsf{Call3}(a)}$*), allora si otterrebbe una storia*

$$\mathsf{RegT}(a, p); \mathsf{unregT}(a)$$

Però in questo modo si sarebbe annullata un'operazione di lettura sulla mappa dopo che questa è stata modificata, e si potrebbe riapplicare l'operazione $\mathsf{Call3}(a)$ *che fallirebbe, o darebbe un risultato diverso, ottenendo una storia*

$$\mathsf{RegT}(a, p); \mathsf{unregT}(a); \mathsf{Call3}(a)$$

In questo modo non si potrebbe far vedere che facendo operazioni di undo-redo si ritornerebbe in uno stato equivalente, una proprietà importante che è auspicabile.

Nelle Figure 2.10 e 2.11 sono presenti le regole di transizione della semantica backward.

$(\overline{Seq})$ $\Gamma; \langle p, \tau(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}$

$(\overline{Call3})$ $\Gamma; \langle p, \tau\mathsf{MM}'(\theta, e, p) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \tau\mathsf{MM}'(\theta, \mathsf{e}, \mathsf{p}) : \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S}$
Se in S' non sono presenti operazioni di scrittura su elementi presenti in M'

$(\overline{Check})$ $\Gamma; \langle p, \mathsf{check}(\theta, e, \mathsf{t}) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}$

$(\overline{Send})$ $\Gamma \cup \{(p'', \{v, \lambda\})\}; \langle p, \mathsf{send}(\theta, e, p'', \{v, \lambda\}) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}$

$(\overline{SendA})$ $\Gamma \cup \{(p'', \{v, \lambda\})\}; \langle p, \mathsf{sendA}(\theta, e, p'', \{v, \lambda\}, a, p) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{sendA}(\theta, e, p'', \{v, \lambda\}, a, p) : \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S}$
Se $\forall$ *op* in scrittura $\in \mathsf{S}'$ $a \notin op$ o $p'' \notin op$

$(\overline{SendF})$ $\Gamma; \langle p, \mathsf{sendF}(\theta, e, a, p) : h, (\theta, sendFail), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{sendF}(\theta, e, a, p) : \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S}$
Se $\forall$ *op* in scrittura $\in \mathsf{S}'$ $a \notin op$

$(\overline{Receive})$ $\Gamma; \langle p, \mathsf{rec}(\theta, e, \{v, \lambda\}, q) : h, (\theta', e'), q \setminus\setminus \{v, \lambda\}\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}$

$(\overline{Spawn})$ $\Gamma; \langle p, \mathsf{spawn}(\theta, e, p') : h, (\theta', e'), q\rangle \mid \langle p', [\,], (id, e''), [\,]\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}$

$(\overline{Self})$ $\Gamma; \langle p, \mathsf{self}(\theta, e) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoonup \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}$

$(\overline{Sched})$ $\Gamma; \langle p, h, (\theta, e), \{v, \lambda\} : q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoonup \Gamma \cup (p, \{v, \lambda\}); \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}$
if the topmost $\mathsf{rec}(\ldots)$ item in h (if any) has the form $\mathsf{rec}(\theta', e', \{v', \lambda'\}, q')$ with $q' \setminus\setminus \{v', \lambda'\} \neq \{v, \lambda\} : q$

Figura 2.10: Backward reversible semantics

Ora verranno discusse brevemente alcune situazioni particolari:

- Innanzitutto, si osserva che la regola $\overline{Send}$ può essere applicata solo quando il messaggio inviato si trova nella mailbox globale. Se questo non è il caso (cioè il messaggio è stato consegnato usando la regola *Sched*), allora si dovranno prima applicare i passaggi a ritroso al processo del destinatario fino a che l'applicazione della regola $\overline{Sched}$ rimette

il messaggio nella mailbox globale e la regola $\overline{Send}$ diventa applicabile. Ciò è necessario per garantire la coerenza causale. Nella sezione successiva, introdurremo una strategia particolare che ottiene questo effetto in modo controllato [2].

- Si può notare che la regola $\overline{SendA}$ (come la regola $\overline{Send}$) può essere applicata solo quando il messaggio inviato si trova nella mailbox globale, cioè il messaggio non è stato ancora consegnato usando la regola *Sched*. Inoltre, essendo un'operazione sulla mappa in lettura, si richiede che non siano presenti operazioni in scrittura sulla mappa eseguite successivamente che coinvolgono l'atomo a o il pid p; questo è necessario per rispettare la coerenza causale.

- La regola $\overline{SendF}$ è anch'essa in lettura sulla mappa, quindi per rispettare la consistenza causale verrà richiesto che non siano presenti operazioni in scrittura sulla mappa eseguite successivamente che coinvolgono l'atomo a.

- La regola $\overline{Call3}$, come $\overline{SendA}$ e $\overline{SendF}$, è una regola in lettura sulla mappa, che però come side-condition richiede che non ci siano operazioni in scrittura sulla mappa eseguite successivamente che coinvolgono atomi o pid presenti in M'.

- Una situazione simile si verifica con la regola $\overline{Spawn}$. Dato un processo p con un elemento della cronologia $\mathsf{spawn}(\theta, e, p')$, la regola $\overline{Spawn}$ non può essere applicata finché la cronologia e la coda del processo p' sono entrambi vuoti. Pertanto, si dovrebbe prima applicare una serie di passaggi all'indietro per elaborare p' in modo da poter annullare l'elemento spawn. Si noti che non è necessario richiedere che nessun messaggio indirizzato al processo p' (che diventerebbe un messaggio *orfano*) sia nella mailbox globale: per inviare un tale messaggio il pid p' è necessario, quindi l'invio del messaggio dipende da spawn e di conseguenza deve essere annullato in anticipo [2].

- si osserva anche che la regola $\overline{Receive}$ può essere applicata solo quando la coda del processo è esattamente la stessa che è stata ottenuta dopo aver applicato il corrispondente passaggio *Receive*. Ciò è necessario per garantire che la coda ripristinata sia effettivamente quella giusta (notare che l'aggiunta del messaggio a una coda arbitraria non funzionerebbe poiché non si conosce la sua posizione "giusta") [2].

- L'applicazione della regola $\overline{Sched}$ può essere cambiata con l'applicazione di qualsiasi altra regola all'indietro eccetto $\overline{Receive}$ o un altro $\overline{Sched}$, poiché non interferisce con le regole rimanenti. È garantito il fatto che due $\overline{Sched}$ che coinvolgono lo stesso processo non si spostino, poiché $\overline{Sched}$ si applica sempre al messaggio più recente di una coda. Il fatto che $\overline{Sched}$ e $\overline{Receive}$ non si spostino è garantito poiché la condizione secondaria di $\overline{Sched}$ controlla che non ci sia nessun elemento $\mathsf{rec}(\ldots)$ nella cronologia del processo che può essere utilizzato per applicare la regola $\overline{Receive}$ con la coda corrente. Pertanto, le loro condizioni di applicabilità non si sovrappongono [2].

$$(\overline{RegisterT}) \quad \begin{array}{c} \Gamma; \langle p, \mathsf{regT}(\theta, e, a, p', p) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M} \cup (a, p'); \mathsf{S}' : \mathsf{regT}(\theta, e, a, p', p) : \mathsf{S} \\ \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \end{array}$$
$$Se \ \forall \ op \text{ in scrittura o lettura} \in \mathsf{S}' \ a \not\in op \circ p' \notin op$$

$$(\overline{RegisterF}) \quad \begin{array}{c} \Gamma; \langle p, \mathsf{regF}(\theta, e, a, p', p) : h, (\theta, regFail), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{regF}(\theta, e, a, p', p) : \mathsf{S} \\ \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \end{array}$$
$$Se \ \forall \ op \text{ in scrittura} \in \mathsf{S}' \ a \notin op \circ p' \notin op$$

$$(\overline{UnregisterT}) \quad \begin{array}{c} \Gamma; \langle p, \mathsf{unregT}(\theta, e, a, p', p) : h, (\theta', e'), q\rangle \mid \Pi; \mathsf{M} \backslash\backslash (a, p'); \mathsf{S}' : \mathsf{unregT}(\theta, e, a, p', p) : \mathsf{S} \\ \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \end{array}$$
$$Se \ \forall \ op \text{ in scrittura o lettura} \in \mathsf{S}' \ a \notin op \circ p' \notin op$$

$$(\overline{UnregisterF}) \quad \begin{array}{c} \Gamma; \langle p, \mathsf{unregF}(\theta, e, a, p) : h, (\theta, unregFail), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{unregF}(\theta, e, a, p) : \mathsf{S} \\ \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \end{array}$$
$$Se \ \forall \ op \text{ in scrittura} \in \mathsf{S}' \ a \notin op$$

$$(\overline{Catch}) \quad \Gamma; \langle p, \mathsf{fail}(\theta, e) : h, (\theta, fail), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoondown \Gamma; \langle p, h, (\theta, e), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}$$

$$(\overline{End}) \quad \begin{array}{l} \Gamma; \langle p, \mathsf{end}(\theta, \epsilon, a', p) : h, (\theta, \epsilon), q\rangle \mid \Pi, \mathsf{M} \backslash\backslash (a', p), \mathsf{S}' : \mathsf{end}(\theta, \epsilon, a', p) : \mathsf{S} \leftharpoondown \Gamma; \langle p, h, (\theta, \epsilon), q\rangle \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \\ Se \ \forall \ op \text{ in scrittura o lettura} \in \mathsf{S}' \ a' \notin op \circ p \notin op \end{array}$$

Figura 2.11: Backward reversible semantics

- Le regole $\overline{RegisterT}$, $\overline{RegisterF}$, $\overline{UnregisterT}$ e $\overline{End}$ sono tutte in scrittura sulla mappa che annullano la corrispondente regola forward restituendo il controllo al processo (come tutte le altre regole). Essendo in scrittura sulla mappa richiederanno, prima di poter essere eseguite, l'annullamento di tutte le regole che coinvolgono atomi o pid coinvolti dalla regola che si vuole annullare.

- Anche $\overline{UnregisterF}$ è una regola in scrittura ma, al contrario delle altre, coinvolge solo un atomo (e non un pid) e quindi la sua side-condition richiederà solo che non siano presenti nella storia della mappa operazioni più recenti che coinvolgano l'atomo rimosso da $\overline{UnregisterF}$.

Proprietà della semantica reversibile incontrollata

Di seguito verranno dimostate diverse proprietà della semantica reversibile mostrata precedentemente. Dati i sistemi s_1, s_2 verrà indicata con $s_1 \rightharpoonup^* s_2$ una derivazione *forward* e con $s_2 \leftharpoonup^* s_1$ una derivazione *backward*. La derivazione che potenzialmente include sia i passi avanti che quelli indietro è indicata da $s_1 \rightleftharpoons^* s_2$. Si etichettano le transizioni come segue: $s_1 \rightleftharpoons_{p,r,k} s_2$, dove:

- p, r sono rispettivamente il pid del processo selezionato e l'etichetta della regola applicata,
- k è un elemento della cronologia se la regola applicata è diversa da *Sched* e $\overline{Sched}$ e
- $k = \mathsf{sched}(\{v, \lambda\})$ quando la regola applicata è *Sched* o $\overline{Sched}$, dove $\{v,\ k\}$ è il messaggio consegnato o reinserito in Γ. Si nota che le informazioni sono disponibili quando si applica la regola.

Si estendono le definizioni delle funzioni init e final date nella Sezione 2.2 a derivazioni reversibili in modo naturale. Anche le nozioni di derivazione componibile, coiniziale e cofinale sono estese in modo semplice. Data un'etichetta di regola r, allora $\overline{r}$ denota la sua versione inversa, ovvero se $r = Send$ allora $\overline{r} = \overline{Send}$ e viceversa (se $r = \overline{Send}$ allora $\overline{r} = Send$). Inoltre, data una transizione t, si dice che $\overline{t} = (s' \leftharpoondown_{p,\overline{r},k} s)$ se $t = (s \rightharpoonup_{p,r,k} s')$ e che $\overline{t} = (s' \rightharpoonup_{p,r,k} s)$ se $t = (s \leftharpoondown_{p,\overline{r},k} s')$. Si dice che $\overline{t}$ è *inverso* di t. Questa notazione è naturalmente estesa alle derivazioni.

Di seguito si limita l'attenzione ai sistemi raggiungibili dall'esecuzione di un programma.

Definizione 11 (Sistema raggiungibile). Un sistema è *iniziale* se è composto da un singolo processo che ha una cronologia e una coda vuote; inoltre, anche la mailbox globale è vuota. Un sistema s è raggiungibile se esiste un sistema iniziale s_0 e una derivazione $s_0 \rightleftharpoons^* s$ utilizzando le regole corrispondenti a un dato programma. □

Le definizioni e i lemmi successivi sono utilizzati per dimostrare che ogni transizione in avanti (risp. indietro) può essere annullata da una transizione all'indietro (risp. avanti), come dimostrato nel Lemma 23 .

Definizione 12 (Equivalenza fra storie (≡)). Date due storie $\mathsf{H} = \mathsf{H}_1 : \mathsf{H}_2 : op : \mathsf{H}_3$ e $\mathsf{H}' = \mathsf{H}_1 : op : \mathsf{H}_2 : \mathsf{H}_3$ si dice che:

- se *op* è un'operazione in lettura tale che $a \; \varepsilon \; op$ e $p \; \varepsilon \; op$ e se per ogni operazione in scrittura $op_1 \in \mathsf{H}_2$ vale che $a \notin op_1$ e $p \notin op_1$, allora $\mathsf{H} \cong \mathsf{H}'$.
- se *op* è un'operazione in scrittura tale che $a \; \varepsilon \; op$ e $p \; \varepsilon \; op$ e se per ogni operazione in lettura o scrittura $op_1 \in \mathsf{H}_2$ vale che $a \notin op_1$ e $p \notin op_1$, allora $\mathsf{H} \cong \mathsf{H}'$.

La relazione di equivalenza fra due storie viene indicata col simbolo ≡ ed è la chiusura transitiva e simmetrica della relazione ≅ □

Definizione 13 (Trasposizione). Data una storia $h = \mathsf{H}_1 : \mathsf{H}_2 : op : \mathsf{H}_3$ e una storia $h' = \mathsf{H}_1 : op : \mathsf{H}_2 : \mathsf{H}_3$ tali per cui $h \cong h'$, si dice che è avvenuta la **trasposizione** di *op* (con H_2). □

Corollario 14. *La relazione* ≡ *fra due storie è anche riflessiva.*

Dimostrazione. Per vedere che una storia *h* sia equivalente a se stessa si può sempre considerare $h = \mathsf{H}_1 : op : \mathsf{H}_2 : \mathsf{H}_3$ con $\mathsf{H}_2 = [\,]$ (H_2 vuoto) e quindi si può mostrare che $\mathsf{H}_1 : op : [\,] : \mathsf{H}_3 \cong \mathsf{H}_1 : [\,] : op : \mathsf{H}_3$ e quindi che $h \equiv h$ □

Lemma 15. *Se* $\mathsf{H}_1 : op : \mathsf{H}_2 \cong \mathsf{H}'_1 : op : \mathsf{H}'_2$ *con trasposizione di op, allora* $\mathsf{H}_1 : \mathsf{H}_2 = \mathsf{H}'_1 : \mathsf{H}'_2$.

Dimostrazione. Visto che è avvenuta la trasposizione di *op* togliendo *op* si ottiene una storia $\mathsf{H}_1 : \mathsf{H}_2 = \mathsf{H}'_1 : \mathsf{H}'_2$ dove non è presente nessuna trasposizione. □

Lemma 16. *Se* $\mathsf{H}_1 : op : \mathsf{H}_2 \cong \mathsf{H}'_1 : op : \mathsf{H}'_2$ *con trasposizione di* $op' \neq op$ *con* H_3*(dove* H_3 *parte della storia), allora* $\mathsf{H}_1 : \mathsf{H}_2 \cong \mathsf{H}'_1 : \mathsf{H}'_2$ *mediante la trasposizione di* op' *con* H_3 *o con* H_3 *senza op.*

Dimostrazione.

- se $op \in \mathsf{H}_3$ la trasposizione di op' con H_3 senza *op* è sempre possibile perché applicata su un sottoinsieme di H_3 e quindi la Definizione 12 è sempre rispettata;
- se $op \notin \mathsf{H}_3$ la trasposizione di op' con H_3 è la medesima. □

Corollario 17. *Se* $\mathsf{H}_1 : op : \mathsf{H}_2 \cong \mathsf{H}'_1 : op : \mathsf{H}'_2$, *allora* $\mathsf{H}_1 : \mathsf{H}_2 \cong \mathsf{H}'_1 : \mathsf{H}'_2$.

Dimostrazione.

- se c'è trasposizione di *op* per Lemma 15;
- se non c'è trasposizione di *op* per Lemma 16. □

Lemma 18. *Se* $\mathsf{H}_1 : op : \mathsf{H}_2 \equiv \mathsf{H}'_1 : op : \mathsf{H}'_2$, *allora* $\mathsf{H}_1 : \mathsf{H}_2 \equiv \mathsf{H}'_1 : \mathsf{H}'_2$.

Dimostrazione. Per induzione strutturale sull'albero di derivazione della chiusura contestuale la dimostrazione è:

- Caso Base:

$$\frac{\mathsf{H}_1 : op : \mathsf{H}_2 \cong \mathsf{H}'_1 : op : \mathsf{H}'_2}{\mathsf{H}_1 : op : \mathsf{H}_2 \equiv \mathsf{H}'_1 : op : \mathsf{H}'_2}$$

  Da $\mathsf{H}'_1 : op : \mathsf{H}'_2 \cong \mathsf{H}_1 : op : \mathsf{H}_2$, per il Corollario 17 vale che $\mathsf{H}_1 : \mathsf{H}_2 \cong \mathsf{H}'_1 : \mathsf{H}'_2$. Quindi si costruisce l'albero:

$$\frac{\mathsf{H}_1 : \mathsf{H}_2 \cong \mathsf{H}'_1 : \mathsf{H}'_2}{\mathsf{H}_1 : \mathsf{H}_2 \equiv \mathsf{H}'_1 : \mathsf{H}'_2}$$

- Caso Induttivo:
  - caso riflessivo:

$$\frac{\mathsf{H}'_1 : op : \mathsf{H}'_2 \equiv \mathsf{H}_1 : op : \mathsf{H}_2}{\mathsf{H}_1 : op : \mathsf{H}_2 \equiv \mathsf{H}'_1 : op : \mathsf{H}'_2}$$

    Per l'ipotesi induttiva su $\mathsf{H}'_1 : op : \mathsf{H}'_2 \equiv \mathsf{H}_1 : op : \mathsf{H}_2$ si ha $\mathsf{H}'_1 : \mathsf{H}'_2 \equiv \mathsf{H}_1 : \mathsf{H}_2$. Quindi si costruisce l'albero:

$$\frac{\mathsf{H}'_1 : \mathsf{H}'_2 \equiv \mathsf{H}_1 : \mathsf{H}_2}{\mathsf{H}_1 : \mathsf{H}_2 \equiv \mathsf{H}'_1 : \mathsf{H}'_2}$$

  - caso transitivo:

$$\frac{\mathsf{H}_1 : op : \mathsf{H}_2 \equiv \mathsf{H}''_1 : op : \mathsf{H}''_2 \qquad \mathsf{H}''_1 : op : \mathsf{H}''_2 \equiv \mathsf{H}'_1 : op : \mathsf{H}'_2}{\mathsf{H}_1 : op : \mathsf{H}_2 \equiv \mathsf{H}'_1 : op : \mathsf{H}'_2}$$

Per l'ipotesi induttiva su $\mathsf{H}_1 : op : \mathsf{H}_2 \equiv \mathsf{H}''_1 : op : \mathsf{H}''_2$ e $\mathsf{H}''_1 : op : \mathsf{H}''_2 \equiv \mathsf{H}'_1 : op : \mathsf{H}'_2$ si ha $\mathsf{H}_1 : \mathsf{H}_2 \equiv \mathsf{H}''_1 : \mathsf{H}''_2$ e $\mathsf{H}''_1 : \mathsf{H}''_2 \equiv \mathsf{H}'_1 : \mathsf{H}'_2$. Quindi si costruisce l'albero:

$$\frac{\mathsf{H}_1 : \mathsf{H}_2 \equiv \mathsf{H}''_1 : \mathsf{H}''_2 \qquad \mathsf{H}''_1 : \mathsf{H}''_2 \equiv \mathsf{H}'_1 : \mathsf{H}'_2}{\mathsf{H}_1 : \mathsf{H}_2 \equiv \mathsf{H}'_1 : \mathsf{H}'_2}$$

□

Due processi nella forma $\langle p, h, (\theta, e), q\rangle$ sono equivalenti se tutti gli elementi sono uguali (la lista h di operazioni deve essere uguale perché altrimenti i processi avrebbero fatto operazioni diverse e quindi non sarebbero uguali).

Definizione 19 (Equivalenza fra sistemi ($\equiv$)). Due sistemi $s_1 = (\Gamma_1, \Pi_1, \mathsf{M}_1, \mathsf{H})$ e $s_2 = (\Gamma_2, \Pi_2, \mathsf{M}_2, \mathsf{H}')$ sono equivalenti se:

- $\Gamma_1 = \Gamma_2$
- $\Pi_1 = \Pi_2$
- $\mathsf{M}_1 = \mathsf{M}_2$
- $\mathsf{H} \equiv \mathsf{H}'$

Due sistemi s_1, s_2 equivalenti vengono indicati con $s_1 \equiv s_2$. □

Lemma 20. *Dato un sistema s con storia $h = \mathsf{H}_2$ op H_1 in cui è possibile applicare $\overline{op}$, e un sistema $s' \equiv s$ con storia $h' = \mathsf{H}'_2$ op H'_1, allora è possibile applicare $\overline{op}$ a s'.*

Dimostrazione. In s è possibile applicare $\overline{op}$ quindi:

- se $\overline{op}$ è in lettura e $a \ \varepsilon \ \overline{op}$ e $p \ \varepsilon \ \overline{op}$, allora in H_2 non sono presenti operazioni in scrittura tali che $a \ \varepsilon \ op_1$ o $p \ \varepsilon \ op_1$;
- se $\overline{op}$ è in scrittura e $a \ \varepsilon \ \overline{op}$ e $p \ \varepsilon \ \overline{op}$, allora in H_2 non sono presenti operazioni in scrittura o lettura tali che $a \ \varepsilon \ op_1$ o $p \ \varepsilon \ op_1$.

Visto che $s \equiv s'$ allora anche $h \equiv h'$ per la Definizione 19. Per applicare $\overline{op}$ in s' bisogna che le side-condition della regola vengano rispettate in h' e quindi che:

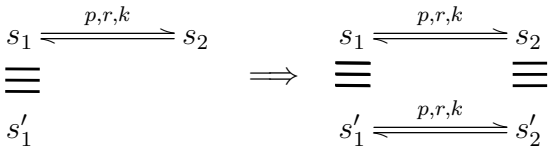
- se $\overline{op}$ è in lettura e $a \ \varepsilon \ \overline{op}$ e $p \ \varepsilon \ \overline{op}$, allora in H'_2 non devono essere presenti operazioni in scrittura tali che $a \ \varepsilon \ op_1$ o $p \ \varepsilon \ op_1$;
- se $\overline{\mathsf{op}}$ è in scrittura e $a \ \varepsilon \ \overline{op}$ e $p \ \varepsilon \ \overline{op}$, allora in H'_2 non devono essere presenti operazioni in scrittura o lettura tali che $a \ \varepsilon \ op_1$ o $p \ \varepsilon \ op_1$;

questo è rispettato in entrambi i casi per la Definizione 12 (di equivalenza fra storie) visto che è possibile applicare $\overline{op}$ in s, allora non ci possono essere trasposizioni in h' che violino le side-condition della regola $\overline{op}$, e quindi è possibile applicare $\overline{op}$ a s'. □

Lemma 21. *Dato un sistema s_1 con storia $h = \mathsf{H}_2$ op H_1 in cui è possibile applicare $\overline{op}$ ottenendo s'_1, e un sistema $s_2 \equiv s_1$ con storia $h' = \mathsf{H}'_2$ op H'_1, allora applicando $\overline{op}$ a s_2 si ottiene un sistema $s'_2 \equiv s'_1$.*

Dimostrazione. Per il Lemma 20 è possibile applicare $\overline{op}$ anche ad s_2.
Applicando $\overline{op}$ a s_1 si ottiene s'_1 con storia $\mathsf{H}_2 : \mathsf{H}_1$, mentre applicando $\overline{op}$ a s_2 si ottiene s'_2 con storia $\mathsf{H}'_2 : \mathsf{H}'_1$.
Per il Lemma 18 si vede che $\mathsf{H}_2 : \mathsf{H}_1 \equiv \mathsf{H}'_2 : \mathsf{H}'_1$ e quindi che $s'_1 \equiv s'_2$. □

Lemma 22. *Se $s_1 \equiv s'_1$ e $s_1 \rightleftharpoons_{p,r,k} s_2$ allora $\exists\ s'_2$ t.c. $s_2 \equiv s'_2$ e $s'_1 \rightleftharpoons_{p,r,k} s'_2$. Graficamente:*



Dimostrazione. La dimostrazione si divide in due parti:

- se r è forward la dimostrazione si divide in due casi:
  - nel caso l'operazione non sia sulla mappa è facile mostrare che applicando a s'_1 la stessa transizione applicata a s_1, questo muova in uno stato s'_2 equivalente a s_2. Se la modifica effettuata al sistema è la stessa, allora anche Γ e Π saranno uguali a quelli di s_2 e, non essendo stata modificata la mappa, allora anch'essa e la sua storia saranno equivalenti;
  - nel caso l'operazione sia sulla mappa invece per dimostrare l'equivalenza di s_2 e s'_2 bisognerà mostrare l'equivalenza fra le storie dei due sistemi.
    In questo caso si sa che la storia di s_1 indicata con H_1 è equivalente alla storia di s'_1 indicata con H'_1. Inoltre, si sa anche che applicando a s'_1 la regola r si muoverà in s'_2 con storia $\mathsf{H}'_2 = \mathsf{H}'_1 : op$. Applicando la regola a s_1 si muoverà in uno stato s_2 con storia $\mathsf{H}_2 = \mathsf{H}_1 : op$. È facile notare che $\mathsf{H}_2 \equiv \mathsf{H}'_2$ visto che $\mathsf{H}_1 \equiv \mathsf{H}'_1$ ed è stato aggiunto alla fine della storia lo stesso elemento;
- se r è backward la dimostrazione si divide in due casi:

- nel caso l'operazione non sia sulla mappa se si applica la stessa regola backward a due sistemi equivalenti, allora muoveranno in due sistemi equivalenti visto che le modifiche al sistema sono le stesse (la mappa e la sua storia non vengono modificate e per questo rimarranno equivalenti);
- nel caso l'operazione sia sulla mappa per il Lemma 20 è possibile applicare a s'_1 la regola backward r. L'equivalenza di s_2 e s'_2 si può vedere grazie al Lemma 21. □

Lemma 23 (Loop lemma)**.** *Per ogni coppia di sistemi raggiungibili, s_1 e s_2, si ha $s_1 \rightharpoonup_{p,r,k} s_2$ sse esiste s'_1, $s_2 \leftharpoondown_{p,\bar{r},k} s'_1$ tale per cui $s_1 \equiv s'_1$. Graficamente:*

$$s_1 \xrightarrow{p,r,k} s_2 \quad \Longleftrightarrow \quad s_1 \equiv s'_1 \xleftarrow[p,r,k]{} s_2$$

Dimostrazione. La prova è l'analisi dei casi sulla regola applicata. Di seguito verranno discussi i casi più interessanti.

- Regola forward sulla mappa: in questo caso è possibile applicare successivamente la regola backward corrispondente perché vengono rispettate tutte le condizioni richieste, essendo l'ultima regola applicata alla mappa è quella forward. In questo caso si può facilmente vedere che si torna allo stato originale.

- Regola backward sulla mappa: in questo caso se si può applicare la regola backward vuol dire che sono rispettate tutte le condizioni richieste dalla regola. Al sistema quindi sarà possibile applicare la regola forward successivamente tornando ad uno stato equivalente al precedente. Questo perché, se sono state rispettate le condizioni imposte dalla regola, allora nella storia:

  - se la regola di backward è in lettura, allora non sono presenti operazioni in scrittura che coinvolgono gli stessi atomi o pid coinvolti dalla regola forward,
  - se la regola è in scrittura, allora nella storia non sono presenti operazioni in scrittura o lettura che coinvolgono gli stessi atomi o pid coinvolti dalla regola forward. □

Definizione 24 (Transizioni concorrenti). Due transizioni coiniziali, $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ e $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$, vengono definite *in conflitto* se almeno una delle seguenti condizioni sussiste:

- **entrambe le transizioni sono forward**, considerano lo stesso processo, cioè $p_1 = p_2$, e $r_1 = r_2 = \mathit{Sched}$ o una transizione applica la regola *Sched* e l'altra la regola *Receive*.

- le regole viste nella Tabella 2.1 presente nella Definizione 4

- una è una transizione **forward** che si applica a un processo p, si definisce $p_1 = \mathrm{p}$, e l'altra è una transizione **backward** che annulla la creazione di p, ovvero $p_2 = \mathrm{p}' \neq \mathrm{p}$, $r_2 = \overline{\mathit{Spawn}}$ e $k_2 = \mathsf{spawn}(\theta, e, \mathrm{p})$ per qualche controllo (θ, e);

- una è una transizione **forward** che consegna un messaggio $\{v, \lambda\}$ a un processo p, si definisce $p_1 = \mathrm{p}$, $r_1 = \mathit{Sched}$ e $k_1 = \mathsf{sched}(\{v,\ k\})$, e l'altra è una transizione **backward** che annulla l'invio di $\{v,\ k\}$ a p, ovvero $p_2 = \mathrm{p}'$ (nota che $\mathrm{p} = \mathrm{p}'$ se il messaggio viene inviato a se stessi), $r_2 = \overline{\mathit{Send}}$ e $k_2 = \mathsf{send}(\theta, \mathsf{e}, \mathrm{p}, \{\mathsf{v},\ \mathsf{k}\})$ per qualche controllo (θ, e);

- una è una transizione **forward** e l'altra è **backward** tale che $p_1 = p_2$ e i) entrambe le regole applicate sono diverse sia da *Sched* che da $\overline{\mathit{Sched}}$, ovvero $\{r_1, r_2\} \cap \{\mathit{Sched}, \overline{\mathit{Sched}}\} = \emptyset$; ii) una regola è *Sched* e l'altra è $\overline{\mathit{Sched}}$; iii) una regola è *Sched* e l'altra è $\overline{\mathit{Receive}}$; o iv) una regola è $\overline{\mathit{Sched}}$ e l'altra è *Receive*.

- nel caso in cui entrambe le transizioni siano sulla mappa, se queste accedono ad uno stesso pid o atomo e se una delle due è in scrittura sulla mappa (se entrambe le transizioni sono in lettura il conflitto non è presente);

- nel caso in cui entrambe le transizioni siano sulla mappa e una transizione sia **forward** e l'altra **backward** i casi di conflitto si vedono nella Tabella 2.2

  Nella Tabella 2.2 sono presenti i casi in cui si verificano i conflitti; a' e p' rappresentano l'atomo e il pid coinvolti da r_1, mentre a'' e p'' rappresentano l'atomo e il pid coinvolto da r_2. M' rappresenta gli elementi letti nella mappa dalla regola $\mathsf{Call3}$ o $\overline{\mathsf{Call3}}$. Le celle vuote rappresentano transizioni coiniziali che non vanno in conflitto.
  Si nota che esistono coppie di operazini in scrittura che non sono in conflitto visto che nei casi in cui sarebbe presente il conflitto queste operazioni non possono essere coiniziali.

Due transizioni coiniziali sono *concorrenti* se non sono in conflitto. Si può notare che due transizioni backward coiniziali non sono mai in conflitto. □

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

Tabella 2.2: Tabella dei conflitti per le regole forward-backward, nel caso delle regole backward-backward non è presente una tabella perchè non è presente il conflitto fra due operazioni backword.

Il seguente lemma è la controparte del Lemma 25 per la semantica standard.

Lemma 25 (Square lemma)**.** *Date due operazioni coiniziali concorrenti* $t_1 = (s \rightleftharpoons_{p_1,r_1,k_1} s_1)$ *e* $t_2 = (s \rightleftharpoons_{p_2,r_2,k_2} s_2)$*, allora esistono due transizioni* $t_2/t_1 = (s_1 \rightleftharpoons_{p_2,r_2,k_2} s')$*,* $t_1/t_2 = (s_2 \rightleftharpoons_{p_1,r_1,k_1} s'')$ *tali per cui* $s' \equiv s''$*. Graficamente,*

$$\begin{array}{ccc} s & \overset{p_1,r_1,k_1}{\rightleftharpoons} & s_1 \\ {\scriptstyle p_2,r_2,k_2}\Big\| & & \\ s_2 & & \end{array} \quad \Longrightarrow \quad \begin{array}{ccc} s & \overset{p_1,r_1,k_1}{\rightleftharpoons} & s_1 \\ {\scriptstyle p_2,r_2,k_2}\Big\| & & \Big\|{\scriptstyle p_2,r_2,k_2} \\ s_2 & \underset{p_1,r_1,k_1}{\rightleftharpoons} & s'' \equiv s' \end{array}$$

Dimostrazione. Si distinguono i seguenti casi a seconda delle regole applicate:

(1) Due transizioni forward. Quindi, si hanno i seguenti casi:

- $r_1 = \mathsf{End}$ e $r_2 = \mathsf{RegisterF}$ nel caso in cui $a \; \varepsilon \; r_1$, $p_1 \; \varepsilon \; r_1$, $a' \; \varepsilon \; r_2$ e $p' \; \varepsilon \; r_2$ dove a e a' sono atomi e p' è un pid:
  - se $a \neq a'$ e $p_1 \neq p'$, allora si nota che applicando prima la transizione t_1 e poi la t_2 o viceversa si arriva in due stati rispettivamente s' e s'' che sono equivalenti. Questo perché la mappa è la stessa

mentre la storia della mappa differisce per gli ultimi elementi che sono scambiati. Visto che $a \neq a'$ e $p_1 \neq p'$ viene rispettata la Definizione 12 (equivalenza fra storie) e quindi $s' \equiv s''$.

- se $a = a'$ o $p_1 = p'$ le due operazioni sono in conflitto (vedere Tabella 2.1).

- questo ragionamento si può effettuare su tutte le regole sulla mappa

(2) Una transizione forward e una backward. Quindi, si distnguono i seguenti casi:

- $r_1 = \mathsf{RegisterT}$ e $r_2 = \overline{\mathsf{UnregisterT}}$ nel caso in cui $a \; \varepsilon \; r_1$, $p \; \varepsilon \; r_1$, $a' \; \varepsilon \; r_2$ e $p' \; \varepsilon \; r_2$, dove a e a' sono atomi e p e p' sono pid:

  - se $a \neq a'$ e $p \neq p'$, allora applicando prima la transizione t_1 e poi t_2 o viceversa si ottengono due transizioni t_1/t_2 e t_2/t_1 che finiscono in due stati equivalenti (la storia della mappa sarà la stessa sia in s' che in s'').
  - se $a = a'$ o $p = p'$ le due operazioni sono in conflitto (vedere Tabella 2.2).

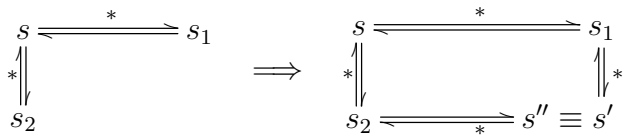
- questo ragionamento si può effettuare su tutte le regole sulla mappa

(3) Due transizioni backward. Quindi, si distinguono i seguenti casi:

- Nel caso di due operazioni sulla mappa (o anche in cui solo una è sulla mappa), è sempre verificato:

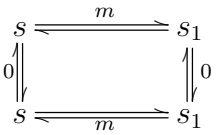
  - nel caso in cui entrambe le operazioni backward sono sulla mappa, allora le condizioni di entrambe le regole sono rispettate, quindi sarà indifferente applicare prima una o l'altra (la storia della mappa sarà la stessa sia in s' che in s'').
  - nel caso in cui una regola è sulla mappa e l'altra no, allora le condizioni di entrambe le regole sono rispettate; applicare prima la regola sulla mappa o prima la regola non sulla mappa non cambia il risultato (la storia della mappa sarà la stessa sia in s' che in s''). □

Lemma 26 (Confluenza)**.** *Date due operazioni coiniziali concorrenti* $t_1 = (s \rightleftharpoons^* s_1)$ *e* $t_2 = (s \rightleftharpoons^* s_2)$*, allora* $t_2/t_1^* = (s_1 \rightleftharpoons^* s')$*,* $t_1/t_2^* = (s_2 \rightleftharpoons *s'')$ *tali per cui* $s' \equiv s''$*. Graficamente,*

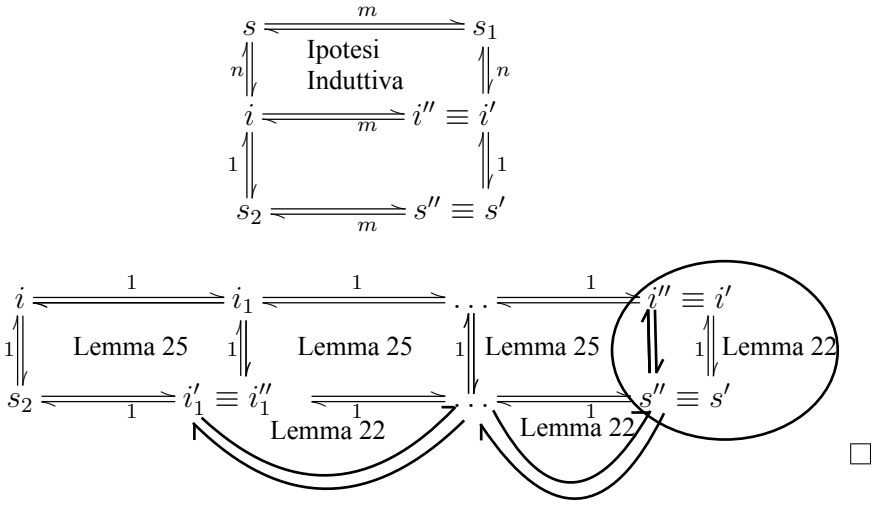


Dimostrazione. Per il Lemma 22 è possibile applicare una stessa regola a stati equivalenti e ottenere due stati equivalenti. Quindi è possibile iterare più volte il Lemma 25 a stati equivalenti. Formalmente la prova avviene per induzione sul numero di passi e poi applicando il Lemma 22 e 25, m volte (dove m è il numero di passi su cui non si è fatta induzione).
La dimostrazione è per induzione sul numero di passi:

- Caso base:



- Caso induttivo $(n+1)$



□

2.4 Semantica rollback

In questa sezione verrà introdotto un'operatore di rollback che avvia un calcolo reversibile per un processo. L'operazione di annullamento (non deterministica) ha alcune somiglianze con l'operatore di rollback di [12, 13]. I processi vengono messi in modalità di "rollback" utilizzando $\lfloor\ \rfloor_\Psi$, dove Ψ è l'insieme di rollback. Un tipico rollback si riferisce a un checkpoint che deve essere superato da una computazione backward del processo prima di riprendere una computazione forward. Per essere precisi, si distinguono le seguenti tipologie di rollback:

- $\#^{\mathtt{t}}_{\mathsf{ch}}$, dove "ch" sta per "checkpoint": un rollback per annullare le azioni di un processo fino a che non ha raggiunto il checkpoint con identificatore $\mathtt{t}$;

- $\#_{\mathsf{sp}}$, dove "sp" sta per "spawn": un rollback per annullare *tutte* le azioni di un processo, infine cancellandolo dal sistema;
- $\#^{\lambda}_{\mathsf{sch}}$, dove "sch" sta per "sched": un rollback per annullare le azioni di un processo fino a che la consegna di un messaggio, nella forma $\{v, \lambda\}$, non è annullata.
- $\#^{\mathtt{par}}_{\mathsf{op}}$: richiede un rollback per annullare le azioni di un processo fino a quando non viene raggiunto l'operatore con identificatore $\mathtt{op}(\mathtt{par})$;
- $\#^{\mathsf{S}'}_{([\mathsf{a}],[\mathsf{p}]),\mathsf{t}}$: si applica ad un insieme di processi Π,
  - se $S' = \emptyset$ allora $\lfloor \Pi \rfloor_{\#^{\emptyset}_{([\mathsf{a}],[\mathsf{p}]),\mathsf{t}}} = \Pi$
  - se $S' \neq \emptyset$ allora $\lfloor \Pi \rfloor_{\#^{\mathsf{S}'}_{([\mathsf{a}],[\mathsf{p}]),\mathsf{t}}} = \lfloor \Pi' \rfloor_{\#^{\mathsf{S}''}_{([\mathsf{a}],[\mathsf{p}]),\mathsf{t}}}$ dove $\mathsf{S}' = op(par) : \mathsf{S}''$ e dove:
    - se $a \notin op(par)$ e $p \notin op(par)$ allora $\Pi' = \Pi$
    - se $a \,\varepsilon\, op(par)$ o $p \,\varepsilon\, op(par)$ allora $\Pi' = \Pi \cup \{newproc\} \setminus \{proc\}$ dove:
      - $proc = \lfloor \langle p', (\theta, e), q, h \rangle \rfloor_{\Psi} \in \Pi$ (è il processo che ha eseguito l'operazione $op(par)$ che viene identificato da p' che è presente in par);
      - $newproc = \lfloor \langle p', (\theta, e), q, h \rangle \rfloor_{\Psi'}$ con $\Psi' = \Psi \cup \#^{par}_{op}$ se $\#^{par}_{op} \notin \Psi$, o altrimenti $\Psi' = \Psi$

    Se $t = l$, allora $\Pi' = \Pi$ anche se $op(par)$ è un'operazione in lettura.

Di seguito, al fine di semplificare le regole di riduzione, si assume che la semantica descritta soddisfi la seguente *equivalenza strutturale*:

$$(SC) \quad \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\emptyset} \mid \Pi \;\equiv\; \Gamma; \langle p, h, (\theta, e), q \rangle \mid \Pi$$

Si può notare che solo il primo dei tipi di rollback sopra indicati è destinato a un checkpoint. Questo tipo di checkpoint è introdotto in modo non deterministico dalla regola seguente, dove viene indicata con $\rightsquigarrow$ la nuova relazione di riduzione che modella i movimenti all'indietro della semantica di rollback:

$$(\overline{Undo}) \quad \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi} \mid \Pi \rightsquigarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_{\Psi \cup \{\#^{\mathsf{t}}_{\mathsf{ch}}\}} \mid \Pi$$
$$\text{if } \mathsf{check}(\theta', e', \mathtt{t}) \text{ occurs in } h, \text{ for some } \theta' \text{ and } e'$$

Solo dopo l'applicazione di questa regola i passaggi possono essere annullati, poiché il calcolo predefinito nella semantica di rollback è in avanti.

$$(\overline{Seq})\quad \Gamma;\lfloor\langle p,\tau(\theta,e):h,(\theta',e'),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S} \rightharpoonup \Gamma;\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S}$$

$$(\overline{Check})\quad \Gamma;\lfloor\langle p,\mathsf{check}(\theta,e,\mathtt{t}):h,(\theta',e'),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S} \rightharpoonup \Gamma;\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_{\Psi\setminus\{\#^{\mathtt{t}}_{\mathsf{ch}}\}} \mid \Pi;\ \mathsf{M};\mathsf{S}$$

$$(\overline{Send1})\quad \Gamma\cup\{(p',\{v,\lambda\})\};\lfloor\langle p,\mathsf{send}(\theta,e,p',\{v,\lambda\}):h,(\theta',e'),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S} \rightharpoonup \Gamma;\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S}$$

$$(\overline{Send2})\quad \begin{array}{l}\Gamma;\lfloor\langle p,\mathsf{send}(\theta,e,p',\{v,\lambda\}):h,(\theta',e'),q\rangle\rfloor_\Psi \mid \lfloor\langle p',h',(\theta'',e''),q'\rangle\rfloor_{\Psi'} \mid \Pi;\ \mathsf{M};\mathsf{S}\\ \rightharpoonup \Gamma;\lfloor\langle p,\mathsf{send}(\theta,e,p',\{v,\lambda\}):h,(\theta',e'),q\rangle\rfloor_\Psi \mid \lfloor\langle p',h',(\theta'',e''),q'\rangle\rfloor_{\Psi'\cup\{\#^{\lambda}_{\mathsf{sch}}\}} \mid \Pi;\ \mathsf{M};\mathsf{S}\\ \qquad\text{if } (p',\{v,\lambda\}) \text{ does not occur in } \Gamma \text{ and } \#^{\lambda}_{\mathsf{sch}}\notin\Psi'\end{array}$$

$$(\overline{Receive})\quad \Gamma;\lfloor\langle p,\mathsf{rec}(\theta,e,\{v,\lambda\},q):h,(\theta',e'),q\backslash\!\backslash\{v,\lambda\}\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S} \rightharpoonup \Gamma;\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S}$$

$$(\overline{Spawn1})\quad \begin{array}{l}\Gamma;\lfloor\langle p,\mathsf{spawn}(\theta,e,p''):h,(\theta',e'),q\rangle\rfloor_\Psi \mid \lfloor\langle[\,],p'',(\theta'',e''),[\,]\rangle\rfloor_{\Psi'} \mid \Pi;\ \mathsf{M};\mathsf{S}\\ \rightharpoonup \Gamma;\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S}\end{array}$$

$$(\overline{Spawn2})\quad \begin{array}{l}\Gamma;\lfloor\langle p,\mathsf{spawn}(\theta,e,p''):h,(\theta,e),q\rangle\rfloor_\Psi \mid \lfloor\langle p'',h'',(\theta'',e''),q''\rangle\rfloor_{\Psi'} \mid \Pi;\ \mathsf{M};\mathsf{S}\\ \rightharpoonup \Gamma;\lfloor\langle p,\mathsf{spawn}(\theta,e,p''):h,(\theta,e),q\rangle\rfloor_\Psi \mid \lfloor\langle p'',h'',(\theta'',e''),q''\rangle\rfloor_{\Psi'\cup\{\#_{\mathsf{sp}}\}} \mid \Pi;\ \mathsf{M};\mathsf{S}\\ \qquad\text{if } h''\neq[\,]\vee q''\neq[\,] \text{ and } \#_{\mathsf{sp}}\notin\Psi'\end{array}$$

$$(\overline{Self})\quad \Gamma;\lfloor\langle p,\mathsf{self}(\theta,e):h,(\theta',e'),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S} \rightharpoonup \Gamma;\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S}$$

$$(\overline{Sched})\quad \begin{array}{l}\Gamma;\lfloor\langle p,h,(\theta,e),\{v,\lambda\}:q\rangle\rfloor_\Psi \mid \Pi;\ \mathsf{M};\mathsf{S} \rightharpoonup \Gamma\cup(p,\{v,\lambda\});\lfloor\langle p,h,(\theta,e),q\rangle\rfloor_{\Psi\setminus\{\#^{\lambda}_{\mathsf{sch}}\}} \mid \Pi;\ \mathsf{M};\mathsf{S}\\ \qquad\text{if the topmost } \mathsf{rec}(\ldots) \text{ item in } h \text{ (if any) has the}\\ \qquad\text{form } \mathsf{rec}(\theta',e',\{v',\lambda'\},q') \text{ with } q'\backslash\!\backslash\{v',\lambda'\}\neq\{v,\lambda\}:q\end{array}$$

Figura 2.12: Rollback semantics: backward reduction rules

Le regole backward della semantica di rollback sono mostrate nelle Figure 2.12, 2.13 e 2.14. Si assume che $\Psi \neq \emptyset$ (ma Ψ' potrebbe essere vuoto).

Si può notare che i rollback fino ai checkpoint vengono generati non deterministicamente dalla regola $\overline{Undo}$, mentre gli altri due tipi di rollback sono generati dalle regole di riduzione backward in un certo ordine per garantire la coerenza causale.
Questo è chiarito dalla discussione seguente, dove si spiegano brevemente le principali differenze rispetto alla semantica backward descritta precedentemente:

- Come nella semantica nella Figura 2.10, l'invio di un messaggio può essere annullato quando il messaggio è ancora nella mailbox globale (regola $\overline{Send1}$). Altrimenti, potrebbe essere necessario applicare prima la regola $\overline{Send2}$ per "propagare" la modalità di rollback fino al destinatario del messaggio, in modo che le regole $\overline{Sched}$ e $\overline{Send1}$ possano essere eventualmente applicate.

- Per l'annullamento della generazione di un processo p'', la regola $\overline{Spawn1}$

$$(\overline{SendA1}) \quad \begin{array}{l} \Gamma \cup \{(p', \{v, \lambda\})\}; \lfloor \langle p, \mathsf{sendA}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \\ \mathsf{S}' : \mathsf{sendA}(\theta, e, p', \{v, \lambda\}, a, p) : \mathsf{S} \twoheadleftarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{S}' : \mathsf{S} \\ Se\ \forall\ op \text{ in scrittura} \in \mathsf{S}'\ a \notin op \end{array}$$

$$(\overline{SendA2}) \quad \begin{array}{l} \Gamma \cup \{(p', \{v, \lambda\})\}; \lfloor \langle p, \mathsf{sendA}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{S}' : \mathsf{sendA}(\theta, e, p', \{v, \lambda\}, a, p) : \mathsf{S} \twoheadleftarrow \\ \Gamma \cup \{(p', \{v, \lambda\})\}; \lfloor \langle p, \mathsf{sendA}(\theta, e, p', \{v, \lambda\}) : h, (\theta, e), q \rangle \rfloor_\Psi \mid \lfloor \Pi \rfloor_{\#^{\mathsf{S}'}_{([\mathsf{a}], \emptyset), \mathsf{I}}}; \\ \mathsf{S}' : \mathsf{sendA}(\theta, e, p', \{v, \lambda\}, a, p) : \mathsf{S} \\ Se\ \exists\ op \text{ in scrittura} \in \mathsf{S}'\ a\ \varepsilon\ op \end{array}$$

$$(\overline{SendA3}) \quad \begin{array}{l} \Gamma; \lfloor \langle p, \mathsf{sendA}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p', h', (\theta'', e''), q' \rangle \rfloor_{\Psi'} \mid \Pi; \\ \mathsf{S}' : \mathsf{sendA}(\theta, e, p', \{v, \lambda\}, a, p) : \mathsf{S} \\ \twoheadleftarrow \Gamma; \lfloor \langle p, \mathsf{sendA}(\theta, e, p', \{v, \lambda\}) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \langle p', h', (\theta'', e''), q' \rangle \rfloor_{\Psi' \cup \{\#^{\lambda}_{\mathsf{sch}}\}} \mid \Pi; \\ \mathsf{S}' : \mathsf{sendA}(\theta, e, p', \{v, \lambda\}, a, p) : \mathsf{S} \\ \text{se } (p', \{v, \lambda\}) \text{ non occorre in } \Gamma \text{ e } \#^{\lambda}_{\mathsf{sch}} \notin \Psi' \end{array}$$

$$(\overline{SendF1}) \quad \begin{array}{l} \Gamma; \lfloor \langle p, \mathsf{sendF1}(\theta, e, p, a) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{sendF1}(\theta, e, p, a) : \mathsf{S} \\ \twoheadleftarrow \Gamma; \lfloor \langle p, h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \\ Se\ \forall\ op \text{ in scrittura} \in \mathsf{S}'\ a \notin op \end{array}$$

$$(\overline{SendF2}) \quad \begin{array}{l} \Gamma; \lfloor \langle p, \mathsf{sendF1}(\theta, e, p, a) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{sendF1}(\theta, e, p, a) : \mathsf{S} \\ \twoheadleftarrow \Gamma; \lfloor \langle p, \mathsf{sendF1}(\theta, e, p, a) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \Pi \rfloor_{\#^{\mathsf{S}'}_{([\mathsf{a}], \emptyset), \mathsf{I}}}; \mathsf{M}; \mathsf{S}' : \mathsf{sendF1}(\theta, e, p, a) : \mathsf{S} \\ Se\ \exists\ op \text{ in scrittura} \in \mathsf{S}'\ a\ \varepsilon\ op \end{array}$$

$$(\overline{RegisterT1}) \quad \begin{array}{l} \Gamma; \lfloor \langle p, \mathsf{regT}(\theta, e, a, p', p) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M} \cup (a, p'); \mathsf{S}' : \mathsf{regT}(\theta, e, a, p', p) : \mathsf{S} \\ \twoheadleftarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \\ Se\ \forall\ op \text{ in scrittura o lettura} \in \mathsf{S}'\ a \notin op \circ p' \notin op \end{array}$$

$$(\overline{RegisterT2}) \quad \begin{array}{l} \Gamma; \lfloor \langle p, \mathsf{regT}(\theta, e, a, p', p) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{regT}(\theta, e, a, p', p) : \mathsf{S} \\ \twoheadleftarrow \Gamma; \lfloor \langle p, \mathsf{regT}(\theta, e, a, p', p) : h, (\theta, e), q \rangle \rfloor_\Psi \mid \lfloor \Pi \rfloor_{\#^{\mathsf{S}'}_{([\mathsf{a}], [\mathsf{p}']), \mathsf{s}}}; \mathsf{M}; \mathsf{S}' : \mathsf{regT}(\theta, e, a, p', p) : \mathsf{S} \\ Se\ \exists\ op \text{ in scrittura o lettura} \in \mathsf{S}'\ a\ \varepsilon\ op \circ p'\ \varepsilon\ op \end{array}$$

$$(\overline{RegisterF1}) \quad \begin{array}{l} \Gamma; \lfloor \langle p, \mathsf{regF1}(\theta, e, a, p', p) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{regF1}(\theta, e, a, p', p) : \mathsf{S} \\ \twoheadleftarrow \Gamma; \lfloor \langle p, h, (\theta, e), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \\ Se\ \forall\ op \text{ in scrittura} \in \mathsf{S}'\ a \notin op \circ p' \notin op \end{array}$$

$$(\overline{RegisterF2}) \quad \begin{array}{l} \Gamma; \lfloor \langle p, \mathsf{regF1}(\theta, e, a, p', p) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{regF1}(\theta, e, a, p', p) : \mathsf{S} \\ \twoheadleftarrow \Gamma; \lfloor \langle p, \mathsf{regF1}(\theta, e, a, p', p) : h, (\theta', e'), q \rangle \rfloor_\Psi \mid \lfloor \Pi \rfloor_{\#^{\mathsf{S}'}_{([\mathsf{a}], [\mathsf{p}']), \mathsf{I}}}; \mathsf{M}; \mathsf{S}' : \mathsf{regF1}(\theta, e, a, p', p) : \mathsf{S} \\ Se\ \exists\ op \text{ in scrittura} \in \mathsf{S}'\ a\ \varepsilon\ op \circ p'\ \varepsilon\ op \end{array}$$

Figura 2.13: Rollback semantics: backward reduction rules

si applica quando sia la cronologia che le code del processo generato p'' sono vuote, eliminando sia l'elemento della cronologia in p che il processo p''. Altrimenti, si applica la regola $\overline{Spawn2}$ che propaga la modalità di rollback in modo che, alla fine, la regola $\overline{Spawn1}$ si possa

$$(\overline{UnregisterT1}) \quad \begin{array}{l}\Gamma; \lfloor\langle p, \mathsf{unregT}(\theta, e, a, p', p) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M} \backslash\backslash (a, p'); \mathsf{S}' : \mathsf{unregT}(\theta, e, a, p', p) : \mathsf{S} \\ \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \\ Se\ \forall\ op \text{ in scrittura o lettura} \in \mathsf{S}'\ a \notin op \circ p' \notin op\end{array}$$

$$(\overline{UnregisterT2}) \quad \begin{array}{l}\Gamma; \lfloor\langle p, \mathsf{unregT}(\theta, e, a, p', p) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{unregT}(\theta, e, a, p', p) : \mathsf{S} \\ \leftharpoondown \Gamma; \lfloor\langle p, \mathsf{unregT}(\theta, e, a, p', p) : h, (\theta, e), q\rangle\rfloor_\Psi \mid \lfloor\Pi\rfloor_{\#^{\mathsf{S}'}_{([\mathsf{a}],[\mathsf{p}']),\mathsf{s}}}; \mathsf{M}; \mathsf{S}' : \mathsf{unregT}(\theta, e, a, p', p) : \mathsf{S} \\ Se\ \exists\ op \text{ in scrittura o lettura} \in \mathsf{S}'\ a\ \varepsilon\ op \circ p'\ \varepsilon\ op\end{array}$$

$$(\overline{UnregisterF1}) \quad \begin{array}{l}\Gamma; \lfloor\langle p, \mathsf{unregF1}(\theta, e, a, p) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{unregF1}(\theta, e, a, p) : \mathsf{S} \\ \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \\ Se\ \forall\ op \text{ in scrittura} \in \mathsf{S}'\ a \notin op\end{array}$$

$$(\overline{UnregisterF2}) \quad \begin{array}{l}\Gamma; \lfloor\langle p, \mathsf{unregF1}(\theta, e, a, p) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{unregF1}(\theta, e, a, p) : \mathsf{S} \\ \leftharpoondown \Gamma; \lfloor\langle p, \mathsf{unregF1}(\theta, e, a, p) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \lfloor\Pi\rfloor_{\#^{\mathsf{S}'}_{([\mathsf{a}],[\emptyset]),\mathsf{I}}}; \mathsf{M}; \mathsf{S}' : \mathsf{unregF1}(\theta, e, a, p) : \mathsf{S} \\ Se\ \exists\ op \text{ in scrittura} \in \mathsf{S}'\ a\ \varepsilon\ op\end{array}$$

$$(\overline{Call3-1}) \quad \begin{array}{l}\Gamma; \lfloor\langle p, \tau\mathsf{MM}'(\theta, e, p) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \tau\mathsf{MM}'(\theta, \mathsf{e}, \mathsf{p}) : \mathsf{S} \\ \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \\ \text{Se in S' non sono presenti operazioni di scrittura su elementi presenti in M'}\end{array}$$

$$(\overline{Call3-2}) \quad \begin{array}{l}\Gamma; \lfloor\langle p, \tau\mathsf{MM}'(\theta, e, p) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \tau\mathsf{MM}'(\theta, \mathsf{e}, \mathsf{p}) : \mathsf{S} \leftharpoondown \\ \Gamma; \lfloor\langle p, \tau\mathsf{MM}'(\theta, e, p) : h, (\theta', e'), q\rangle\rfloor_\Psi \mid \lfloor\Pi\rfloor_{\#^{\mathsf{S}'}_{(\mathsf{M}'),\mathsf{I}}}; \mathsf{M}; \mathsf{S}' : \tau\mathsf{MM}'(\theta, \mathsf{e}, \mathsf{p}) : \mathsf{S} \\ \text{Se in S' sono presenti operazioni di scrittura su elementi presenti in M'}\end{array}$$

$$(\overline{Catch}) \quad \Gamma; \lfloor\langle p, \mathsf{fail}(\theta, e) : h, (\theta, fail), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S} \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}$$

$$(\overline{End1}) \quad \begin{array}{l}\Gamma; \lfloor\langle p, \mathsf{end}(\theta, \epsilon, a', p) : h, (\theta, \epsilon), q\rangle\rfloor_\Psi \mid \Pi, \mathsf{M} \backslash\backslash (a', p), \mathsf{S}' : \mathsf{end}(\theta, \epsilon, a', p) : \mathsf{S} \leftharpoondown \Gamma; \lfloor\langle p, h, (\theta, \epsilon), q\rangle\rfloor_\Psi \mid \Pi; \mathsf{M}; \mathsf{S}' : \mathsf{S} \\ Se\ \forall\ op \text{ in scrittura o lettura} \in \mathsf{S}'\ a' \notin op \circ p \notin op\end{array}$$

$$(\overline{End2}) \quad \begin{array}{l}\Gamma; \lfloor\langle p, \mathsf{end}(\theta, \epsilon, a', p) : h, (\theta, \epsilon), q\rangle\rfloor_\Psi \mid \Pi, \mathsf{M}, \mathsf{S}' : \mathsf{end}(\theta, \epsilon, a', p) : \mathsf{S} \leftharpoondown \\ \Gamma; \lfloor\langle p, \mathsf{end}(\theta, \epsilon, a', p) : h, (\theta, \epsilon), q\rangle\rfloor_\Psi \mid \lfloor\Pi\rfloor_{\#^{\mathsf{S}'}_{([\mathsf{a}'],[\mathsf{p}]),\mathsf{s}}}; \mathsf{M}, \mathsf{S}' : \mathsf{end}(\theta, \epsilon, a', p) : \mathsf{S} \\ Se\ \exists\ op \text{ in scrittura o lettura} \in \mathsf{S}'\ a'\ \varepsilon\ op \circ p\ \varepsilon\ op\end{array}$$

Figura 2.14: Rollback semantics: backward reduction rules

applicare a p''.

- Si osserva che la regola $\overline{Sched}$ richiede la stessa side-condition della semantica incontrollata. Questo è necessario per evitare la commutazione delle regole $\overline{Receive}$ e $\overline{Sched}$.

- Infine, si può notare che per ogni nuova regola $\leftharpoondown$ sulla mappa sono state aggiunte due regole $\leftharpoondown\!\!\!\!\leftharpoondown$: una per il caso in cui in S' siano presenti operazioni eseguite sugli stessi pid e/o atomi che inseriscono gli ope-

ratori di rollback ove necessario, l'altra che effettua effettivamente la regola di rollback dell'operazione, visto che in S' non sono presenti le regole che non consentono il rollback.

La semantica di rollback è modellata dalla relazione $\not\hookrightarrow$, che è definita come l'unione della relazione reversibile diretta $\rightharpoonup$ (Figure 2.8 e 2.9) e la relazione a ritroso $\not\leftharpoonup$ definita nelle Figure 2.12, 2.13 e 2.14. Si noti che, in contrasto con la semantica reversibile (non controllata) della Sezione 2.3, la semantica di rollback data dalla relazione $\not\hookrightarrow$ ha meno scelte non deterministiche: tutti i calcoli vengono eseguiti in avanti tranne quando un'azione di rollback richiede alcuni passi all'indietro per ripristinare uno stato precedente di un processo, che può essere propagato ad altri processi al fine di annullare la generazione di un processo o l'invio di un messaggio.
Si noti, tuttavia, che oltre all'introduzione dei rollback, è ancora presente un certo livello di non determinismo nelle regole all'indietro della semantica del rollback: da un lato, la selezione del processo quando ci sono diversi rollback in corso è non deterministica e dall'altro, in molti casi, sia la regola $\overline{Sched}$ sia un'altra sono applicabili allo stesso processo. La semantica potrebbe essere resa deterministica utilizzando una particolare strategia per selezionare i processi (es. Round robin) e applicando la regola $\overline{Sched}$ ogniqualvolta possibile (es. dare a $\overline{Sched}$ una priorità più alta rispetto alle restanti regole all'indietro).
Si vuole dimostrare di seguito la validità della semantica di rollback. Per farlo, si definisce $\mathsf{rolldel}(s)$ che denota il sistema che risulta da s rimuovendo i rollback in corso; formalmente, $\mathsf{rolldel}(\Gamma; \Pi; \mathsf{M}; \mathsf{S}) = \Gamma; \mathsf{rolldel}'(\Pi); \mathsf{M}; \mathsf{S}$, dove:

$$\begin{array}{lcl}
\mathsf{rolldel}'(\langle p, h, (\theta, e), q\rangle) & = & \langle p, h, (\theta, e), q\rangle \\
\mathsf{rolldel}'(\lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi) & = & \langle p, h, (\theta, e), q\rangle \\
\mathsf{rolldel}'(\langle p, h, (\theta, e), q\rangle \mid \Pi) & = & \langle p, h, (\theta, e), q\rangle \mid \mathsf{rolldel}'(\Pi) \\
\mathsf{rolldel}'(\lfloor\langle p, h, (\theta, e), q\rangle\rfloor_\Psi \mid \Pi) & = & \langle p, h, (\theta, e), q\rangle \mid \mathsf{rolldel}'(\Pi)
\end{array}$$

Dove si assume che Π non sia vuoto. Si estende anche la definizione di sistemi iniziali e raggiungibili alla semantica di rollback.

Definizione 27 (Sistemi raggiungibili sotto la semantica di rollback). Un sistema è *iniziale* sotto la semantica di rollback se è composto da un unico processo con un set vuoto Ψ di rollback attivi; inoltre, la storia, la coda dei massaggi, la mail-box globale, la storia della mappa e la mappa sono vuote. Un sistema s è *raggiungibile* sotto la semantica di rollback se esiste un sistema iniziale s_0 e una derivazione $s_0 \not\hookrightarrow^* s$ utilizzando le regole corrispondenti a un determinato programma.

Teorema 28 (Soundness)**.** *Sia s un sistema raggiungibile con la semantica di rollback. Se $s \not\hookrightarrow^* s'$, allora $\mathsf{rolldel}(s) \rightleftharpoons^* \mathsf{rolldel}(s')$.*

Dimostrazione. Per le transizioni in avanti la dimostrazione è banale poiché le regole in avanti sono le stesse in entrambe le semantiche e si applicano solo ai processi che non sono in fase di rollback. Per le transizioni all'indietro la dimostrazione è per casi analizzando la regola applicata, tramite l'equivalenza strutturale e l'utilizzo di rolldel:

- Regola $\overline{Undo}$: l'effetto viene rimosso da rolldel, quindi un'applicazione di questa regola corrisponde a un file derivazione a passo zero sotto la semantica incontrollata;
- Regole $\overline{Seq}$, $\overline{Check}$, $\overline{Send1}$, $\overline{Receive}$, $\overline{Spawn1}$, $\overline{Self}$, $\overline{Sched}$, $\overline{SendA1}$, $\overline{SendF1}$, $\overline{Call3-1}$, $\overline{RegisterT1}$, $\overline{RegisterF1}$, $\overline{UnregisterT1}$, $\overline{UnregisterF1}$, $\overline{Catch}$ e $\overline{End1}$: corrispondono, rispettivamente, alle regole $\overline{Seq}$, $\overline{Check}$, $\overline{Send}$, $\overline{Receive}$, $\overline{Spawn}$, $\overline{Self}$, $\overline{Sched}$, $\overline{SendA}$, $\overline{SendF}$, $\overline{Call3}$, $\overline{RegisterT}$, $\overline{RegisterF}$, $\overline{UnregisterT}$, $\overline{UnregisterF}$, $\overline{Catch}$ e $\overline{End}$ della semantica incontrollata;
- Regole $\overline{Send2}$, $\overline{Spawn2}$, $\overline{SendA2}$, $\overline{SendA3}$, $\overline{SendF2}$, $\overline{Call3-2}$, $\overline{RegisterT2}$, $\overline{RegisterF2}$, $\overline{UnregisterT2}$, $\overline{UnregisterF2}$ e $\overline{End2}$: l'effetto viene rimosso da rolldel, quindi un'applicazione di una qualsiasi di queste regole corrisponde a una derivazione a passo zero sotto semantica incontrollata. □

Ora si può mostrare la completezza della semantica di rollback a condizione che il processo coinvolto sia in modalità rollback:

Lemma 29 (Completezza in modalità rollback)**.** *Sia s un sistema raggiungibile. Se $s \rightharpoonup s'$, allora un qualsiasi sistema s_r tale che $\mathsf{rolldel}(s_r) = s$ e dove il processo che ha eseguito la transizione $s \rightharpoonup s'$ è in modalità rollback per un insieme non vuoto di rollback. Allora esiste s'_r tale che $s_r \rightharpoonup\!\!\!\!\!\rightharpoonup s'_r$ e $\mathsf{rolldel}(s'_r) = s'$.*

Dimostrazione. La prova è l'analisi dei casi sulla regola applicata. Ogni passaggio è corrispondente alla regola omonima, ma per $\overline{Send}$, $\overline{Spawn}$, $\overline{SendA}$, $\overline{SendF}$, $\overline{Call3}$, $\overline{RegisterT}$, $\overline{RegisterF}$, $\overline{UnregisterT}$, $\overline{UnregisterF}$ e $\overline{End}$ corrispondono rispettivamente le regole $\overline{Send1}$, $\overline{Spawn1}$, $\overline{SendA1}$, $\overline{SendF1}$, $\overline{Call3-1}$, $\overline{RegisterT1}$, $\overline{RegisterF1}$, $\overline{UnregisterT1}$, $\overline{UnregisterF1}$ e $\overline{End1}$. □

Capitolo 3

Lavori Futuri e Conclusione

In questa tesi è stata estesa la semantica dell'articolo [2], inizialmente è stata descritta l'estensione del linguaggio utilizzato con l'aggiunta delle funzioni:

- **register**: dati in input atomo e pid, inserisce nella mappa la coppia (atomo,pid) e restituisce l'atomo **true**. Altrimenti il processo fallisce;
- **unregister**: dato in input un atomo, essa toglie dalla mappa la coppia (atomo,pid) e restituisce l'atomo **true**. Altrimenti il processo fallisce.
- **registered**: restituisce una lista di tutti gli atomi nella mappa;
- **whereis**: dato in input un atomo restituisce il pid associato, altrimenti otterremo **undefined**.

Inoltre al linguaggio viene aggiunto anche il non terminale *end* utilizzato per la terminazione di un processo.
Successivamente è stata mostrata sia la semantica del linguaggio presentato che quella del sistema. In queste semantiche sono presenti anche le valutazioni delle funzioni aggiunte e dei fallimenti, quest'ultimi sono stati aggiunti per avere un comportamento più simile a quello di Erlang. È stata estesa anche la definizione di transizioni concorrenti in accordo con le nuove regole.
Dopo di che è stata introdotta la nozione di semantica reversibile, ovvero una semantica non deterministica che permette di eseguire operazioni all'indietro, descrivendo anche le sue principali caratteristiche.
Infine è stata descritta la semantica rollback in cui viene descritto un'operatore di rollback che serve per rendere deterministica la semantica reversibile introdotta precedentemente preservando le caratterestiche principali.
Come abbiamo già accennato prima nella sezione 1.4, CauDEr supporta un sottoinsieme funzionale del linguaggio Core Erlang; lo scopo di questo lavoro

è stato anche di estendere quel sottoinsieme con feature imperative descritte precedentemente. In altre parole, l'obiettivo è realizzare una nuova versione di CauDEr, partendo dalla versione [10], che sia anche in grado di eseguire il debug di programmi più complessi. I costrutti aggiunti sono: register, unregister, whereis e registered. L'aggiunta di questi costrutti è stata fedele alla descrizione fatta nella tesi, però sono presenti delle differenze; infatti nelle funzioni ausiliari EvalM, matchPid, matchMapReg e matchMapUnreg non è stata passata la mappa come input, per evitare di modificare il codice di tutte le funzioni che dovrebbero chiamare quelle ausiliarie, ma è stato deciso di implementare un attore che avesse accesso alla mappa e che sostituisse le funzioni ausiliarie. Inoltre la storia della mappa in CauDEr non è come quella presentata, per evitare di fare funzioni ad hoc per il confronto fra le varie storie si sono tutte estese con elementi, che non vengono considerati, per avere storie tutte della stessa lunghezza e con lo stesso tipo di elementi. Riassumendo, seguendo l'approccio in [2] sono state definite le varie semantiche, inclusi i nuovi costrutti, prestando attenzione a preservare la consistenza causale, infine è stato realizzato il debugger.

Possibili lavori futuri su questa tesi sono: l'ulteriore estensione del linguaggio con i costrutti di register globale che richiederebbero l'estensione con la semantica relativa ai nodi. Un'altra possibilità sarebbe quella di estendere la semantica con altre feature imperative come le funzioni legate a **mnesia**, un database distribuito molto utilizzato nelle applicazioni. La semantica può anche essere estesa tramite l'aggiunta delle eccezzioni, che sono operatori di controllo che non sfruttano il fallimento come side-condition per segnalare un errore come le funzioni aggiunte, e che vengono utilizzate per la gestione dei fallimenti.
Un altro possibile lavoro sarebbe quello di implementare una semantica alternativa a quella presentata in cui, al posto di creare una mappa globale imperativa, si utilizza un attore che la sostituisce. In questo modo non bisognerebbe estendere la semantica; le funzioni sarebbero considerate come dei messaggi inviati all'attore, che corrisponde alla mappa, e i risultati sarebbero dei messaggi inviati dall'attore-mappa al processo che ha invocato la funzione. Dopo di che sarebbe interessante dimostrare l'equivalenza delle due semantiche. Le maggiori difficoltà in questa dimostrazione sarebbero nella gestione dei processi terminati e dei fallimenti, ad esempio nel caso in cui viene deregistrato un processo terminato. Questa difficoltà è presente in quanto in questo lavoro è stata modificata la semantica per gestire la terminazione di processi.

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] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr: A causal-consistent reversible debugger for Erlang. In J. P. Gallagher and M. Sulzmann, editors, *Proceedings of the 14th International Symposium on Functional and Logic Programming (FLOPS 2018)*, volume 10818 of *Lecture Notes in Computer Science*, pages 247–263. Springer-Verlag, Berlin, 2018.

[9] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr website. URL: `https://github.com/mistupv/cauder`.

[10] I. Lanese, N. Nishida, A. Palacios, and G. Vidal. CauDEr website. URL: `https://github.com/mistupv/cauder-v2`.

[11] 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`.

[12] I. Lanese, C. A. Mezzina, A. Schmitt, and J. Stefani. Controlling reversibility in higher-order pi. In J. Katoen and B. König, editors, *Proceedings of the 22nd International Conference on Concurrency Theory (CONCUR 2011)*, volume 6901 of *Lecture Notes in Computer Science*, pages 297–311. Springer, 2011.

[13] E. Giachino, I. Lanese, and C. A. Mezzina. Causal-consistent reversible debugging. In S. Gnesi and A. Rensink, editors, *Proc. of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE 2014)*, volume 8411 of *Lecture Notes in Computer Science*, pages 370–384. Springer, 2014.