Università degli Studi di Padova

Dipartimento di Matematica

Corso di Laurea Magistrale in Informatica

Un calcolo formale per gli attori Scala Akka e i loro tipi

Laureando

Giulia Brusaferro

Relatore

Prof. Silvia Crafa

Anno Accademico 2015/2016

Sommario

Scopo di questa tesi è presentare un calcolo formale che modelli il comportamento degli attori Scala Akka. In particolare in questo lavoro viene presentato un linguaggio A che modella la creazione degli attori, lo scambio di messaggi e il cambio di comportamento. All'interno dell'elaborato viene data evidenza di come l'esecuzione dei programmi scritti in A e in Akka coincida. Viene poi data un'estensione del linguaggio A in modo da inserire la gerarchia degli attori e la terminazione. Anche le esecuzioni dei programmi scritti nel nuovo linguaggio esteso $A+$ mostrano essere le stesse dei corrispondenti programmi scritti in Akka. Infine viene presentato un sistema di tipi per il linguaggio A che si ispira al modulo sperimentale Akka Typed e viene dimostrato che se un programma scritto nel nostro linguaggio formale è ben tipato, allora si può garantire che, a tempo di esecuzione, ogni attore sarà in grado di gestire tutti i messaggi che gli vengono inviati.

Indice

Introduzione

Il linguaggio Scala, negli ultimi anni, sta crescendo in popolarità. Secondo l'indice *PYPL (PopularitY of Programming Language)* [2] calcolato analizzando la frequenza di ricerca su Google di tutorial sui linguaggi di programmazione, a gennaio 2016 Scala si trova al sedicesimo posto per popolarità, con una crescita dello 0,3% rispetto a gennaio 2015.

Scala (acronimo di *Scalable Language*) è un linguaggio di programmazione staticamente tipato, object-oriented che miscela lo stile di programmazione imperativo con quello funzionale. È proprio l'attenta integrazione tra concetti presi da stili di programmazione diversi che rende Scala appunto scalabile [4]. Scala esegue su una JVM (Java Virtual Machine) ed è completamente integrabile con Java.

Il linguaggio Scala offre supporto alla concorrenza tramite gli attori. Il modello ad attori è un modello di computazione concorrente per sviluppare sistemi paralleli e distribuiti, introdotto da Carl Hewitt nel 1973. Gli attori sono oggetti concorrenti e asincroni, che comunicano attraverso lo scambio di messaggi e possono creare nuovi attori [10]. Lo scopo del modello è risolvere i problemi di concorrenza legati alla memoria condivisa, eliminandola completamente. La libreria di default per gli attori da Scala 2.10.0 è Akka, un toolkit open source per sviluppare applicazioni concorrenti e distribuite.

In questo elaborato presentiamo un calcolo formale il cui intento è modellare il comportamento degli attori Scala Akka. Un modello formale è utile a descrivere precisamente alcuni aspetti del design del linguaggio, ad enunciarne le proprietà e dimostrarle e talvolta ad individuare problemi che a livello più informale non erano stati individuati [6]. Un modello formale deve raggiungere un giusto compromesso tra semplicità e completezza: è necessario scegliere accuratamente che aspetti del linguaggio modellare in modo da mantenere il linguaggio formale semplice ma al tempo stesso sufficientemente espressivo.

Gli aspetti degli attori Akka che andiamo a modellare all'interno di questo elaborato sono gli aspetti che fanno di un attore un attore: lo scambio di messaggi, la creazione e (per gli attori Akka) il cambio di comportamento. Successivamente espandiamo il linguaggio formale inserendo anche la gerarchia tra attori e modellando la loro terminazione. Selezionando questi aspetti otteniamo un linguaggio espressivo ma semplice, utile per ragionare sul modello ad attori offerto da Akka.

Partendo da questo linguaggio formale costruiamo un sistema di tipi per gli attori Akka, ispirato al modulo sperimentale Akka Typed. Il modulo Akka Typed si propone di assegnare agli attori un tipo tale da poter garantire staticamente che, se un programma è ben tipato, runtime ogni attore sarà in grado di ricevere tutti i messaggi che gli verranno inviati. Con il sistema di tipi analogo scritto per il nostro linguaggio formale, dimostriamo formalmente questa proprietà.

Il resto dell'elaborato è strutturato come segue: il capitolo 1 offre una panoramica sugli attori Scala Akka e in particolare sugli aspetti che andremo poi a modellare.

Nel capitolo 2 presenteremo un linguaggio formale A che modella la creazione degli attori, lo scambio di messaggi e il cambio di comportamento. Useremo poi una serie di esempi per illustrare che i programmi scritti in A si comportano nello stesso modo dei corrispondenti programmi in Akka. In particolare l'esempio 2.3.1 illustra la creazione degli attori, sia top-level che dal corpo di un attore. Lo stesso esempio ci torna utile per mostrare che la gestione dei messaggi dei due linguaggi A e Akka è identica. L'esempio 2.3.2 illustra la gestione della mailbox. Gli esempi 2.3.3, 2.3.4, 2.3.5 illustrano il cambio di comportamento, mostrando anche come questo sia utile per implementare macchine a stati finiti. Attraverso questi esempi diamo evidenza che la modellazione che abbiamo fatto e' corretta.

Nel capitolo 3 estenderemo il linguaggio A in modo da gestire anche la gerarchia tra attori e la terminazione. Il sistema gerarchico e' una caratteristica importante degli attori Akka perché sta alla base del supervision system e quindi della programmazione fault tolerant. Gli esempi che presenteremo in questo capitolo illustreranno ciò che viene eseguito prima che un attore termini (esempio 3.2.1) e la relazione tra terminazione di un attore e terminazione dei suoi figli (esempi 3.2.2, 3.2.3), ponendo l'attenzione su cosa succede quando un attore termina dal punto di vista dell'invio dei messaggi.

Infine nel capitolo 4 presenteremo un sistema di tipi per il nostro linguaggio ispirato ad AkkaTyped. Mostreremo che in un programma ben tipato secondo il sistema di tipi dato tutti gli attori saranno in grado di gestire tutti i messaggi che gli verranno inviati a tempo di esecuzione, ovvero nessun messaggio verrà scartato. Infine modificheremo il sistema di tipi permettendo ai comportamenti di cambiare tipo nel corso dell'esecuzione e dimostreremo portando un controesempio (paragrafo 4.4) che con il sistema di tipi modificato si perde la proprietà di soundness.

Capitolo 1

Gli attori Scala Akka

Akka è un toolkit Open Source, disponibile sotto licenza *Apache 2 Licence* per costruire applicazioni concorrenti e distribuite su una JVM. Le applicazioni costruite con la piattaforma Akka si propongono di essere scalabili, elastiche e responsive, in accordo con il Reactive Manifesto [5]. Le applicazioni che seguono il modello *Reactive* mirano ad essere più tolleranti ai fallimenti ed in particolare a gestirli con eleganza nel momento in cui si verificano.

Akka è attualmente alla versione 2.4.1 per Scala 2.11.

1.1 Il modello ad attori

Il modello ad attori (i.e. *Actor Model*), è una teoria matematica nata nel 1973 utilizzata sia come framework per ragionare sulla concorrenza, sia come base teorica per l'implementazione di sistemi concorrenti [8]. Lo scopo del modello è risolvere i problemi di concorrenza legati alla memoria condivisa, eliminandola completamente. Si basa su entità, gli attori appunto, che comunicano attraverso l'invio di messaggi. Quando un attore riceve un messaggio può:

- inviare messaggi ad attori di cui conosce il nome (i.e. ne possiede l'indirizzo)
- creare nuovi attori
- decidere il comportamento da utilizzare per processare il prossimo messaggio ricevuto

Nel modello Akka creare attori e inviare messaggi è indipendente dalla locazione fisica dell'attore (*Location Transparency*); questo rende il modello adatto a implementare sistemi distribuiti senza avere conoscenza delle relazioni all'interno della rete di computer.

Gli attori in Akka sono organizzati in gerarchie chiamate *Actor Systems*. Un *Actor System* è un gruppo gerarchico di attori che condividono opzioni di configurazione. Lo scopo del sistema di attori è dividere i compiti e delegarli, in modo che ogni attore abbia una minima parte di computazione da eseguire.

1.1.1 La creazione degli attori

Creare un attore è una procedura che comporta tre passi [9]: come prima cosa si crea una classe che definisce il comportamento dell'attore, ovvero come reagisce l'attore ai messaggi che gli vengono inviati; si crea poi un oggetto che contenga le informazioni utili alla creazione dell'attore (configurazione), ovvero classe, argomenti del costruttore, mailbox, implementazione del dispatcher; infine l'`Actor System` crea una nuova istanza dell'attore utilizzando la configurazione data. Quando una nuova istanza viene creata dall' `Actor System`, questo ritorna una `Actor Reference` a quella istanza, ovvero un oggetto che permette di inviare messaggi a quello specifico attore, indipendentemente da dove è allocato. Al nuovo attore vengono associati un `path` (i.e. una locazione che può contenere un attore attivo) e una mailbox.

Per creare una nuova `actor class` in Scala Akka è necessario estendere il trait `Actor` e implementarne il metodo astratto `receive`. Il metodo `receive` deve definire una serie di case statements che definiscono quali messaggi l'attore può gestire e come processarli. Questo è implementato attraverso una funzione parziale di tipo `PartialFunction[Any, Unit]`, ovvero una funzione unaria definita su valori di tipo `Any` (tipo qualsiasi, la classe `Any` è superclasse di tutte le classi) e ha come tipo di ritorno `Unit` (non restituisce nulla). L'esempio seguente è tratto dalla documentazione e mostra la definizione di una classe attore con la `receive` definita su tutti i messaggi:

```
import akka.actor.Actor
import akka.actor.Props
import akka.event.Logging

class MyActor extends Actor {
  val log = Logging(context.system, this)
  def receive = {
    case "test" => log.info("received test")
    case _      => log.info("received unknown message")
  }
}
```

Il valore della `receive` è una funzione parziale che identifica il comportamento iniziale dell'attore. Vedremo che questo comportamento può essere modificato in seguito alla ricezione di un messaggio attraverso il costrutto `become`.

L'azione di creare un'istanza di un attore `MyActor` e mandarla in esecuzione è eseguita dall'`Actor System`. Per poter creare un'istanza il sistema di attori richiede una configurazione, ovvero un oggetto che contiene informazioni sull'attore tra cui la classe e la mailbox. Tale oggetto appartiene alla classe `Props`. Una volta creata un'istanza di `Props`, questa viene passata all'`Actor System` che crea la nuova istanza dell'attore attraverso il metodo `actorOf`. L'esempio seguente, anch'esso tratto dalla documentazione, mostra la creazione di un si-

stema di attori e di un attore della classe `MyActor`.

```
import akka.actor.ActorSystem

val system = ActorSystem("mySystem")
val myActor = system.actorOf(Props[MyActor], "myactor2")
```

Non è necessario definire una classe per l'attore che si vuole creare, si può anche utilizzare una classe anonima in questo modo:

```
system.actorOf(Props(new Actor {
      def receive = {
        case msg => ..
      }
    }))
```

Questo sarà il metodo utilizzato all'interno di questo lavoro.

Creare un attore dal corpo di un attore

Un attore può anche essere creato da un altro attore attraverso il metodo `actorOf` del contesto (vedi paragrafo successivo). In questo modo si crea un attore figlio, realizzando quindi una gerarchia di attori. Anche in questo caso si può utilizzare una classe anonima che contenga il comportamento dell'attore, in questo modo:

```
context.actorOf(Props(new Actor {
      def receive = {
        case msg => ..
      }
    }))
```

La documentazione Akka raccomanda però di fare attenzione a non chiamare metodi sull'attore padre dall'interno della classe anonima che definisce l'attore figlio. Questo romperebbe l'incapsulamento dell'attore e rischierebbe di provocare bugs di sincronizzazione e race conditions.

Dal corpo di un attore è possibile invocare il metodo `actorOf` anche sul sistema di attori (`system.actorOf(Props(new ...))`). Questo non crea un attore figlio ma un nuovo attore top-level.

Actor API

Il trait `Actor` definisce solo, come abbiamo detto, il metodo astratto `receive` che implementa il comportamento iniziale dell'attore. Offre però anche [3]

- il riferimento `self` all'`ActoRef` dell'attore
- un riferimento `sender` all'`ActorRef` dell'attore che ha inviato l'ultimo messaggio gestito
- un contesto con le informazioni contestuali per l'attore e il messaggio corrente. In particolare il contesto contiene, tra le altre cose:
  - il metodo `actorOf` per creare attori figli
  - il sistema a cui l'attore appartiene
  - il riferimento al padre
  - il riferimento ai figli
  - lifecycle monitoring

  Abbiamo già parlato della creazione di un attore dal corpo di un attore e del sistema di attori. Gli altri concetti verranno chiariti nel seguito di questo capitolo.

1.1.2 Lo scambio di messaggi

Come riportato nell'introduzione a questo capitolo, un modello ad attori si basa su entità che si scambiano messaggi, ignare dell'effettiva locazione le une delle altre. I messaggi nel modello ad attori Akka, devono essere immutabili. L'immutabilità non è imposta da Akka, che però raccomanda questa convenzione come modello di buona programmazione. I messaggi vengono inviati da un attore ad un altro attore attraverso l'operatore ! (`tell`), con una politica fire-and-forget. Questo significa che l'operazione di invio di un messaggio è un'operazione non bloccante, che permette all'attore che ha inviato il messaggio di continuare a processare il suo corpo, senza aspettare una risposta dall'attore a cui ha inviato il messaggio. Tuttavia esiste anche l'operatore ? (`ask`) se un attore che ha inviato una richiesta ha necessità di attendere la risposta. L'espressione di invio dei messaggi è nella forma

```
B ! message(args)
```

dove `B` è l'attore a cui il messaggio è destinato e `message(args)` il corpo del messaggio, con parametri `args`.

I messaggi inviati ad un attore vengono depositati nella sua mailbox e passati poi all'attore da un dispatcher. Analizziamo nel dettaglio questi due nuovi elementi.

Una mailbox è una parte di memoria dove vengono allocati i messaggi destinati ad un attore. Normalmente ogni attore ha la propria mailbox. Questa

parte di memoria è necessaria in quanto un attore processa un solo messaggio per volta: tutti i messaggi che arrivano all'attore mentre sta processando vengono accodati nella mailbox in attesa di poter essere gestiti. In Scala Akka ci sono diversi tipi di mailbox, ognuna delle quali estende `MailboxType`. Se non viene specificato diversamente viene utilizzata quella di default. La mailbox di default è una `UnboundedMailbox`, che contiene potenzialmente un numero illimitato di messaggi e viene gestita come una coda FIFO. Questa è una fondamentale differenza con la mailbox utilizzata dagli attori Scala deprecati: non c'è più ricerca all'interno della mailbox di un messaggio che possa essere gestito, ma ogni messaggio viene gestito, o dal comportamento definito per l'attore o, come vedremo in seguito, come `Unhandled Message`. La gestione della mailbox come coda FIFO garantisce anche che i messaggi inviati dall'attore A all'attore B, vengano ricevuti da B nell'esatto ordine in cui A li ha inviati.

Il dispatcher può essere definito il motore del sistema. Si tratta di un componente che decide quando un attore può processare un messaggio e gli fornisce le risorse computazionali per farlo. I messaggi arrivano nella mailbox attraverso un'operazione di accodamento non bloccante. Questo significa che sia il richiedente che l'attore di destinazione, continuano a processare il loro corpo dopo l'invio del messaggio [11]. Quando un nuovo messaggio viene accodato alla mailbox il dispatcher viene risvegliato e si accorge che c'è un nuovo messaggio da processare per l'attore. Se l'attore di destinazione sta già processando allora, in accordo con la politica del modello ad attori per cui un attore fa una sola cosa alla volta, il dispatcher non fa nulla. Quando invece l'attore non sta processando nulla, o ha finito di processare il precedente messaggio, allora viene schedulato per l'esecuzione su un thread e il dispatcher gli consegna il messaggio da processare. È importante notare che, mentre un attore processa un messaggio, è completamente ignaro di quello che succede alla sua mailbox. Quando l'attore finisce di processare il messaggio corrente, viene inviato un segnale al dispatcher che può consegnargli il prossimo messaggio.

Unhandled messages Il comportamento definito per un attore attraverso la `receive` riesce a gestire determinati messaggi, identificati dai case statements. La `receive` può non essere esaustiva, ovvero non è richiesto che definisca un codice da eseguire per ogni tipo di messaggio ricevuto. La mailbox tuttavia è implementata di default come coda FIFO, ovvero il dispatcher fornisce all'attore tutti i messaggi, uno per volta. Quando il comportamento di un attore non definisce la gestione di un determinato messaggio, questo viene inscatolato in un oggetto di tipo `Unhandled Message` e inviato all'event stream del sistema di attori, che tiene traccia dell'evento nel log, mentre l'attore prosegue nella gestione del prossimo messaggio della sua mailbox.

1.1.3 Cambio di comportamento

Abbiamo detto che, quando si crea un attore, si definisce il metodo `receive` che definisce il suo comportamento iniziale, i.e. il set di messaggi che l'attore è in grado di gestire. Tuttavia nel corso dell'esecuzione può essere necessario cambiare il modo in cui i messaggi vengono gestiti da un attore, o i messaggi stessi che possono essere gestiti. Per cambiare il comportamento di un attore si usa il metodo `become` sull'oggetto `context` dell'attore. Il metodo `become` richiede come parametro un comportamento di tipo `Receive`, ovvero una funzione parziale del tipo `PartialFunction[Any, Unit])`. I cambi di comportamento sono particolarmente utili per implementare le FSM (Finite State Machines, Macchine a Stati Finiti), ad esempio per modellare il classico esempio dei filosofi a cena.

Per completezza citiamo anche un altro modo d'uso del metodo `become`, che non rimpiazza il comportamento corrente, ma aggiunge alcuni case statement in cima allo stack del comportamento. Tali aggiunte possono essere poi tolte con `unbecome`. Riportiamo un esempio di questo secondo uso del metodo become tratto dalla documentazione:

```
  def receive = {
    case Swap =>
      log.info("Hi")
      become({
        case Swap =>
          log.info("Ho")
          unbecome() // resets the latest 'become'
      }, discardOld = false) // push on top instead of replace
  }
}
```

In questo esempio il metodo `become` ha due parametri: la funzione parziale che definisce il nuovo comportamento e un parametro `discardOld` che, se settato a `true` rimpiazza il comportamento corrente con quello nuovo, se settato a `false`, come in questo caso, mantiene il comportamento corrente, inserendo i nuovi case statement all'inizio del comportamento. Nell'esempio il nuovo case statement gestisce messaggi `Swap` stampando una riga di log e poi chiamando il metodo `unbecome`. Il metodo `unbecome` ripristina il comportamento precedente dell'attore, ovvero il comportamento che aveva prima che gli fosse aggiunto il nuovo case statement in cima.

1.1.4 Arrestare gli attori

Ci sono diversi modi per arrestare un attore. Il più comune è chiamare il metodo `stop` a cui viene passato il riferimento all'attore. Il metodo `stop` può essere invocato su un'istanza di `ActorSystem` oppure sull'oggetto `context` all'interno del corpo dell'attore. In genere il contesto è utilizzato per arrestare l'attore stesso o i suoi figli, mentre il sistema per arrestare gli attori top-level. Il

seguente esempio (tratto dalla documentazione), mostra un attore con un figlio `child` che gestisce due tipi di messaggi: in seguito alla ricezione di un messaggio interrupt-child invoca il metodo stop passando come parametro il riferimento al figlio, provocandone l'arresto; in seguito alla ricezione di un messaggio done invoca il metodo stop passando come parametro il riferimento a se stesso.

```
1   class MyActor extends Actor {
2
3     val child: ActorRef = ...
4
5     def receive = {
6       case "interrupt-child" =>
7         context stop child
8
9       case "done" =>
10        context stop self
11    }
12
13  }
```

L'invocazione del metodo `stop` permette all'attore che deve essere arrestato di terminare di processare il messaggio corrente prima di iniziare il suo processo di terminazione. Il processo di terminazione di un attore avviene in due fasi:

1. l'attore smette di processare i messaggi contenuti nella mailbox e invia un segnale di stop a tutti i suoi figli. In questa fase alla mailbox dell'attore possono ancora giungere messaggi, che però non verranno mai gestiti.
2. quando l'attore ha ricevuto le notifiche di terminazione da tutti i figli termina, ovvero invoca il proprio metodo `postStop`, svuota la mailbox (che possibilmente contiene ancora messaggi) e avverte il genitore della sua terminazione.

DeadLetters Dopo che un attore è stato arrestato entra in uno stato chiamato `Terminated`. Tuttavia un attore che ne ha ancora il riferimento può inviare messaggi all'attore se è ignaro della sua terminazione. Tutti i messaggi che vengono inviati ad un attore che è in uno stato `Terminated` vengono incapsulati in un oggetto `DeadLetter` e inviati all'`EventStream`. Il risultato è una riga di log di questo tipo:

```
[LifecycleActorSystem-akka.actor.default-dispatcher-4] INFO
akka.actor.RepointableActorRef - Message [java.lang.String]
from Actor[akka://LifecycleActorSystem/deadLetters] to
Actor[akka://LifecycleActorSystem/user/lifecycleActor#-782937925]
was not delivered. [1] dead letters encountered. This logging
can be turned off or adjusted with configuration settings
'akka.log-dead-letters' and 'akka.log-dead-letters-during-shutdown'.
```

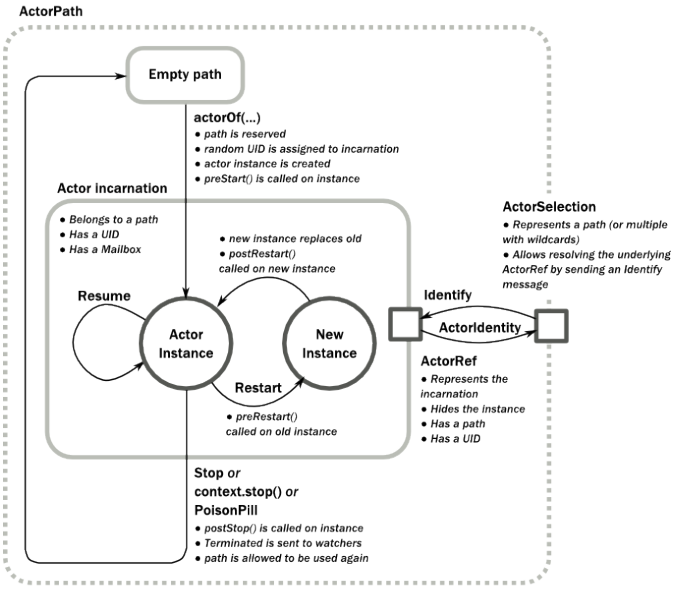


Figura 1.1: Ciclo di vita di un attore

L'invio di un messaggio ad un attore terminato non provoca interruzioni né eccezioni, in accordo con i principi non bloccanti del modello ad attori.

PoisonPill Un altro metodo per arrestare un attore è inviargli un messaggio di tipo `Poison Pill`. Il messaggio viene accodato in mailbox come un messaggio ordinario. L'attore a cui è stato inviato processa normalmente la mailbox e, quando arriva a processare il messaggio `PoisonPill`, avvia la sua terminazione come descritto sopra. L'effetto è che, a differenza del metodo `stop`, l'attore può processare tutti i messaggi arrivati in mailbox prima di `Poison Pill`.

1.2 Argomenti non trattati

Scopo di questa tesi è, come detto, modellare formalmente il comportamento degli attori Scala Akka. Tuttavia gli attori Scala Akka sono molto complessi quindi abbiamo dovuto effettuare una scelta su quali aspetti modellare, accantonandone per il momento altri. Vale comunque la pena citare rapidamente, per completezza, anche gli aspetti di Akka che non vengono toccati da questa tesi.

Il ciclo di vita di un attore

L'immagine 1.1, tratta dalla documentazione Scala Akka, mostra il ciclo di vita di un attore. Quando un attore viene creato, l'istanza dell'attore viene assegnata ad un `path`. Un attore che viene attivato ha, come detto, una mailbox

associata e un comportamento associato. Prima che l'attore inizi a processare messaggi viene chiamato il metodo `preStart`. Questo metodo è utile per inizializzare l'istanza dell'attore.

Durante l'esecuzione si può verificare un'eccezione che richiede il riavvio dell'attore. Quando ciò si verifica viene chiamato il metodo `preRestart` e vengono arrestati tutti i figli dell'attore da riavviare. Successivamente l'oggetto `Props` che era stato usato per creare l'attore viene utilizzato per creare un nuovo oggetto attore, su cui viene chiamato il metodo `postRestart`. Al nuovo attore viene associata la stessa mailbox del precedente e può continuare a processare i messaggi che erano nella mailbox prima del riavvio. Quando un attore viene riavviato, il suo riferimento e il suo `path` rimangono gli stessi. Pertanto un attore che possiede il riferimento di un attore riavviato può continuare a inviare messaggi utilizzando lo stesso riferimento, sebbene questi vengano gestiti da due oggetti attore diversi.

Quando un attore viene arrestato (vedi sezione 1.1.4), viene invocato il metodo `postStop` dell'attore e il `path` viene rilasciato.

Notiamo che l'attore può venire riavviato n volte in modo del tutto trasparente al resto del sistema, che continua a vedere lo stesso riferimento.

Il Supervision System

In Akka ogni attore agisce come supervisore per i propri figli. Quando un figlio fallisce invia un messaggio al padre che decide che cosa fare per il figlio. Questa politica è chiamata strategia di supervisione. Quando un figlio termina il genitore può decidere di:

- riavviarlo (inviandogli un messaggio `Restart`)
- fargli riprendere il proprio lavoro senza riavviarsi (`Resume`)
- arrestarlo (`Stop`)
- fallire egli stesso con la stessa eccezione (`Escalate`)

.

Quando non viene definita una strategia di supervisione per l'attore padre, viene utilizzata quella di default. La strategia di default prevede, a seconda del tipo di eccezione incontrata dal figlio, un diverso comportamento:

- `ActorInitializationException` e `ActorKilledException` arrestano l'attore figlio che è fallito
- `Exception` riavvia l'attore figlio fallito
- altri tipi di `Throwable` risalgono al padre

Una strategia di supervisione può essere di tipo One-For-One oppure All-For-One. La strategia One-For-One definisce che cosa fare per il singolo attore che ha incontrato un'eccezione (è anche la strategia di default). Se invece in un

set di figli questi dipendono così strettamente gli uni dagli altri che il fallimento di uno influenza tutti gli altri, si può usare una strategia All-For-One, per cui la strategia indicata per l'attore fallito viene utilizzata anche su tutti i suoi fratelli.

Vi sono tre supervisori top-level:

- \user o Guardian Actor : gli attori creati con `system.actorOf()` sono figli di questo attore.
- \system o System Guardian: introdotto per realizzare uno shut-down ordinato quando il log rimane attivo mentre i normali attori terminano.
- \ o Root Guardian: supervisiona gli attori \ `user` e \ `system`.

1.3 Akka Typed

Akka Typed è un modulo ad ora sperimentale disponibile dalla versione 2.4 di Akka. Akka Typed nasce per poter garantire staticamente che, quando un messaggio viene inviato ad un attore, il tipo di quel messaggio è uno di quelli che l'attore è in grado di gestire. Attualmente è possibile definire un comportamento per un attore e in seguito inviare all'attore un messaggio che questo comportamento non è in grado di gestire. Quando ciò si verifica tale messaggio, come visto alla sezione 1.1.2, viene inviato al sistema di attori e finisce in una sorta di limbo a meno che non venga definito esplicitamente un comportamento in tali situazioni. Con Akka Typed si cerca di prevenire questo, creando un sistema che garantisca a tempo di compilazione che non vi saranno, durante l'esecuzione, messaggi non gestiti.

Come abbiamo visto nella sezione 1.1.1, nell'attuale versione degli attori Akka un attore viene creato estendendo il trait `Actor` e implementando il suo metodo `receive`. In Akka Typed, per creare un attore è sufficiente definirne il comportamento, che contiene già tutto quello che c'è da sapere su un determinato attore. Ovviamente tale comportamento è tipato. Il tipo del comportamento identifica il tipo dei messaggi che quel comportamento è in grado di gestire.

Vediamo un esempio tratto dalla documentazione [3]

```
object HelloWorld {
  final case class Greet(whom: String, replyTo: ActorRef[Greeted])
  final case class Greeted(whom: String)

  val greeter: Behavior[Greet] = Static[Greet] { msg =>
    println(s"Hello ${msg.whom}!")
    msg.replyTo ! Greeted(msg.whom)
  }
}
```

L'esempio mostra la definizione di un comportamento (i.e. il valore `greeter`) per un attore. `Static` è un costruttore per comportamenti di cui discuteremo

più avanti. Per ora notiamo che al costruttore è associato il tipo `Greet`. Questo significa che il comportamento definito è in grado di gestire messaggi che abbiano tipo `Greet`. Avendo `msg` tipo `Greet`, `msg.whom` e `msg.replyTo` sono espressioni ben tipate. Notiamo anche che un comportamento non ha il riferimento all'attore che ha inviato il messaggio, ma questo deve essere passato come parametro del messaggio. Il comportamento definito riceve quindi messaggi di tipo `Greet` e risponde con un messaggio di tipo `Greeted`.

Un attore viene creato a partire dal suo comportamento in questo modo:

```
val system: ActorSystem[Greet] = ActorSystem("hello", Props(greeter))
```

questo crea e avvia un Actor System anch'esso tipato, a cui viene passato il comportamento definito. In questo modo viene avviato un top-level actor con il comportamento definito da `greeter`.

Ci sono diversi costruttori per definire un comportamento [7], solo alcuni dei quali permettono all'attore di cambiare comportamento in seguito alla ricezione di un determinato tipo di messaggio. Il comportamento che verrà assunto dall'attore alla ricezione del messaggio successivo ha il vincolo di essere dello stesso tipo del comportamento attuale. I costruttori per i comportamenti sono:

- Total: permette di cambiare comportamento scegliendo tra i predefiniti o creandone uno personalizzato e richiede di gestire tutti i messaggi del tipo definito, ma ignora i segnali di sistema (notifiche generate da eventi di sistema).

  ```
  Total {
       case Eat(food) =>
         println(s"Tasty $food!")
         Stopped
       case Wait() =>
         Same
  }
  ```

  in questo esempio viene costruito un comportamento che risponde a messaggi di tipo `Eat` e `Wait`. Quando riceve un messaggio di tipo `Eat` l'attore termina (valore `Stopped`), se invece riceve un messaggio di tipo `Wait` mantiene lo stesso comportamento (valore `Same`).

- Tap: permette di eseguire alcune azioni prima di ricevere messaggi con il comportamento corrente. Corrisponde alla creazione di un attore con corpo non vuoto negli attuali attori Akka.

- Partial: permette di costruire un nuovo comportamento attraverso una funzione parziale. Tutti i messaggi che non vengono gestiti dal comportamento vengono considerati `Unhandled`

- Static: definisce un comportamento che non cambia (ritorna di default il valore `Same`).
- Full: funziona come `Total` ma è in grado di gestire anche i segnali.

Akka Typed è il risultato di molti anni di ricerca e precedenti tentativi (tra cui i `Typed Channel` delle versioni 2.2.x). Il modulo è tuttora in fase sperimentale e si stima che l'implementazione attuale degli attori non sarà deprecata entro breve.

Capitolo 2

Il linguaggio formale A

2.1 Sintassi

Assumiamo un insieme infinito di nomi di attori $Act = \{a, b, c...\}$ e un insieme infinito di variabili $Var = \{x, y, z...\}$. Consideriamo u, w identificatori che possono essere nomi o variabili. La lettera m identifica l'etichetta (label) dei messaggi. $\tilde{u}, \tilde{w}$ sono tuple di nomi/variabili distinti.

Espressioni	$e ::= \underline{0} \mid u!m(\tilde{u}); e \mid val\ a = Actor\{\beta; e\}; e \mid become(\beta); e$
Comportamenti	$\beta ::= \underline{0} \mid \{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I}$

L'espressione $u!m(\tilde{u}); e$ invia all'attore u il messaggio m con la tupla di parametri attuali $\tilde{u}$ e poi prosegue come e. Il messaggio viene accodato nella mailbox dell'attore u, come viene specificato dalla semantica operazionale.

Un nuovo attore può essere creato dinamicamente con l'espressione $val\ a = Actor\{\beta; e_1\}; e_2$. La definizione genera il nuovo nome legato a il cui scope è sia il corpo e_1 dell'attore, sia la continuazione e_2. Il corpo dell'attore deve essere costituito dal suo comportamento β (i.e. implementazione del suo metodo receive) seguito da un'espressione e_1 possibilmente nulla. Il comportamento di un attore è la serie di case statements che definiscono quali messaggi possono essere gestiti dall'attore. I case statements sono nella forma $m(\tilde{x}) \Rightarrow e$ dove il pattern $m(\tilde{x})$ è costituito dall'etichetta m e da una tupla di variabili legate che compaiono nel corpo e. Perché il comportamento sia ben formato occorre che nei diversi pattern $\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I}$ le labels m_i siano distinte. Se $I = \emptyset$ si ha il comportamento nullo $\underline{0}$, che corrisponde al costrutto `PartialFunction.empty` di Scala Akka.

L'espressione $become(\beta); e$ permette all'attore di assumere un nuovo comportamento β e poi continuare in accordo con e.

Un programma sorgente è un'espressione chiusa, cioè senza variabili libere né nomi liberi (vedi def. 1, 2). Lo stato runtime di un programma è rappresentato

da una configurazione:

Configurazioni

$F ::= \underline{0} \mid [a \mapsto M](\beta)a\{e\} \mid F|F \mid (\nu a)F \mid d$
$d ::= \underline{0} \mid val\ a = Actor\{\beta; e\}; d$
$M ::= \emptyset \mid m(\tilde{a}) \cdot M$

$(\nu a)F$ è una configurazione dove a è un nome privato di attore che ha come scope F. Una configurazione è una composizione parallela di un certo numero di attori attivi e un'espressione d che contiene la sequenza residua di definizioni di attori top-level. Da notare che la sintassi, così com'è definita, non permette l'uso di espressioni $u!m(\tilde{u}')$ e $become(\beta)$ al di fuori del corpo di un attore.

Un attore attivo é rappresentato runtime da $[a \mapsto M](\beta)a\{e\}$ dove e è il corpo dell'attore che deve essere ancora eseguito, $[a \mapsto M]$ la mailbox associata all'attore e un contesto che per ora contiene solamente il comportamento attuale β dell'attore, ovvero la funzione parziale che definisce la sua receive.

Le mailbox sono liste ordinate e possibilmente vuote di messaggi ricevuti, nella forma $m_1(\tilde{a}_1) \cdot m_2(\tilde{a}_2) \cdot m_3(\tilde{a}_3)$.... I parametri dei messaggi sono valori, che nel nostro semplice linguaggio sono solo nomi di attori.

Per definire nomi liberi e variabili libere, ci aiutiamo con due funzioni $isVar$ e $isName$ definite induttivamente sulla lunghezza delle tuple in input in questo modo:

$$isVar(u) = \begin{cases} \{u\} & u \text{ è una variabile} \\ \emptyset & u \text{ è un nome} \end{cases}$$
$$isVar(u, \tilde{u}') = isVar(u) \cup isVar(\tilde{u}')$$

$$isName(u) = \begin{cases} \{u\} & u \text{ è un nome} \\ \emptyset & u \text{ è una variabile} \end{cases}$$
$$isName(u, \tilde{u}') = isName(u) \cup isName(\tilde{u}')$$

Definizione 1 (Free Names)**.** *I nomi liberi di una configurazione, di un'espressione o di una lista di messaggi, denotati rispettivamente $fn(F)$, $fn(e)$, $fn(M)$, vengono definiti per induzione sulla struttura di F, e, M come segue:*

Casi base:

- $fn(\underline{0}) = \emptyset$

Casi induttivi:

- $fn(u\,!\,m(\tilde{u}'); e) = isName(u) \cup isName(\tilde{u}') \cup fn(e)$
- $fn(\beta) = fn(\{m_i(\tilde{x_i}) \Rightarrow e_i\}_{i \in I}) = \bigcup_{i \in I} fn(e_i)$
- $fn(become(\beta); e) = fn(\beta) \cup fn(e)$
- $fn(val\ a = Actor\{\beta; e\}; e') = fn(\beta) \cup fn(e) \cup fn(e') \setminus \{a\}$
- $fn([a \mapsto M](\beta)a\{e\}) = \{a\} \cup fn(\beta) \cup fn(e) \cup fn(M)$
- $fn(F_1|F_2) = fn(F_1) \cup fn(F_2)$
- $fn((\nu a)F) = fn(F) \setminus \{a\}$
- $fn(m(\tilde{a}) \cdot M) = \{\tilde{a}\} \cup fn(M)$

Definizione 2 (Free Variables)**.** *le variabili libere di una configurazione e di un'espressione, denotate rispettivamente $fv(F)$ e $fv(e)$, vengono definite per induzione sulla struttura di F, e come segue:*

Casi base:

- $fv(\underline{0}) = \emptyset$

Casi induttivi:

- $fv(u\,!\,m(\tilde{u}'); e) = isVar(u) \cup isVar(\tilde{u}') \cup fv(e)$
- $fv(\beta) = fv(\{m_i(\tilde{x_i}) \Rightarrow e_i\}_{i \in I}) = \bigcup_{i \in I}(fv(e_i) \setminus \{x_i\})$
- $fv(become(\beta); e) = fv(\beta) \cup fv(e)$
- $fv(val\ a = Actor\{\beta; e\}; e') = fv(\beta) \cup fv(e) \cup fv(e')$
- $fv([a \mapsto M](\beta)a\{e\}) = fv(\beta) \cup fv(e)$
- $fv(F_1|F_2) = fv(F_1) \cup fv(F_2)$
- $fv((\nu a)F) = fv(F)$

Definizione 3 (Sostituzione su nomi e variabili)**.** *data una tupla di variabili $\tilde{x} = \{x_1 ... x_n\}$ e una tupla di nomi $\tilde{a} = \{a_1 ... a_n\}$ definiamo:*

- $x_i\{\tilde{a}/\tilde{x}\} = a_i \quad if\ x_i \in \tilde{x}$

- $y\{\tilde{a}/\tilde{x}\} = y \quad if\ y \notin \tilde{x}$
- $b\{\tilde{a}/\tilde{x}\} = b$

Definizione 4 (Sostituzione)**.** *data una tupla di variabili $\tilde{x} = \{x_1...x_n\}$ e una tupla di nomi $\tilde{a} = \{a_1...a_n\}$ definiamo la sostituzione $e\{\tilde{a}/\tilde{x}\} = e\{a_1/x_1, ..., a_n/x_n\}$ per induzione sulla struttura di e:*

Casi base:

- $\underline{0}\{\tilde{a}/\tilde{x}\} = \underline{0}$

Casi induttivi:

- $(u\ !\ m(\tilde{u}'); e)\{\tilde{a}/\tilde{x}\} = u\{\tilde{a}/\tilde{x}\}\ !\ m(\tilde{u}'\{\tilde{a}/\tilde{x}\}); e\{\tilde{a}/\tilde{x}\}$
- $(\beta)\{\tilde{a}/\tilde{x}\} = (\{m_i(\tilde{y_i}) \Rightarrow e_i\}_{i\in I})\{\tilde{a}/\tilde{x}\} = \{m_i(\tilde{y_i}) \Rightarrow e_i\{\tilde{a}/\tilde{x}\}\}_{i\in I}$ *con* $\{y_i\} \cap \{x_i\} = \emptyset$
- $(become(\beta); e)\{\tilde{a}/\tilde{x}\} = become(\beta\{\tilde{a}/\tilde{x}\}); e\{\tilde{a}/\tilde{x}\}$
- $(val\ a = Actor\{\beta; e\}; e')\{\tilde{a}/\tilde{x}\} = (val\ a = actor\{\beta\{\tilde{a}/\tilde{x}\}; e\{\tilde{a}/\tilde{x}\}\}; e'\{\tilde{a}/\tilde{x}\})$ *con* $a \notin \tilde{a}$

Definizione 5 (Structural Congruence)**.** *La congruenza strutturale (denotatata con $\equiv$) è la minima relazione tra configurazioni definita dalle seguenti regole e assiomi:*

$$\frac{F \equiv F'}{\begin{array}{l} F_1 \mid F \equiv F_1 \mid F' \\ F \mid F_1 \equiv F' \mid F_1 \\ (\nu a)F \equiv (\nu a)F' \end{array}} \qquad \text{(Riflessiva)}\ \frac{}{F \equiv F} \qquad \text{(Simmetrica)}\ \frac{F \equiv F'}{F' \equiv F} \qquad \text{(Transitiva)}\ \frac{F \equiv F_1 \quad F_1 \equiv F_2}{F \equiv F_2}$$

(1) $(\nu a)(\nu b)F \equiv (\nu b)(\nu a)F$
(2) $(\nu a)(F|F') \equiv F|(\nu a)F'\ a \notin fn(F)$
(3) $F|\underline{0} \equiv F$
(4) $F|F' \equiv F'|F$
(5) $(F_1|F_2)|F_3 \equiv F_1|(F_2|F_3)$
(6) $(\nu a)\underline{0} \equiv \underline{0}$

2.2 Semantica operazionale

La semantica operazionale è la relazione di riduzione $F \longrightarrow F'$ definita dal seguente sistema di regole:

$$\text{(PAR)}\ \frac{F_1 \longrightarrow F_1'}{F_1|F_2 \longrightarrow F_1'|F_2} \qquad \text{(RES)}\ \frac{F \longrightarrow F'}{(\nu a)F \longrightarrow (\nu a)F'} \qquad \text{(STRUCT)}\ \frac{F \equiv F' \quad F' \to F'' \quad F'' \equiv F'''}{F \longrightarrow F'''}$$

(TOP-SPAWN)

$$\frac{}{val\ a = Actor\{\beta; e\}; d \longrightarrow (\nu a)([a \mapsto \emptyset](\beta)a\{e\} \mid d)}$$

(SPAWN)

$$\frac{a \notin (fn(\beta) \cup fn(M))}{[b \mapsto M](\beta)b\{val\ a = Actor\{\beta'; e\}\ ; e'\} \longrightarrow (\nu a)([b \mapsto M](\beta)b\{e'\} \mid [a \mapsto \emptyset](\beta')a\{e\})}$$

(BECOME)

$$\frac{}{[a \mapsto M](\beta)a\{become(\beta');\ e\} \longrightarrow [a \mapsto M](\beta')a\{e\}}$$

(SEND)

$$\frac{}{[a \mapsto M](\beta)a\{e\} \mid [b \mapsto M'](\beta')b\{a!m(\tilde{c}); e'\} \longrightarrow [a \mapsto M \cdot m(\tilde{c})](\beta)a\{e\} \mid [b \mapsto M'](\beta')b\{e'\}}$$

(RECEIVE)

$$\frac{j \in I}{[a \mapsto m_j(\tilde{c}) \cdot M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I})a\{\underline{0}\} \longrightarrow [a \mapsto M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I})a\{e_j\{\tilde{c}/\tilde{x_j}\}\}}$$

(JUNK)

$$\frac{j \notin I}{[a \mapsto m_j(\tilde{c}) \cdot M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I})a\{\underline{0}\} \longrightarrow [a \mapsto M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I})a\{\underline{0}\}}$$

(TOP-SPAWN) e (SPAWN) creano e avviano un attore rispettivamente top-level e dal corpo di un altro attore. In entrambi i casi alla configurazione viene aggiunto un thread per eseguire il corpo del nuovo attore, una mailbox vuota associata e un contesto associato, che contiene il comportamento iniziale β

dell'attore. Tale comportamento può cambiare nel corso dell'esecuzione (regola BECOME). Le regole (SEND), (RECEIVE) e (JUNK) formalizzano lo scambio di messaggi tra attori.

La regola (SEND) ripone un messaggio nella mailbox dell'attore di destinazione. La mailbox viene trattata come lista FIFO: il primo messaggio che viene ricevuto viene processato. Questo significa che messaggi inviati dallo stesso attore verranno processati uno dopo l'altro nello stesso ordine in cui sono stati inviati. Un messaggio presente in mailbox viene ricevuto quando l'attore di destinazione ha finito di processare il messaggio precedente, ovvero ha corpo vuoto, se il comportamento prevede la gestione di un messaggio di quel tipo. Se quest'ultima cosa non accade, il messaggio viene semplicemente tolto dalla mailbox (regola JUNK).

Le forward references per i riferimenti Le regole date per la generazione degli attori prevedono che un nome di attore non possa essere utilizzato prima di essere dichiarato. Ad esempio, il programma $val\ a = Actor\{\beta; e\}; val\ b = Actor\{\beta'; e'\}$ evolve in:

$$\longrightarrow^{TOP-SPAWN} (\nu a)([a \mapsto \emptyset](\beta \uparrow \emptyset \downarrow \emptyset)a\{e\} \mid val\ b = Actor\{\beta'; e'\})$$

$$\longrightarrow^{TOP-SPAWN} (\nu a)([a \mapsto \emptyset](\beta \uparrow \emptyset \downarrow \emptyset)a\{e\} \mid (\nu b)([b \mapsto \emptyset](\beta' \uparrow \emptyset \downarrow \emptyset)b\{e'\}))$$

ovvero il corpo e' di b può usare il nome a, mentre il corpo e si a non può utilizzare il nome b che non è ancora stato dichiarato.
Tuttavia in Scala il seguente programma, in cui l'attore a manda un messaggio a b prima che questo venga dichiarato, compila correttamente:

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5
6  object Scope extends App{
7
8    val system = ActorSystem()
9    val a : ActorRef = system.actorOf(Props(new Actor{
10     def receive = {case x => println("a: received " + x)}
11     b ! "msg from a"
12   }))
13
14    val b : ActorRef = system.actorOf(Props(new Actor{
15     def receive = {case x => println("b: received " + x)}
16     a ! "msg from b"
17    }))
```

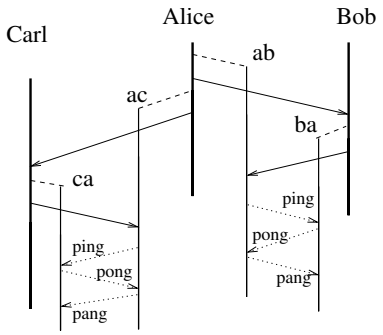


Figura 2.1: Alice Bob e Carl ping pong

18 }

In Scala infatti, quando un oggetto viene inizializzato, i riferimenti vengono inizialmente settati al valore di default (`null` in questo caso) e in seguito il costruttore (i.e. il corpo dell'oggetto) viene eseguito. Questo permette le forward references (riferimenti in avanti). Runtime si possono verificare due situazioni: o l'attore b viene inizializzato prima che a ne utilizzi il nome e quindi il programma esegue correttamente, altrimenti quando a cercherà di fare qualche operazione utilizzando il nome di b troverà un riferimento a null che farà references, ma questo non causa comunque perdita di espressività: l'esempio di cui sopra può essere implementato facendo in modo che l'attore b insegni ad a il proprio nome inviandoglielo come parametro di un messaggio.

2.3 Esempi

2.3.1 Alice Bob e Carl ping pong

Consideriamo un programma in cui due attori effettuano una three-way handshake: un attore a invia a un attore b un messaggio *ping*, si prepara a ricevere un *pong* e poi risponde con un *pang*. Qui l'esempio coinvolge due handshake paralleli: uno tra un attore *Alice* e un attore *Bob* e uno tra *Alice* e un attore *Carl*. Nel prossimo capitolo l'esempio verrà esteso gestendo anche la terminazione degli attori.

Il programma scritto in linguaggio A è il seguente:

$$
\begin{aligned}
&val\ Bob = Actor\{\beta_B; \underline{0}\};\\
&val\ Carl = Actor\{\beta_C; \underline{0}\};\\
&val\ Alice = Actor\{\beta_A;\ val\ ab = Actor\{\beta_{ab}; \underline{0}\}; Bob!new(ab);\\
&\qquad\qquad\qquad\qquad val\ ac = Actor\{\beta_{ac}; \underline{0}\}; Carl!new(ac)\}\}
\end{aligned}
$$

dove
$\beta_A = \underline{0}$
$\beta_B = \{new(z) \Rightarrow val\ ba = Actor\{\beta_{ba}; \underline{0}\}; z!dest(ba)\}$
$\beta_{ab} = \{dest(y) \Rightarrow y!ping(); become(\beta'_{ab})\}$
$\beta_{ba} = \{ping() \Rightarrow z!pong(); become(\beta'_{ba})\}$
$\beta'_{ab} = \{pong() \Rightarrow y!pang()\}$
$\beta'_{ba} = \{pang() \Rightarrow \underline{0}\}$

β_C, β_{ac}, β_{ca}, β'_{ac} e β'_{ca}, usati da $Carl$, si ottengono dai corrispettivi di Bob.

Chiamiamo P il programma sorgente appena descritto e analizziamone l'evoluzione aiutandoci con le seguenti abbreviazioni:

$e_A ::= val\ ab = Actor\{\beta_{ab}\}; Bob\ !\ new(ab);\ val\ ac = Actor\{\beta_{ac}\}; Carl\ !new(ac)$
$e' ::= val\ Carl = Actor\{\beta_C; \underline{0}\};\ val\ Alice = Actor\{\beta_A; e_A\}$
$e'' ::= val\ Alice = Actor\{\beta_A; e_A\}$

$$P \longrightarrow^{TOP-SPAWN} (\nu Bob)([Bob \mapsto \emptyset](\beta_B)Bob\{\underline{0}\}\ |e')$$

$$\longrightarrow^{TOP-SPAWN} (\nu Bob)([Bob \mapsto \emptyset](\beta_B)Bob\{\underline{0}\} \mid (\nu Carl)([Carl \mapsto \emptyset](\beta_C)Carl\{\underline{0}\}\ |e''))$$

$$\longrightarrow^{TOP-SPAWN} (\nu Bob)([B \mapsto \emptyset](\beta_B)Bob\{\underline{0}\} \mid (\nu Carl)([C \mapsto \emptyset](\beta_C)Carl\{\underline{0}\} \mid (\nu Alice)([Alice \mapsto \emptyset](\beta_A)Alice\{e_A\})))$$

Vediamo l'esecuzione di questa composizione parallela di attori. Per la regola (PAR) una configurazione di attori in parallelo evolve in una configurazione di attori in parallelo in cui il primo ha fatto un passo di computazione. Dalle regole di congruenza abbiamo però che $P_1|P_2$ è congruente con $P_2|P_1$. Questo significa che in ogni momento si può scegliere a quale attore della configurazione far fare un passo di computazione.

Bob e $Carl$ hanno corpo e mailbox vuote, quindi l'unica regola che si può applicare è quella che fa eseguire il corpo di $Alice$. Con quattro passi di riduzione, corrispondenti alle regole (SPAWN), (SEND), (SPAWN), (SEND), si arriva a:

$(\nu ab)(\nu Bob)(\ [Bob \mapsto new(ab)](\beta_B)Bob\{\underline{0}\} \mid$
$(\nu ac)(\nu Carl)(\ [Carl \mapsto new(ac)](\beta_C)Carl\{\underline{0}\} \mid$
$(\nu Alice)[Alice \mapsto \emptyset](\beta_A)Alice\{\underline{0}\} \mid [ab \mapsto \emptyset](\beta_{ab})ab\{\underline{0}\} \mid [ac \mapsto \emptyset](\beta_{ac})ac\{\underline{0}\}))$

Ora sia Bob che $Carl$ possono eseguire, quindi sono possibili vari interleaving. Illustriamo solo quello in cui evolve Bob:

$$\longrightarrow^{RECEIVE} (\nu ab)(\nu Bob)(\ [Bob \mapsto \emptyset](\beta_B)Bob\{val\ ba = Actor\{\beta_{ba}\}; ab\ !\ dest(ba)\} \mid ...)$$

$\longrightarrow^{SPAWN}$ $(\nu ba)(\nu ab)(\nu Bob)(\ [Bob \mapsto \emptyset](\beta_B)Bob\{ab\ !\ dest(ba)\} \mid ... \mid [ba \mapsto \emptyset](\beta_{ba})ba\{\underline{0}\})$

$\longrightarrow^{SEND}$ $(\nu ba)(\nu ab)(\nu Bob)(\ [Bob \mapsto \emptyset](\beta_B)Bob\{\underline{0}\} \mid ... \mid [ab \mapsto dest(ba)](\beta_{ab})ab\{\underline{0}\}$

A questo punto può evolvere $Carl$ oppure ab. Vediamo questo secondo caso. In particolare vediamo tutta l'evoluzione del corpo di ab prima di passare il controllo ad un altro attore.

$\longrightarrow^{RECEIVE}$ $(\nu ba)(\nu ab)(\nu Bob)(\ [Bob \mapsto \emptyset](\beta_B)Bob\{\underline{0}\} \mid ... \mid [ab \mapsto \emptyset](\beta_{ab})ab\{ba\ !\ ping(); become(\beta'_{ab})\})$

$\longrightarrow^{SEND}$ $(\nu ba)(\nu ab)(\nu Bob)(\ ... \mid [ab \mapsto \emptyset](\beta_{ab})ab\{become(\beta'_{ab}\} \mid [ba \mapsto ping()](\beta_{ba})ba\{\underline{0}\})$

$\longrightarrow^{BECOME}$ $(\nu ba)(\nu ab)(\nu Bob)(\ ... \mid [ab \mapsto \emptyset](\beta'_{ab})ab\{\underline{0}\} \mid [ba \mapsto ping()](\beta_{ba})ba\{\underline{0}\})$

Scegliendo di far evolvere prima il corpo di ab non abbiamo perso informazioni su un possibile andamento del programma. Infatti se avessimo scelto dopo la seconda riduzione di ab (che avviene per la regola (SEND)) di far evolvere ba, questo avrebbe ricevuto il messaggio $ping$ e inviato un messaggio $pong$ ad ab che non sarebbe comunque stato ricevuto prima che ab avesse finito di processare il messaggio precedente. Ciò significa che, qualsiasi attore scegliamo di far evolvere, per come è scritto il programma, ad ab non viene mai passato un messaggio $pong$ prima che ab abbia cambiato il proprio comportamento in uno in grado di gestire messaggi di quel tipo.

Facciamo evolvere ora il corpo di ba:

$\longrightarrow^{RECEIVE}$ $(\nu ba)(\nu ab)(\nu Bob)(\ ... \mid [ab \mapsto \emptyset](\beta'_{ab})ab\{\underline{0}\} \mid [ba \mapsto \emptyset](\beta_{ba})ba\{ab\ !\ pong(); become(\beta'_{ba})\})$

$\longrightarrow^{SEND}$ $(\nu ba)(\nu ab)(\nu Bob)(\ ... \mid [ab \mapsto pong()](\beta'_{ab})ab\{\underline{0}\} \mid [ba \mapsto \emptyset](\beta_{ba})ba\{become(\beta'_{ba})\})$

$\longrightarrow^{BECOME}$ $(\nu ba)(\nu ab)(\nu Bob)(\ ... \mid [ab \mapsto pong()](\beta'_{ab})ab\{\underline{0}\} \mid [ba \mapsto \emptyset](\beta'_{ba})ba\{\underline{0}\})$

Anche in questo caso, come nel precedente, abbiamo scelto di completare l'esecuzione del corpo di ba prima di passare il controllo ad un altro attore. Di nuovo, per come è scritto il programma, questo non fa perdere informazioni utili sull'andamento dell'esecuzione. Per lo stesso motivo possiamo vedere come si

conclude la sessione tra Alice e Bob, ovvero lo scambio di messaggi tra ab e ba, ignorando l'esecuzione parallela della sessione tra Alice e Carl, ovvero degli attori ac e bc, che comunque si comporteranno in modo analogo. Nell'esecuzione che segue, per semplicità di lettura, sottolineiamo ad ogni riduzione l'attore che eseguirà il prossimo passo di computazione.

$$\longrightarrow^{RECEIVE} (\nu ba)(\nu ab)(\nu Bob)(\ ... \mid \underline{[ab \mapsto \emptyset](\beta'_{ab})ab\{ba\,!\,pang()\}} \mid [ba \mapsto \emptyset](\beta'_{ba})ba\{\underline{0}\})$$

$$\longrightarrow^{SEND} (\nu ba)(\nu ab)(\nu Bob)(\ ... \mid [ab \mapsto \emptyset](\beta'_{ab})ab\{\underline{0}\} \mid \underline{[ba \mapsto pang()](\beta'_{ba})ba\{\underline{0}\}})$$

$$\longrightarrow^{RECEIVE} (\nu ba)(\nu ab)(\nu Bob)(\ ... \mid [ab \mapsto \emptyset](\beta'_{ab})ab\{\underline{0}\} \mid [ba \mapsto \emptyset](\beta'_{ba})ba\{\underline{0}\})$$

Codice Scala Akka

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5
6  object PingPong extends App{
7
8
9      case class dest(y:ActorRef)
10     case class new_(xy: ActorRef)
11     case class ping()
12     case class pong()
13     case class pang()
14
15     val s = ActorSystem()
16
17     val Bob = s.actorOf(Props(new Actor{
18       def receive = {
19         case new_(z) =>
20           println("B: RECEIVE new(ab)")
21           println("B: SPAWN ba")
22           val ba = context.actorOf(Props(new Actor{
23             def receive = {
24               case ping() =>
25                 println("ba: RECEIVE ping")
26                 println("ba: SEND ab ! pong"); z!pong();
27                 println("ba: BECOME case pang");
```

```
28                context.become({case pang() =>
29                                println("ba: RECEIVE pang")})
30              }
31            }))
32            println("B: SEND ab ! dest(ba)")
33            z!dest(ba)
34        }
35
36      }))
37
38       val Carl = s.actorOf(Props(new Actor{
39        def receive = {
40          case new_(z) =>
41            println("C: RECEIVE new(ac)")
42            println("C: SPAWN ca")
43            val ca = context.actorOf(Props(new Actor{
44              def receive = {
45                case ping() =>
46                  println("ca: RECEIVE ping")
47                  println("ca: SEND ac ! pong"); z!pong();
48                  println("ca: BECOME case pang");
49                  context.become({case pang() =>
50                                  println("ca: RECEIVE pang")})
51              }
52            }))
53            println("C: SEND ac ! dest(ca)")
54            z!dest(ca)
55        }
56
57      }))
58
59      val Alice = s.actorOf(Props(new Actor{
60        println("A: SPAWN ab")
61        val ab = context.actorOf(Props(new Actor{
62          def receive = {
63            case dest(y) =>
64              println("ab: RECEIVE dest(ba)")
65              println("ab: SEND ba ! ping")
66              y ! ping()
67
68              println("ab: BECOME case pong")
69              context.become({case pong() =>
70                println("ab: RECEIVE pong")
71                println("ab: SEND B ! pang"); y ! pang()})
72          }
```

```
73        }))
74
75        println("A: SEND B ! new(ab)")
76        Bob ! new_(ab)
77
78        println("A: SPAWN ac")
79         val ac = context.actorOf(Props(new Actor{
80          def receive = {
81            case dest(y) =>
82              println("ac: RECEIVE dest(ca)")
83              println("ac: SEND ca ! ping")
84              y ! ping()
85              println("ac: BECOME case pong")
86              context.become({case pong() =>
87                println("ac: RECEIVE pong")
88                println("ac: SEND C ! pang"); y ! pang()})
89          }
90        }))
91
92        println("A: SEND C ! new(ac)")
93        Carl ! new_(ac)
94
95        def receive = {
96          case _ =>
97        }
98      }))
99  }
```

Notiamo come il codice Akka utilizzato per l'esempio sia identico al nostro programma P scritto in linguaggio A, a meno delle stampe che sono state inserite per tenere traccia delle operazioni eseguite dagli attori e confrontarle con quelle attese derivate dalla riduzione di P. Nelle stampe, per comodità di lettura, gli attori *Alice*, *Bob* e *Carl* vengono chiamati rispettivamente A, B, C.

La regola (PAR), che ad ogni passo di computazione fa evolvere arbitrariamente uno degli attori in parallelo, rende l'output del programma non deterministico. Abbiamo visto però che la semantica del linguaggio A impone che il corpo di ogni attore evolva sequenzialmente. Per esempio, associando ad ogni azione dell'attore *Alice* una stampa, ci si aspetta la seguente serie di stampe in questo ordine:

A: SPAWN ab
A: SEND TO Bob
A: SPAWN ac
A: SEND TO Carl

possibilmente intervallate da stampe degli altri attori. Eseguiamo alcune volte il programma e vediamo che effettivamente l'output (non deterministico) Scala Akka rivela per ogni attore l'esecuzione sequenziale del suo corpo.

Output 1

```
1  A: SPAWN ab
2  A: SEND B ! new(ab)
3  A: SPAWN ac
4  B: RECEIVE new(ab)
5  B: SPAWN ba
6  A: SEND C ! new(ac)
7  B: SEND ab ! dest(ba)
8  ab: RECEIVE dest(ba)
9  ab: SEND ba ! ping
10 C: RECEIVE new(ac)
11 C: SPAWN ca
12 ab: BECOME case pong
13 C: SEND ac ! dest(ca)
14 ac: RECEIVE dest(ca)
15 ac: SEND ca ! ping
16 ac: BECOME case pong
17 ca: RECEIVE ping
18 ca: SEND ac ! pong
19 ba: RECEIVE ping
20 ba: SEND ab ! pong
21 ca: BECOME case pang
22 ab: RECEIVE pong
23 ba: BECOME case pang
24 ac: RECEIVE pong
25 ac: SEND C ! pang
26 ab: SEND B ! pang
27 ca: RECEIVE pang
28 ba: RECEIVE pang
```

L'output rivela che Alice esegue $SPAWN \hookrightarrow SEND \hookrightarrow SPAWN \hookrightarrow SEND$ in cui $\hookrightarrow$ indica che le operazioni possono essere intervallate da operazioni degli altri attori, che è l'ordine che ci aspettiamo dalle regole di semantica date. Dopo `A: SEND B ! new(ab)` ci aspettiamo che Bob, in accordo con la riduzione data sopra, esegua le operazioni $RECEIVE \hookrightarrow SPAWN \hookrightarrow SEND$ come effettivamente accade. Dalla riduzione del corpo di *ab* ci aspettiamo che in seguito a un `SEND ab ! dest(ba)` ci sia la sequenza $RECEIVE \hookrightarrow SEND \hookrightarrow BECOME$ per l'attore *ab* (e analogamente per *ac*) e che, in seguito ad una `ab: BECOME case pong` e un `SEND ab ! pong`, *ab* esegua la sequenza $RECEIVE \hookrightarrow SEND$. Anche queste due sequenze di stampe sono verificate

dall'output 1. Infine dalla derivazione del corpo di *ba* ci aspettiamo che *ba*, a seguito di un `SEND ba ! ping` esegua $RECEIVE \hookrightarrow SEND \hookrightarrow BECOME$, e che a seguito di `ba: BECOME case pang` e `SEND ba ! pang` si generi una $RECEIVE$. Anche queste ultime riduzioni sono verificate dall'output dato. In modo analogo si verifica anche il comportamento di *Carl* e *ac*. Il programma scritto in Scala Akka genera quindi un output conforme a quello generato dalle regole semantiche presentate. Questi stessi ordinamenti si ritrovano anche nelle seguenti quattro esecuzioni:

Output 2

```
1  A: SPAWN ab
2  A: SEND B ! new(ab)
3  A: SPAWN ac
4  B: RECEIVE new(ab)
5  B: SPAWN ba
6  A: SEND C ! new(ac)
7  C: RECEIVE new(ac)
8  C: SPAWN ca
9  B: SEND ab ! dest(ba)
10 C: SEND ac ! dest(ca)
11 ac: RECEIVE dest(ca)
12 ac: SEND ca ! ping
13 ab: RECEIVE dest(ba)
14 ab: SEND ba ! ping
15 ab: BECOME case pong
16 ca: RECEIVE ping
17 ac: BECOME case pong
18 ba: RECEIVE ping
19 ba: SEND ab ! pong
20 ca: SEND ac ! pong
21 ba: BECOME case pang
22 ab: RECEIVE pong
23 ab: SEND B ! pang
24 ba: RECEIVE pang
25 ca: BECOME case pang
26 ac: RECEIVE pong
27 ac: SEND C ! pang
28 ca: RECEIVE pang
```

Output 3

```
A: SPAWN ab
A: SEND B ! new(ab)
A: SPAWN ac
B: RECEIVE new(ab)
B: SPAWN ba
A: SEND C ! new(ac)
C: RECEIVE new(ac)
C: SPAWN ca
C: SEND ac ! dest(ca)
ac: RECEIVE dest(ca)
ac: SEND ca ! ping
B: SEND ab ! dest(ba)
ab: RECEIVE dest(ba)
ab: SEND ba ! ping
ab: BECOME case pong
ac: BECOME case pong
ba: RECEIVE ping
ba: SEND ab ! pong
ba: BECOME case pang
ab: RECEIVE pong
ab: SEND B ! pang
ba: RECEIVE pang
ca: RECEIVE ping
ca: SEND ac ! pong
ca: BECOME case pang
ac: RECEIVE pong
ac: SEND C ! pang
ca: RECEIVE pang
```

Output 4

```
A: SPAWN ab
A: SEND B ! new(ab)
A: SPAWN ac
B: RECEIVE new(ab)
B: SPAWN ba
B: SEND ab ! dest(ba)
ab: RECEIVE dest(ba)
ab: SEND ba ! ping
A: SEND C ! new(ac)
ab: BECOME case pong
C: RECEIVE new(ac)
```

```
C: SPAWN ca
ba: RECEIVE ping
ba: SEND ab ! pong
C: SEND ac ! dest(ca)
ac: RECEIVE dest(ca)
ac: SEND ca ! ping
ac: BECOME case pong
ba: BECOME case pang
ca: RECEIVE ping
ca: SEND ac ! pong
ab: RECEIVE pong
ac: RECEIVE pong
ca: BECOME case pang
ac: SEND C ! pang
ab: SEND B ! pang
ca: RECEIVE pang
ba: RECEIVE pang
```

Output 5

```
A: SPAWN ab
A: SEND B ! new(ab)
B: RECEIVE new(ab)
B: SPAWN ba
A: SPAWN ac
B: SEND ab ! dest(ba)
ab: RECEIVE dest(ba)
ab: SEND ba ! ping
ab: BECOME case pong
ba: RECEIVE ping
ba: SEND ab ! pong
ba: BECOME case pang
ab: RECEIVE pong
ab: SEND B ! pang
A: SEND C ! new(ac)
C: RECEIVE new(ac)
C: SPAWN ca
ba: RECEIVE pang
C: SEND ac ! dest(ca)
ac: RECEIVE dest(ca)
ac: SEND ca ! ping
ac: BECOME case pong
ca: RECEIVE ping
ca: SEND ac ! pong
```

```
25 ca: BECOME case pang
26 ac: RECEIVE pong
27 ac: SEND C ! pang
28 ca: RECEIVE pang
```

Analisi della receive

La regola di inferenza (RECEIVE) data per la ricezione dei messaggi richiede che l'attore abbia corpo vuoto per poter ricevere un messaggio, cioè assume che il dispatcher non passi all'attore nessun messaggio mentre questo ne sta processando un altro. Usiamo l'esempio del ping pong per mostrare che un programma Scala Akka si comporta esattamente in questo modo.

Supponiamo di trovarci nella seguente situazione:

$$F = [ab \mapsto \emptyset](dest \Rightarrow ..)ab\{become(pong() \Rightarrow ..)\} \mid [ba \mapsto \emptyset](\beta_{ba})ba\{ab\,!\,pong()\}$$

Per le regole di inferenza F può evolvere in F' utilizzando la regola (BECOME) su *ab* oppure in F'' utilizzando la regola (SEND) su *ba*. Intuitivamente vorremmo che il messaggio *pong* inviato da *ba* fosse effettivamente gestito da *ab*. Con l'attuale comportamento però, *ab* gestisce solo messaggi *dest*, quindi il messaggio *pong* sarebbe scartato per la regola (JUNK). Vediamo però che questo non succede: con l'interleaving $F \longrightarrow^{BECOME} F'$ l'attore sarà in grado di gestire il messaggio *pong* quando gli arriverà esattamente come ci aspettiamo, vediamo l'altro caso $F \longrightarrow^{SEND} F''$:

$$F'' = [ab \mapsto pong()](dest \Rightarrow ..)ab\{become(pong() \Rightarrow)\} \mid [ba \mapsto \emptyset](\beta_{ba})ba\{\underline{0}\}$$

.

Secondo le regole date, a questo punto *ba* ha corpo vuoto e mailbox vuota, quindi non può eseguire un passo di computazione; *ab* ha un messaggio in mailbox, ma non può eseguire una receive perchè non ha corpo vuoto; non può proseguire nemmeno con (JUNK) sempre perchè il corpo non è vuoto. L'unica regola che permette un passo di computazione è (BECOME), che fa assumere all'attore un comportamento che gli permette la ricezione di messaggi di tipo *pong*. Vediamo l'esecuzione di questo programma. Facciamo in modo che *ab*, subito prima di invocare il metodo become si metta in pausa per un minuto, tempo sufficiente per assicurarci che gli venga inviato un messaggio di tipo *pong* prima che l'attore sia in grado di riceverlo. Ci aspettiamo che, in questo programma, il messaggio *pong* venga processato correttamente, cioè l'attore *ab* completi l'esecuzione del corpo (in particolare quindi che esegua la BECOME) prima di passare al messaggio successivo.

Il codice dell'attore *ab* diventa:

```
1          val ab = context.actorOf(Props(new Actor{
```

```
2           def receive = {
3             case dest(y) =>
4               println("ab: RECEIVE dest(ba)")
5               y ! ping()
6
7               println("ab: .... GOING TO SLEEP ....")
8               Thread.sleep(60000)
9               println("ab: .... WAKING UP ....")
10
11
12              println("ab: BECOME case pong")
13              context.become({case pong() =>
14                println("ab: RECEIVE pong")
15                println("ab: CONTINUE"); y ! pang()})
16          }
17        }))
```

Output 1

```
1 ab: RECEIVE dest(ba)
2 ab: .... GOING TO SLEEP ....
3 ba: DO SOMETHING
4 ba: SEND ab ! pong
5 ba: CONTINUE
6 ab: .... WAKING UP ....
7 ab: BECOME case pong
8 ab: RECEIVE pong
9 ab: CONTINUE
```

Output 2

```
1 ab: RECEIVE dest(ba)
2 ba: DO SOMETHING
3 ab: .... GOING TO SLEEP ....
4 ba: SEND ab ! pong
5 ba: CONTINUE
6 ab: .... WAKING UP ....
7 ab: BECOME case pong
8 ab: RECEIVE pong
9 ab: CONTINUE
```

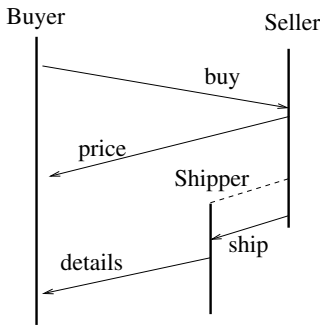


Figura 2.2: Buyer-Seller

Output 3

```
1  B: SEND ab ! dest(ba)
2  ab: RECEIVE dest(ba)
3  ba: DO SOMETHING
4  ba: SEND ab ! pong
5  ba: CONTINUE
6  ab: .... GOING TO SLEEP ....
7  ab: .... WAKING UP ....
8  ab: BECOME case pong
9  ab: RECEIVE pong
10 ab: CONTINUE
```

Vediamo che nelle esecuzioni il messaggio *pong* viene processato correttamente. Questo significa che l'attore attende di aver finito di processare il messaggio precedente prima di processarne uno nuovo. Quindi la semantica che abbiamo definito, tramite (RECEIVE), modella correttamente il comportamento degli attori Akka.

2.3.2 Buyer-Seller

Consideriamo un attore *Buyer* che invia ad un attore *Seller* il suo nome e l'oggetto che vuole comprare e aspetta risposta. Solo dopo aver saputo il prezzo *Buyer* può procedere con l'acquisto. *Seller* riceve il messaggio, risponde con il prezzo e genera un nuovo attore *Shipper* che interagisce direttamente con *Buyer* per concludere l'acquisto.

La situazione è la seguente:

$val\ Seller = Actor\{\beta_S; \underline{0}\}; val\ Buyer = Actor\{\beta_B; Seller\ !\ buy(Buyer, item)\};$

$\beta_B = \{price(z) \Rightarrow become(\beta'_B)\}$
$\beta'_B = \{details(w) \Rightarrow ...\}$
$\beta_S = \{buy(x,y) \Rightarrow x \,!\, price(f(y));\ val\ Shipper = Actor\{\beta_{SH}; \underline{0}\};\ Shipper \,!\, ship(x,y)\}$
$\beta_{SH} = \{ship(x,y) \Rightarrow x \,!\, details(f'(y))\}$

dove f e f' sono rispettivamente la funzione che calcola il prezzo di y e la funzione che genera i dettagli di consegna di y.

Questo esempio è utile per descrivere la gestione della mailbox. Analizziamo il programma: una volta inviata la richiesta *Buyer* si mette in attesa di una risposta di tipo $price(z)$. Quando la riceve cambia il suo comportamento e si mette in attesa di messaggi di tipo $details(w)$. Supponiamo di trovarci nella situazione appena successiva alla ricezione della richiesta da parte del *Seller*, cioè nella configurazione:

$F = (\nu Buyer)(\nu Seller)([Buyer \mapsto \emptyset](\beta_B)Buyer\{\underline{0}\} \mid [Seller \mapsto \emptyset](\beta_S)Seller\{e_S\})$

con $e_S ::= Buyer \,!\, price(f(item));$
$val\ Shipper = Actor\{\beta_{SH}; \underline{0}\};$
$Shipper \,!\, ship(Buyer, item)$

Supponiamo che lo scheduler scelga di eseguire tutto il corpo di *Seller* e tutto quello di *Shipper* prima di far ricevere *price* a *Buyer*.

Nota: in questo esempio e nei successivi, per comodità viene evidenziato ad ogni passo di computazione il termine che reagisce. Ad esempio $\underline{F_1} \mid F_2 \rightarrow F'_1 \mid F_2$ indica che al termine F_1 verrà applicata la regola che porta alla configurazione successiva.

Abbiamo quindi la seguente esecuzione (per comodità chiamiamo gli attori B, S ed Sh):

Consideriamo:

$e'_S ::= val\ Shipper = Actor\{\beta_{SH}; \underline{0}\};\ Shipper \,!\, ship(Buyer, item)$
$e''_S ::= Shipper \,!\, ship(Buyer, item)$

$F \longrightarrow^{SEND} (\nu S)(\nu B)([B \mapsto price(f(item))](\beta_B)B\{\underline{0}\} \mid \underline{[S \mapsto \emptyset](\beta_S)S\{e'_S\}})$

$\longrightarrow^{SPAWN} (\nu S)(\nu B)([B \mapsto price(f(item))](\beta_B)B\{\underline{0}\} \mid$
$(\nu\ Sh)(\underline{[S \mapsto \emptyset](\beta_S)S\{e''_S\}} \mid [Sh \mapsto \emptyset](\beta_{SH})Sh\{\underline{0}\}))$

$$\longrightarrow^{SEND} (\nu S)(\nu B)([B \mapsto price(f(item))](\beta_B)B\{\underline{0}\} \mid (\nu\ Sh)([S \mapsto \emptyset](\beta_S)S\{\underline{0}\} \mid \underline{[Sh \mapsto ship(B, item)](\beta_{SH})Sh\{\underline{0}\}}))$$

$$\longrightarrow^{RECEIVE} (\nu S)(\nu B)([B \mapsto price(f(item))](\beta_B)B\{\underline{0}\} \mid (\nu\ Sh)([S \mapsto \emptyset](\beta_S)S\{\underline{0}\} \mid \underline{[Sh \mapsto \emptyset](\beta_{SH})Sh\{B!details(f'(item))\}}))$$

$$\longrightarrow^{SEND} (\nu S)(\nu B)([B \mapsto price(f(item)) \cdot details(f'(item))](\beta_B)B\{\underline{0}\} \mid (\nu\ Sh)([S \mapsto \emptyset](\beta_S)S\{\underline{0}\} \mid [Sh \mapsto \emptyset](\beta_{Sh})Sh\{\underline{0}\}))$$

Notiamo che l'attore B continua a non conoscere il nome dell'attore Sh, anche se questo gli ha inviato un messaggio. Questo perché Sh ha inviato un messaggio a B senza inviare il riferimento a se stesso. In Akka invece il contesto di un attore memorizza il riferimento all'attore che gli ha inviato l'ultimo messaggio gestito (sender). Nella nostra sintassi non c'è ancora il concetto di contesto, che verrà introdotto nel capitolo successivo ma senza inserire il riferimento all'ultimo sender. Questo non è restrittivo, infatti possiamo comunque passare il riferimento all'attore che ha inviato il messaggio come parametro del messaggio stesso.

A questo punto *Buyer* ha in mailbox due messaggi, uno che può ricevere e uno no. Se scegliesse casualmente quale messaggio processare tra quelli presenti nella mailbox non avremmo la certezza che riesca a processarli entrambi. Infatti solo se sceglie di processare prima *price* il suo comportamento cambia e sarà poi in grado di processare *details*. Mostriamo invece che questa scelta non avviene in modo casuale, ma che l'attore sceglierà di processare sempre il messaggio arrivato per primo, sia nella semantica del linguaggio A (per come è definita infatti si applica solo (RECEIVE) e non (JUNK)), sia in Akka. Notiamo che in questo caso abbiamo la certezza che il messaggio *price* arriva prima di *details*, infatti *details* viene mandato da un attore creato solo dopo l'invio del primo messaggio.

Codice Scala Akka

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5
6  object BuyerSeller extends App{
7
8      case class buy(b:ActorRef, i:String)
9      case class price(i:String)
10     case class details(i:String)
11     case class ship(x:ActorRef, y:String)
```

```
12
13      val s = ActorSystem()
14
15      val Seller = s.actorOf(Props(new Actor{
16        def f(y: String) : String = y match{
17          case "sweater" => "50$"
18          case "T-shirt" => "20$"
19        }
20        def f2(y:String) : String =y match{
21          case x:String => "here the details"
22        }
23        def receive = {
24          case buy(x,y) =>
25            x ! price(f(y))
26            val Shipper = context.actorOf(Props(new Actor{
27              def receive = {case ship(x,y) => x!details(f2(y))}
28            }))
29            Shipper ! ship(x,y)
30        }
31      }))
32
33      val Buyer = s.actorOf(Props(new Actor{
34        Seller ! buy(self,"sweater")
35        def receive = {
36          case price(z) =>
37            context.become({case details(w) =>
38              println("Buyer: processing details");
39              })
40        }
41      }))
42  }
```

Eseguiamo il codice alcune volte e mostriamo che *Buyer* arriva sempre a gestire correttamente il messaggio *details*. Nel nostro linguaggio l'esempio è completamente deterministico, quindi il comportamento è unico. Akka non ha la semantica formale, quindi per vedere che l'esecuzione è anche qui deterministica, eseguiamo più volte e controlliamo che succeda quanto dichiarato nella documentazione.

```
1  scala> for(x <- 1 to 20){BuyerSeller.main(Array())}
2
3  Buyer: processing details
4  Buyer: processing details
5  Buyer: processing details
6  Buyer: processing details
7  Buyer: processing details
```

```
8  Buyer: processing details
9  Buyer: processing details
10 Buyer: processing details
11 Buyer: processing details
12 Buyer: processing details
13 Buyer: processing details
14 Buyer: processing details
15 Buyer: processing details
16 Buyer: processing details
17 Buyer: processing details
18 Buyer: processing details
19 Buyer: processing details
20 Buyer: processing details
21 Buyer: processing details
22 Buyer: processing details
```

2.3.3 La libreria

Consideriamo il caso di una libreria: un cliente richiede un libro, gli viene consegnato, lo legge e poi lo riconsegna alla libreria e il libro diventa di nuovo disponibile. Se un altro cliente richiede il libro mentre questo è in prestito non gli viene consegnato nulla. Consideriamo due clienti che competono per lo stesso libro, cioè la configurazione:

$[Book \mapsto \emptyset](\beta_B)Book\{\underline{0}\} \mid$
$[C_1 \mapsto \emptyset](\beta_{C_1})C_1\{Book \,!\, acquire(C_1)\} \mid$
$[C_2 \mapsto \emptyset](\beta_{C_2})C_2\{Book \,!\, acquire(C_2)\}$

$\beta_B = \{acquire(x) \Rightarrow become(\beta'_B); x \,!\, here(Book)\}$
$\beta'_B = \{release \Rightarrow become(\beta_B)\}$

$\beta_{C_1} = \beta_{C_2} = \{here(y) \Rightarrow y \,!\, release\}$
dove per convenzione $release$ corrisponde al messaggio senza parametri $release()$.

Partendo dalla configurazione iniziale supponiamo che il sistema faccia eseguire un passo di computazione a C_1 attraverso la regola (SEND) e poi un passo di computazione a C_2 sempre per (SEND). Ci troviamo nella seguente situazione:

$F = [Book \mapsto acquire(C_1) \cdot acquire(C_2)](\beta_B)Book\{\underline{0}\} \mid [C_1 \mapsto \emptyset](\beta_{C_1})C_1\{\underline{0}\} \mid$
$[C_2 \mapsto \emptyset](\beta_{C_2})C_2\{\underline{0}\}$

In accordo con la regola (RECEIVE) $Book$ processa il primo messaggio ricevuto, quindi cambia il comportamento in modo da non essere ulteriormente prestato e concede il libro a C_1

$$F \longrightarrow^{RECEIVE} [Book \mapsto acquire(C_2)](\beta_B)Book\{become(\beta'_B); C_1!here(Book)\} \mid [C_1 \mapsto \emptyset](\beta_{C_1})C_1\{\underline{0}\} \mid [C_2 \mapsto \emptyset](\beta_{C_2})C_2\{\underline{0}\}$$

le azioni successive sono deterministiche poiché C_1 e C_2 mantengono corpo vuoto

$$\longrightarrow^{BECOME} [Book \mapsto acquire(C_2)](\beta'_B)Book\{C_1!here(Book)\} \mid [C_1 \mapsto \emptyset](\beta_{C_1})C_1\{\underline{0}\} \mid [C_2 \mapsto \emptyset](\beta_{C_2})C_2\{\underline{0}\}$$

$$\longrightarrow^{SEND} [Book \mapsto acquire(C_2)](\beta'_B)Book\{\underline{0}\} \mid [C_1 \mapsto here(Book)](\beta_{C_1})C_1\{\underline{0}\} \mid [C_2 \mapsto \emptyset](\beta_{C_2})C_2\{\underline{0}\}$$

A questo punto sono applicabili due regole tra quelle date: (RECEIVE) applicata a C_1 perchè questo ha corpo vuoto e il messaggio in mailbox è corretto per il comportamento dell'attore; oppure (JUNK) applicata a *Book* che scarta la richiesta di C_2 in quanto al momento non concorda con il comportamento dell'attore. Se il sistema sceglie questa seconda strada ci aspettiamo per le regole date che C_2 non riesca mai ad acquisire il libro. Osserviamo lo stesso comportamento in Akka.

Codice Scala Akka

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5
6  object Book extends App{
7    case class acquire(x:ActorRef, s:String)
8    case class release(y:ActorRef)
9    case class here(w:ActorRef)
10
11   val s = ActorSystem()
12
13   val Book = s.actorOf(Props(new Actor{
14     def FREE:Actor.Receive = {
15       case acquire(x,s) =>
16           println("Book: BECOME BUSY")
17           context.become(BUSY);
18           println("Book: SEND " + s + " ! here")
19           x ! here(self)
20      }
21
22     def BUSY: Actor.Receive = {
```

```
23          case release =>
24              println("Book: BECOME FREE")
25              context.become(FREE)
26        }
27        def receive = FREE
28      }))
29
30
31      val Client1 = s.actorOf(Props(new Actor{
32        println("C1: SEND Book ! acquire")
33         Book ! acquire(self, "C1")
34         def receive = {
35           case here(y) =>
36             println("C1: RECEIVE Book")
37             println("C1: ... READING ...")
38             Thread.sleep(1000)
39             println("C1: SEND Book ! release")
40             y ! release
41         }
42      }))
43
44      val Client2 = s.actorOf(Props(new Actor{
45         println("C2: SEND Book ! acquire")
46         Book ! acquire(self, "C2")
47         def receive = {
48           case here(y) =>
49             println("C2: RECEIVE Book")
50             println("C2: ... READING ...")
51             Thread.sleep(1000)
52             println("C2: SEND Book ! release")
53             y ! release
54         }
55      }))
56
57  }
```

Output 1

```
1  C1: SEND Book ! acquire
2  C2: SEND Book ! acquire
3  Book: BECOME BUSY
4  Book: SEND C1 ! here
5  C1: RECEIVE Book
6  C1: ... READING ...
```

```
7 C1: SEND Book ! release
8 Book: BECOME FREE
```

Output 2

```
1 C2: SEND Book ! acquire
2 C1: SEND Book ! acquire
3 Book: BECOME BUSY
4 Book: SEND C2 ! here
5 C2: RECEIVE Book
6 C2: ... READING ...
7 C2: SEND Book ! release
8 Book: BECOME FREE
```

in queste esecuzioni abbiamo evidenza del caso in cui viene scelto di applicare la regola (JUNK): la prima richiesta che arriva viene soddisfatta, mentre la seconda viene ignorata perché l'attore ha cambiato comportamento. Forziamo ora il sistema in modo che l'attore C_2 invii la richiesta quando il libro è tornato disponibile, ovvero quando *Book* accetta nuovamente messaggi di tipo *acquire*. È sufficiente lasciar dormire il thread per un minuto prima di inviare un *acquire*, in modo da dare tempo a C_1 di restituire il libro.

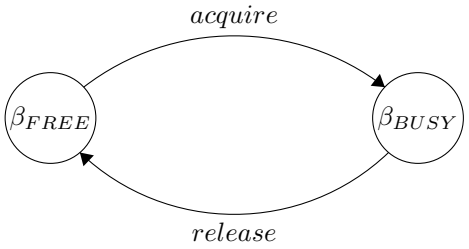
Output 3

```
1 C1: SEND Book ! acquire
2 Book: BECOME BUSY
3 Book: SEND C1 ! here
4 C1: RECEIVE Book
5 C1: ... READING ...
6 C1: SEND Book ! release
7 Book: BECOME FREE
8 C2: SEND Book ! acquire
9 Book: BECOME BUSY
10 Book: SEND C2 ! here
11 C2: RECEIVE Book
12 C2: ... READING ...
13 C2: SEND Book ! release
14 Book: BECOME FREE
```

2.3.4 Lock

Nell'esempio della libreria l'attore *Book* si comporta come un *lock*, ovvero un oggetto con due comportamenti: se non è acquisito da nessuno accetta richieste

di acquisizione, se è già impegnato le richieste falliscono. Il lock torna disponibile quando viene rilasciato da chi l'ha acquisito. Modelliamo in linguaggio A un programma in cui si ha un lock e due client che lo contendono. Consideriamo inoltre due gestioni diverse delle richieste fallite: nel primo caso il lock occupato restituisce un messaggio *busy* al richiedente, che aspetta un po' e poi invia una nuova richiesta; nel secondo caso il lock tiene traccia di chi l'ha richiesto mentre era occupato e, quando torna ad essere libero, invia un messaggio di notifica agli attori la cui richiesta era fallita. Notiamo che un lock può essere considerato una macchina a stati finiti con due stati: FREE e BUSY. Dallo stato FREE con la ricezione di un messaggio *acquire* passa nello stato BUSY e da lì torna allo stato FREE con input *release*. I due stati corrispondono strettamente ai due comportamenti possibili per l'attore *Lock*: β_{FREE} attende messaggi *acquire* che portano l'attore ad avere comportamento β_{BUSY}; l'attore con comportamento β_{BUSY} torna ad avere comportamento β_{FREE} quando cattura un messaggio *release*.



In entrambi gli esempi abbiamo necessità di controllare che il lock venga rilasciato proprio dallo stesso attore che l'ha acquisito. Per farlo ci aiutiamo con l'espressione $if-else$ e con i confronti nella forma $u_1 = u_2$, che non compaiono nella sintassi del linguaggio A, ma che usiamo qui per semplicità. Inoltre, indichiamo con $\beta(\tilde{y})$ il comportamento β che contiene le variabili libere $\tilde{y}$.

Polling Lock

Partiamo da una configurazione iniziale dove il lock non è acquisito e i due attori *Client*1 e *Client*2 hanno concorrentemente la possibilità di acquisirlo.

$$[](\beta_{FREE})Lock\{\underline{0}\} \mid [](\beta_{C_1})C_1\{Lock!acquire(C_1)\} \mid [](\beta_{C_2})C_2\{Lock!acquire(C_2)\}$$

$$\begin{aligned}\beta_{FREE} &= \{acquire(x) \Rightarrow become(\beta_{BUSY}(x)); x \,!\, here(Lock)\}\\ \beta_{BUSY}(y) &= \{release(x) \Rightarrow if(x = y)\ become(\beta_{FREE})\\ &\qquad\quad acquire(x) \Rightarrow x \,!\, busy(Lock)\}\end{aligned}$$

$$\begin{aligned}\beta_{C_1} &= \{here(x) \Rightarrow e_{C_1}; x \,!\, release(C_1)\\ &\qquad busy(x) \Rightarrow x \,!\, acquire(C_1)\}\end{aligned}$$

dove e_C è un'espressione qualsiasi che utilizza il lock. β_{C_2} è analogo.

Il comportamento parametrizzato $\beta_{BUSY}(y)$ modella il comportamento del lock quando è detenuto dall'attore y.

Abbiamo visto che il comportamento del *Lock* può essere visto come stato della macchina a stati finiti. Vediamo quindi le esecuzioni possibili che conseguono ad una richiesta, a seconda dello stato in cui si trova il lock al momento di una richiesta, ovvero FREE (i.e. comportamento β_{FREE}) o BUSY (i.e. comportamento β_{BUSY}).

Caso 1: lock nello stato FREE

Supponiamo C_1 abbia inviato una richiesta al lock. La situazione runtime è la seguente:

$$[Lock \mapsto acquire(C_1)](\beta_{FREE})Lock\{\underline{0}\} \mid []({\beta_{C_1}})C_1\{\underline{0}\}$$

in accordo con le regola date questa configurazione evolve in:

$$\longrightarrow^{RECEIVE} \underline{[](\beta_{FREE})Lock\{become(\beta_{BUSY}(C_1)); C_1 \,!\, here(Lock)\}} \mid []({\beta_{C_1}})C_1\{\underline{0}\}$$

$$\longrightarrow^{BECOME} \underline{[](\beta_{BUSY}(C_1))Lock\{C_1 \,!\, here(Lock)\}} \mid []({\beta_{C_1}})C_1\{\underline{0}\}$$

$$\longrightarrow^{SEND} []({\beta_{BUSY}(C_1)})Lock\{\underline{0}\}\} \mid \underline{[C_1 \mapsto here(Lock)](\beta_{C_1})C_1\{\underline{0}\}}$$

$$\longrightarrow^{RECEIVE} []({\beta_{BUSY}(C_1)})Lock\{\underline{0}\} \mid \underline{[](\beta_{C_1})C_1\{e_{C_1}; Lock \,!\, release(C_1)\}}$$

$$\longrightarrow^{\cdots} []({\beta_{BUSY}(C_1)})Lock\{\underline{0}\} \mid \underline{[](\beta_{C_1})C_1\{Lock \,!\, release(C_1)\}}$$

$$\longrightarrow^{SEND} \underline{[Lock \mapsto release(C_1)](\beta_{BUSY}(C_1))Lock\{\underline{0}\}} \mid []({\beta_{C_1}})C_1\{\underline{0}\}$$

$$\longrightarrow^{RECEIVE} \underline{[](\beta_{BUSY}(C_1))Lock\{if(C_1 = C_1)\ become(\beta_{FREE})\}} \mid []({\beta_{C_1}})C_1\{\underline{0}\}$$

$$\longrightarrow^{BECOME} []({\beta_{FREE}})Lock\{\underline{0}\} \mid []({\beta_{C_1}})C_1\{\underline{0}\}$$

Caso 2: lock nello stato BUSY

Nel caso 1 abbiamo supposto che C_1 fosse l'unico ad aver fatto una richiesta di *acquire* sul lock. Vediamo ora cosa succede se l'attore C_2 invia una richiesta dopo C_1. L'ordine di ricezione dei messaggi è deterministico, in quanto la mailbox viene trattata come coda FIFO. Quindi se il primo *acquire* arrivato è di C_1, abbiamo due casi possibili:

- $acquire(C_2)$ arriva dopo la *release* di C_1

- $acquire(C_2)$ arriva prima della $release$ di C_1

Nel primo caso il lock è tornato in uno stato FREE e ci riconduciamo al caso 1. Nel secondo caso il messaggio viene ricevuto quando il lock è in uno stato BUSY. Ci aspettiamo quindi la seguente esecuzione, in accordo con le regola date:

$$[](\beta_{BUSY}(C_1))Lock\{\underline{0}); \} \mid \underline{[](\beta_{C_2})C_2\{Lock!acquire(C_2)\}}$$

$$\longrightarrow^{SEND} \underline{[Lock \mapsto acquire(C_2)](\beta_{BUSY}(C_1))Lock\{\underline{0}\}} \mid [](\beta_{C_2})C_2\{\underline{0}\}$$

$$\longrightarrow^{RECEIVE} \underline{[](\beta_{BUSY}(C_1))Lock\{C_2 \,!\, busy(Lock); \}} \mid [](\beta_{C_2})C_2\{\underline{0}\}$$

$$\longrightarrow^{SEND} [](\beta_{BUSY}(C_1))Lock\{\underline{0}\} \mid \underline{[C_2 \mapsto busy(Lock)](\beta_{C_2})C_2\{\underline{0}\}}$$

$$\longrightarrow^{RECEIVE} [](\beta_{BUSY}(C_1))Lock\{\underline{0}\} \mid [](\beta_{C_2})C_2\{Lock \,!\, acquire(C_2)\}$$

da qui si ripresentano i due casi sopra: o nel frattempo è arrivato un messaggio $release$ e C_2 può acquisire il lock, altrimenti continua a ritentare finché non lo ottiene.

Verifichiamo runtime che le esecuzioni previste siano esattamente quelle ottenute eseguendo il programma scritto in Scala Akka e vediamo che in effetti, se il lock è occupato con C_i, allora C_j fa polling.

Codice Scala Akka

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5
6  object PollingLock extends App{
7    case class acquire(x:ActorRef)
8    case class release(x:ActorRef)
9    case class here(x:ActorRef)
10   case class busy(x: ActorRef)
11
12   val s = ActorSystem()
13
14   val Lock = s.actorOf(Props(new Actor{
15     def FREE:Actor.Receive = {
16       case acquire(x) =>
17           println("Lock: BECOME BUSY")
```

```
18          context.become(BUSY(x));
19          println("Lock concesso a " + x.path.name)
20          x ! here(self)
21        }
22
23      def BUSY(client: ActorRef): Actor.Receive = {
24          case release(x) =>
25            if(x == client){
26             println("Lock: BECOME FREE")
27             context.become(FREE)
28            }
29          case acquire(x) =>
30             x ! busy(self)
31
32      }
33      def receive = FREE
34    }))
35
36    val Client1 = s.actorOf(Props(new Actor{
37      println("C1 richiede il lock")
38       Lock ! acquire(self)
39       def receive = {
40          case here(y) =>
41            println("C1 acquisisce il lock")
42            Thread.sleep(6000)
43            println("C1 rilascia il lock")
44            y ! release(self)
45          case busy(y) =>
46            println("C1 fallisce")
47            Thread.sleep(1000)
48            println("C1 richiede nuovamente il lock")
49            y ! acquire(self)
50       }
51    }), "C1")
52
53    val Client2 = s.actorOf(Props(new Actor{
54      println("C2 richiede il lock")
55       Lock ! acquire(self)
56       def receive = {
57          case here(y) =>
58            println("C2 acquisisce il lock")
59            Thread.sleep(6000)
60            println("C2 rilascia il lock")
61            y ! release(self)
62          case busy(y) =>
```

```
63            println("C2 fallisce")
64            Thread.sleep(1000)
65            println("C2 richiede nuovamente il lock")
66            y ! acquire(self)
67        }
68    }), "C2")
69
70  }
```

Output 1

```
1  C1 richiede il lock
2  C2 richiede il lock
3  Lock: BECOME BUSY
4  Lock concesso a C1
5  C1 acquisisce il lock
6  C2 fallisce
7  C2 richiede nuovamente il lock
8  C2 fallisce
9  C2 richiede nuovamente il lock
10 C2 fallisce
11 C2 richiede nuovamente il lock
12 C2 fallisce
13 C2 richiede nuovamente il lock
14 C2 fallisce
15 C2 richiede nuovamente il lock
16 C2 fallisce
17 C1 rilascia il lock
18 Lock: BECOME FREE
19 C2 richiede nuovamente il lock
20 Lock: BECOME BUSY
21 Lock concesso a C2
22 C2 acquisisce il lock
```

Output 2

```
1  C2 richiede il lock
2  C1 richiede il lock
3  Lock: BECOME BUSY
4  Lock concesso a C2
5  C2 acquisisce il lock
6  C1 fallisce
7  C1 richiede nuovamente il lock
```

```
C1 fallisce
C1 richiede nuovamente il lock
C1 fallisce
C1 richiede nuovamente il lock
C1 fallisce
C1 richiede nuovamente il lock
C1 fallisce
C1 richiede nuovamente il lock
C1 fallisce
C2 rilascia il lock
Lock: BECOME FREE
C1 richiede nuovamente il lock
Lock: BECOME BUSY
Lock concesso a C1
C1 acquisisce il lock
```

Output 3

```
C1 richiede il lock
Lock: BECOME BUSY
Lock concesso a C1
C2 richiede il lock
C1 acquisisce il lock
C2 fallisce
C2 richiede nuovamente il lock
C2 fallisce
C2 richiede nuovamente il lock
C2 fallisce
C2 richiede nuovamente il lock
C2 fallisce
C2 richiede nuovamente il lock
C2 fallisce
C2 richiede nuovamente il lock
C2 fallisce
C1 rilascia il lock
Lock: BECOME FREE
C2 richiede nuovamente il lock
Lock: BECOME BUSY
Lock concesso a C2
C2 acquisisce il lock
```

Kind Lock

Cambiamo ora il comportamento del lock in modo da eliminare il polling e fare in modo che tenga traccia dei nomi degli attori che eseguono le richieste non soddisfatte e li avverta una volta tornato nello stato FREE. Cambiamo i comportamenti come segue:

$$\beta_{FREE} = \{acquire(x) \Rightarrow become(\beta_{BUSY}(x, \emptyset)); x \,!\, here(Lock)\}$$

$$\beta_{BUSY}(z, \tilde{y}) = \{release(x) \Rightarrow if(x = z)\ become(\beta_{FREE}); (y \,!\, free(Lock))_{y \in \tilde{y}}$$
$$acquire(x) \Rightarrow become(\beta_{BUSY}(z, (\tilde{y}, x)))\}$$

$$\beta_{C_1} = \{here(x) \Rightarrow e_{C_1}; x \,!\, release(C_1)$$
$$free(x) \Rightarrow x \,!\, acquire(C_1)\}$$

β_{C_2} analogo.

Il comportamento β_{BUSY} ha ora due parametri: z che come prima identifica l'attore che detiene il lock, e una tupla (possibilmente vuota) $\tilde{y}$ che contiene i nomi degli attori che richiedono il lock mentre questo è detenuto da z. Ogni volta che un attore fa una richiesta di acquisizione al lock impegnato, questo aggiorna la tupla degli attori insoddisfatti e assume un nuovo comportamento che ha come secondo parametro la nuova tupla. Quando il lock viene rilasciato da z, lo stesso lock effettua una sequenza di invii di messaggi $(y \,!\, free(Lock))_{y \in \tilde{y}}$ a tutti gli attori y appartenenenti alla tupla $\tilde{y}$ degli insoddisfatti, per notificare che è tornato disponibile.

Analizziamo quindi la nuova esecuzione che consegue alla richiesta di un lock quando questo si trova in uno stato BUSY.

$$[](\beta_{BUSY}(C_1, \emptyset))Lock\{\underline{0}); \} \mid \underline{[](\beta_{C_2})C_2\{Lock!acquire(C_2)\}}$$

$$\longrightarrow^{SEND} \underline{[Lock \mapsto acquire(C_2)](\beta_{BUSY}(C_1, \emptyset))Lock\{\underline{0}\}} \mid [](\beta_{C_2})C_2\{\underline{0}\}$$

$$\longrightarrow^{RECEIVE} \underline{[](\beta_{BUSY}(C_1, \emptyset)Lock\{become(\beta_{BUSY}(C_1, C_2))\}} \mid [](\beta_{C_2})C_2\{\underline{0}\}$$

$$\longrightarrow^{BECOME} [](\beta_{BUSY}(C_1, C_2))Lock\{\underline{0}\} \mid [](\beta_{C_2})C_2\{\underline{0}\}$$

quando poi il lock viene rilasciato da C_1 con un messaggio $release$, il lock notifica a tutti gli attori che lo avevano richiesto che è tornato disponibile:

$$\underline{[Lock \mapsto release(C_1)](\beta_{BUSY}(C_1, C_2))Lock\{\underline{0}\}} \mid [](\beta_{C_2})C_2\{\underline{0}\}$$

$$\longrightarrow^{RECEIVE} \underline{[](\beta_{BUSY}(C_1, C_2))Lock\{if(C_1 = C_1)\ become(\beta_{FREE}); C_2 \,!\, free(Lock)\}} \mid$$
$$[](\beta_{C_2})C_2\{\underline{0}\}$$

$\longrightarrow^{BECOME} \underline{[](\beta_{FREE})Lock\{C_2 \,!\, free(Lock)\}} \mid [](\beta_{C_2})C_2\{\underline{0}\}$

$\longrightarrow^{SEND} [](\beta_{FREE})Lock\{\underline{0}\} \mid \underline{[C_1 \mapsto free(Lock)](\beta_{C_2})C_2\{\underline{0}\}}$

$\longrightarrow^{RECEIVE} [](\beta_{FREE})Lock\{\underline{0}\} \mid [](\beta_{C_2})C_2\{Lock \,!\, acquire(C_2)\}$

Codice Scala Akka

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5  import scala.collection.mutable.ArrayBuffer
6
7  object KindLock extends App{
8    case class acquire(x:ActorRef)
9    case class release(x: ActorRef)
10   case class here(x:ActorRef)
11   case class free(x: ActorRef)
12
13   val s = ActorSystem()
14
15   val Lock = s.actorOf(Props(new Actor{
16     def FREE: Actor.Receive = {
17       case acquire(x) =>
18          val buf = new ArrayBuffer[ActorRef]()
19          println("Lock: BECOME BUSY")
20          context.become(BUSY(x,buf));
21          println("Lock concesso a " + x.path.name)
22          x ! here(self)
23      }
24
25     def BUSY(client:ActorRef, buf:ArrayBuffer[ActorRef]):Actor.Receive={
26        case release(x) =>
27          if(x==client){
28           println("Lock: BECOME FREE")
29           context.become(FREE)
30           for (i <- 0 to buf.size-1){
31             println("Lock: NOTIFY to " + buf(i).path.name)
32             buf(i) ! free(self)
33           }
34          }
```

```
35          case acquire(x) =>
36             buf.+=(x)
37             context.become(BUSY(client, buf))
38        }
39        def receive = FREE
40      }))
41
42
43      val Client1 = s.actorOf(Props(new Actor{
44        println("C1 richiede il lock")
45         Lock ! acquire(self)
46         def receive = {
47           case here(y) =>
48             println("C1 acquisisce il lock")
49             println(".... C1 working ...")
50             Thread.sleep(6000)
51             println("C1 rilascia il lock")
52             y ! release(self)
53           case free(y) =>
54             println("C1 richiede nuovamente il lock")
55             y ! acquire(self)
56         }
57      }), "C1")
58
59      val Client2 = s.actorOf(Props(new Actor{
60        println("C2 richiede il lock")
61         Lock ! acquire(self)
62         def receive = {
63           case here(y) =>
64             println("C2 acquisisce il lock")
65             println(".... C2 working ...")
66             Thread.sleep(6000)
67             println("C2 rilascia il lock")
68             y ! release(self)
69           case free(y) =>
70             println("C2 richiede nuovamente il lock")
71             y ! acquire(self)
72         }
73      }), "C2")
74
75  }
```

Output 1

```
C1 richiede il lock
C2 richiede il lock
Lock: BECOME BUSY
Lock concesso a C2
C2 acquisisce il lock
.... C2 working ...
C2 rilascia il lock
Lock: BECOME FREE
Lock: NOTIFY to C1
C1 richiede nuovamente il lock
Lock: BECOME BUSY
Lock concesso a C1
C1 acquisisce il lock
.... C1 working ...
C1 rilascia il lock
Lock: BECOME FREE
```

Vale la pena sottolineare che le due stampe C1 richiede il lock e C2 richiede il lock non sono completamente indicative del momento in cui viene effettuata la richiesta. Le stampe sono infatti asincrone, come dimostrato dal fatto che il lock viene concesso a C2, che è quindi, per la gestione a coda FIFO della mailbox, il primo che ha inviato una richiesta al Lock.

Osserviamo inoltre che, in accordo con la semantica asincrona del modello ad attori, se un attore chiede un lock mentre questo è occupato, l'attore non si sospende in attesa che il lock si liberi, ma prosegue la sua esecuzione.

Infine il lock qui modellato non serve nel modello ad attori, in quanto il modello è costruito per essere appunto asincrono e non bloccante. Tuttavia il lock ci torna utile come esempio per testare il nostro linguaggio A.

Output 2

```
C1 richiede il lock
C2 richiede il lock
Lock: BECOME BUSY
Lock concesso a C1
C1 acquisisce il lock
.... C1 working ...
C1 rilascia il lock
Lock: BECOME FREE
Lock: NOTIFY to C2
C2 richiede nuovamente il lock
Lock: BECOME BUSY
Lock concesso a C2
```

```
13 C2 acquisisce il lock
14 .... C2 working ...
15 C2 rilascia il lock
16 Lock: BECOME FREE
```

La libreria gentile

Con l'implementazione del lock in forma gentile, che quando torna libero avverte gli attori che ne avevano fatto richiesta, possiamo estendere anche l'esempio della libreria creando una libreria gentile che avverte il cliente quando un libro é tornato disponibile.

$$[](\beta_B)Book\{\underline{0}\} \mid (\beta_{C_1})C_1\{Book!acquire(C_1)\} \mid (\beta_{C_2})C_2\{Book!acquire(C_2)\}$$

$$\beta_B = \{acquire(x) \Rightarrow become(\beta'_B(\emptyset)); x \,!\, here(Book)\}$$

$$\begin{aligned}\beta'_B(\tilde{y}) = \{&release \Rightarrow become(\beta_B);\ (y \,!\, free(Book))_{y\in\tilde{y}}\\ &acquire(x) \Rightarrow become(\beta'_B(\tilde{y}, x))\}\end{aligned}$$

$$\begin{aligned}\beta_{C_1} = \beta_{C_2} = \{&here(x) \Rightarrow x \,!\, release\\ &free(x) \Rightarrow x \,!\, acquire(C_1)\}\end{aligned}$$

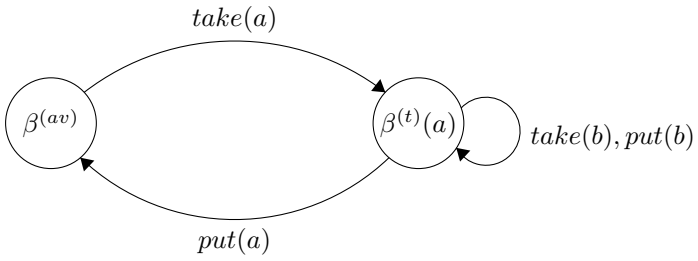
2.3.5 Tre filosofi a cena

Implementiamo nel linguaggio A il problema dei filosofi a cena: ci sono 3 filosofi e 3 bacchette; ogni filosofo ha una bacchetta alla sua destra e una alla sua sinistra e ha bisogno di due bacchette per mangiare. In particolare l'esempio che riportiamo è un riadattamento dell'esempio [1].
Chiamiamo le tre bacchette C_1, C_2 e C_3 e consideriamo tre filosofi $Kant$, $Hegel$ e $Marx$.

Possiamo considerare ogni bacchetta un lock che, come abbiamo visto nell'esempio precedente, può a sua volta essere considerato una macchina a stati finiti. La bacchetta può essere disponibile o acquisita dal filosofo a. Definiamo quindi i comportamenti corrispondenti a questi due stati direttamente in linguaggio A e visualizziamo il diagramma a stati che identifica il protocollo d'uso dell'oggetto bacchetta.

$$\begin{aligned}&\beta_{C_i}^{(available)} = \{take(x) \Rightarrow become(\beta_{C_i}^{(taken)}(x));\ x \,!\, taken(C_i)\}\\ &\beta_{C_i}^{(taken)}(y) = \{take(x) \Rightarrow x \,!\, busy(C_i)\\ &\qquad\qquad put(x) \Rightarrow if(x = y)\ become(\beta_{C_i}^{(available)})\ else\ \underline{0}\}\end{aligned}$$

con $i = 1, 2, 3$.



Anche in questo esempio, per semplicità, abbiamo introdotto i confronti nella forma $u_1 = u_2$ e l'espressione $if(u_1 = u_2)\ e_1\ else\ e_2$ che non sono presenti nel linguaggio A. Banalmente se la condizione di uguaglianza dell'if è verificata viene eseguito il ramo if, in caso contrario il ramo $else$.

Un filosofo inizialmente pensa. Quando ha fame invia una richiesta alle sue due bacchette. Se entrambe le bacchette sono disponibili inizia a mangiare. Se una sola delle bacchette è disponibile il filosofo, dopo aver accertato che l'altra non lo sia, ripone quella acquisita e torna a pensare. Se invece entrambe le bacchette sono occupate il filosofo torna a pensare. Possiamo descrivere questo protocollo attraverso i seguenti comportamenti in linguaggio A. I comportamenti sono scritti per il filosofo $Kant$ a cui sono associate le bacchette C_1 e C_2, ma sono facilmente riscrivibili per gli altri due filosofi:

$$\beta_K = \{think \Rightarrow become(\beta_K^{(think)});\ e_{think};\ Kant\,!\,eat\}$$

$$\beta_K^{(think)} = \{eat \Rightarrow become(\beta_K^{(hungry)});\ C_1\,!\,take(Kant);\ C_2\,!\,take(Kant)\}$$

$$\begin{aligned}\beta_K^{(hungry)} = \{&taken(x) \Rightarrow if(x = C_1)\ become(\beta_K^{(wait)}(C_2, C_1))\\ &\qquad\qquad else\ if(x = C_2)\ become(\beta_K^{(wait)}(C_1, C_2))\\ &\qquad\qquad else\ \underline{0}\\ &busy(x) \Rightarrow become(\beta_K^{(denied)})\}\end{aligned}$$

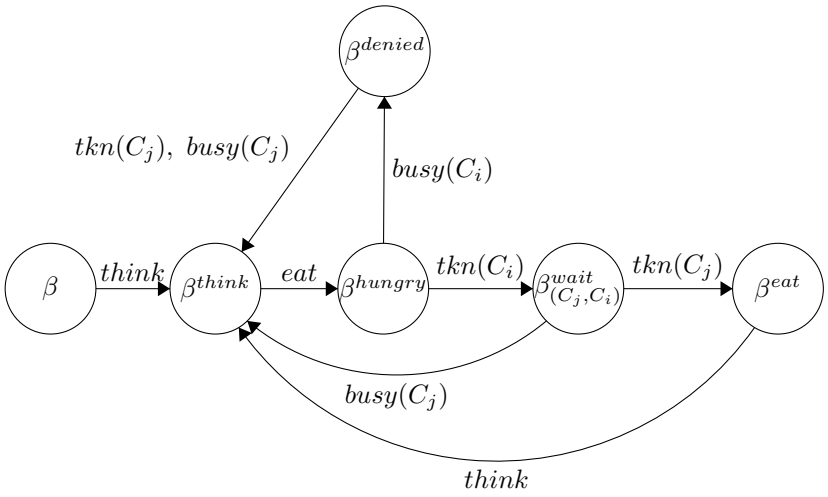
$$\begin{aligned}\beta_K^{(wait)}(y, y') = \{&taken(x) \Rightarrow if\ (x = y)\ become(\beta_K^{(eat)});\ e_{eat};\ Kant\,!\,think\ else\ \underline{0}\\ &busy(x) \Rightarrow y'\,!\,put(Kant);\ become(\beta_K^{(think)});\ e_{think};\ Kant\,!\,eat\}\end{aligned}$$

$$\begin{aligned}\beta_K^{(denied)} = \{&taken(x) \Rightarrow x\,!\,put(Kant);\ become(\beta_K^{(think)});\ e_{think};\ Kant\,!\,eat\\ &busy(x) \Rightarrow become(\beta_K^{(think)});\ e_{think};\ Kant\,!\,eat\}\end{aligned}$$

$$\beta_K^{(eat)} = \{think \Rightarrow C_1\,!\,put(Kant);\ C_2\,!\,put(Kant);\ become(\beta_K^{(think)});\ e_{think};\ Kant\,!\,eat\}$$

$$e_K = Kant\,!\,think$$

Il diagramma a stati che si genera è il seguente:



Un filosofo in uno stato iniziale risponde ad un messaggio *think* e passa in uno stato $\beta^{(think)}$. Ricordiamo che qui indichiamo gli stati con i nomi dei comportamenti assunti dal filosofo, per lo stretto collegamento che c'è tra questi due concetti. Quando ha fame, un filosofo manda a se stesso un messaggio *eat* che lo fa passare in uno stato $\beta^{(hungry)}$, e invia una richiesta alle due bacchette. A seconda della risposta che riceve per prima da una delle due bacchette, il filosofo passa dallo stato $\beta^{(hungry)}$ ad uno stato $\beta^{(wait)}$ (se ha acquisito una delle bacchette) o $\beta^{(denied)}$ se la prima bacchetta ha risposto di essere occupata. Se la prima bacchetta è stata negata (stato $\beta^{(denied)}$), il filosofo aspetta il messaggio della seconda bacchetta. Se anche questa è occupata torna direttamente nello stato $\beta^{(think)}$, altrimenti prima ripone la seconda bacchetta appena acquisita e poi torna allo stato $\beta^{(think)}$. Se invece la prima bacchetta è stata acquisita (il filosofo si trova nello stato $\beta^{(wait)}$) allora, se anche la seconda bacchetta risponde di essere libera il filosofo inizia a mangiare e passa quindi nello stato $\beta^{(eat)}$. Altrimenti ripone la prima bacchetta e torna nello stato $\beta^{(think)}$. Quando il filosofo si trova nello stato $\beta^{(eat)}$ mangia per un po', poi ripone le bacchette e torna in uno stato $\beta^{(think)}$, da cui ricomincia il ciclo.

Scriviamo il programma in linguaggio A e vediamo come evolve. Supponiamo di aver già creato tre filosofi $Kant$, $Hegel$ e $Marx$ e tre bacchette C_1, C_2 e C_3. Supponiamo inoltre che a $Kant$ siano state associate le bacchette C_1 e C_2 (come già visto sopra), a $Hegel$ le bacchette C_2 e C_3 e a $Marx$ le bacchette C_3 e C_1. Partiamo quindi dalla seguente configurazione:

$$[](\beta^{(available)}_{C_1})\ C_1\{\underline{0}\} \mid [](\beta^{(available)}_{C_2})\ C_2\{\underline{0}\} \mid [](\beta^{(available)}_{C_3})\ C_3\{\underline{0}\} \mid$$
$$[](\beta_K)\ Kant\{e_K\} \mid [](\beta_H)\ Hegel\{e_H\} \mid [](\beta_M)\ Marx\{e_M\}$$

Una bacchetta C_1 con $i \in \{1, 2, 3\}$, runtime, si comporta nei seguenti modi, in accordo con le regole date:

1. La bacchetta è disponibile. Le arriva la richiesta del filosofo a, allora passa nello stato taken by a e invia una notifica al filosofo:

   $[C_i \mapsto take(a)](\beta_{C_i}^{(available)})C_i\{\underline{0}\}$

   $\longrightarrow^{RECEIVE}\ [](\beta_{C_i}^{(available)})C_i\{become(\beta_{C_i}^{(taken)}(a));\ a\,!\,taken(Ci)\}$

   $\longrightarrow^{BECOME}\ [](\beta_{C_i}^{(taken)}(a))C_i\{a\,!\,taken(Ci)\}$

   $\longrightarrow^{SEND}\ [](\beta_{C_i}^{(taken)}(a)\})C_i\{\underline{0}\}$

2. La bacchetta è stata presa dal filosofo a e le arriva una richiesta dal filosofo b. Allora risponde a b avvisando che è già occupata:

   $[C_i \mapsto take(b)](\beta_{C_i}^{(taken)}(a))C_i\{\underline{0}\}$

   $\longrightarrow^{RECEIVE}\ [](\beta_{C_i}^{(taken)}(a))C_i\{b\,!\,busy(C_1)\}$

   $\longrightarrow^{SEND}\ [](\beta_{C_i}^{(taken)}(a))C_i\{\underline{0}\}$

3. La bacchetta è stata presa dal filosofo a e viene rilasciata dal filosofo a:

   $[C_i \mapsto put(a)](\beta_{C_i}^{(taken)}(a))C_i\{\underline{0}\}$

   $\longrightarrow^{RECEIVE}\ [](\beta_{C_i}^{(taken)}(a))C_i\{become(\beta_{C_i}^{(available)})\}$

   $\longrightarrow^{BECOME}\ [](\beta_{C_i}^{(available)})C_i\{\underline{0}\}$

Analizziamo ora l'esecuzione dei filosofi. Prendiamo come esempio il filosofo $Kant$:

$[](\beta_K)Kant\{e_K\}\ \rightarrow^{SEND}\ [Kant \mapsto think](\beta_K)Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ [](\beta_K)Kant\{become(\beta_K^{(think)});\ e_{think};\ Kant\,!\,eat\}$

$\longrightarrow^{BECOME}\ [](\beta_K^{(think)})Kant\{e_{think};\ Kant\,!\,eat\}$

$\longrightarrow^{\cdots}\ []({\beta_K^{(think)}})Kant\{Kant\,!\,eat\}$

$\longrightarrow^{SEND}\ [Kant \mapsto eat]({\beta_K^{(think)}})Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ []({\beta_K^{(think)}})Kant\{become({\beta_K^{(hungry)}});\ C_1\,!\,take(Kant);\ C_2\,!\,take(Kant)\}$

$\longrightarrow^{BECOME}\ []({\beta_K^{(hungry)}})Kant\{C_1\,!\,take(Kant);\ C_2\,!\,take(Kant)\}$

$\longrightarrow^{SEND}\ []({\beta_K^{(hungry)}})Kant\{C_2\,!\,take(Kant)\}$

$\longrightarrow^{SEND}\ []({\beta_K^{(hungry)}})Kant\{\underline{0}\}$

Anche gli altri filosofi possono procedere in parallelo quindi, a seconda dello stato di occupazione delle forchette, l'esecuzione può prendere quatto strade diverse:

Caso 1: le forchette sono entrambe libere. La prima che risponde è C_2 e poi risponde C_1.

$[Kant \mapsto taken(C_2) \cdot taken(C_1)]({\beta_K^{(hungry)}})Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ [Kant \mapsto taken(C_1)]({\beta_K^{(hungry)}})Kant\{become({\beta_K^{(wait)}}(C_1, C_2))\}$

$\longrightarrow^{BECOME}\ [Kant \mapsto taken(C_1)]({\beta_K^{(wait)}}(C_1, C_2))Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ []({\beta_K^{(wait)}}(C_1, C_2)Kant\{become({\beta_K^{(eat)}});\ e_{eat};\ Kant\,!\,think\}$

$\longrightarrow^{BECOME}\ []({\beta_K^{(eat)}})Kant\{e_{eat};\ Kant\,!\,think\}$

$\longrightarrow^{\cdots}\ []({\beta_K^{(eat)}})Kant\{Kant\,!\,think\}$

$\longrightarrow^{SEND}\ [Kant \mapsto think]({\beta_K^{(eat)}})Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ []({\beta_K^{(eat)}})Kant\{C_1\,!\,put(Kant);\ C_2\,!\,put(Kant);\ become({\beta_K^{(think)}});\ e_{think};\ Kant\,!\,eat\}$

$\longrightarrow^{SEND}\ []({\beta_K^{(eat)}})Kant\{C_2\,!\,put(Kant);\ become({\beta_K^{(think)}});\ e_{think};\ Kant\,!\,eat\}$

$\longrightarrow^{SEND}\ []({\beta_K^{(eat)}})Kant\{become({\beta_K^{(think)}});\ e_{think};\ Kant\,!\,eat\}$

$\longrightarrow^{BECOME}\ []({\beta_K^{(think)}})Kant\{e_{think};\ Kant\,!\,eat\}$

$\longrightarrow^{\cdots}\ [](\beta_K^{(think)})Kant\{Kant\,!\,eat\}$

$\longrightarrow^{SEND}\ [Kant \mapsto eat](\beta_K^{(think)})Kant\{\underline{0}\}$

e ricomincia il ciclo.

Notiamo che se il secondo messaggio non fosse ancora arrivato, il filosofo si sarebbe sospeso dopo la seconda reazione, ovvero nella configurazione $[](\beta_K^{(wait)}(C_1, C_2))Kant\{\underline{0}\}$, per passare poi alla configurazione $[Kant \mapsto taken(C_1)](\beta_K^{(wait)}(C_1, C_2))Kant\{\underline{0}\}$ con l'arrivo del messaggio e continuare come descritto.

Caso 2: la prima bacchetta che risponde è C_2 ed è libera, mentre la seconda, C_1, no.

$[Kant \mapsto taken(C_2) \cdot busy(C_1)](\beta_K^{(hungry)})Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ [Kant \mapsto busy(C_1)](\beta_K^{(hungry)})Kant\{become(\beta_K^{(wait)}(C_1, C_2))\}$

$\longrightarrow^{BECOME}\ [Kant \mapsto busy(C_1)](\beta_K^{(wait)}(C_1, C_2))Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ [](\beta_K^{(wait)}(C_1, C_2))Kant\{C_2\,!\,put(Kant);\ become(\beta_K^{(think)});\ Kant\,!\,eat\}$

$\longrightarrow^{SEND}\ [](\beta_K^{(wait)}(C_1, C_2))Kant\{become(\beta_K^{(think)});\ e_{think};\ Kant\,!\,eat\}$

$\longrightarrow^{BECOME}\ [](\beta_K^{(think)})Kant\{e_{think};\ Kant\,!\,eat\}$

$\longrightarrow^{\cdots}\ [](\beta_K^{(think)})Kant\{Kant\,!\,eat\}$

$\longrightarrow^{SEND}\ [eat](\beta_K^{(think)})Kant\{\underline{0}\}$

e ricomincia il ciclo.

Anche qui, se il secondo messaggio (in questo caso *busy*) non fosse ancora arrivato, il filosofo si sospenderebbe dopo la seconda riga in attesa che arrivi.

Caso 3: la prima bacchetta che risponde è C_2 che è occupata, mentre la seconda (C_1) è libera.

$[Kant \mapsto busy(C_2) \cdot taken(C_1)](\beta_K^{(hungry)})Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ [Kant \mapsto taken(C_1)](\beta_K^{(hungry)})Kant\{become(\beta_K^{(denied)})\};$

$\longrightarrow^{BECOME}\ [Kant \mapsto taken(C_1)](\beta_K^{(denied)})Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ [](\beta_K^{(denied)})Kant\{C_1\ !\ put(Kant);\ become(\beta_K^{(think)});$
$e_{think};\ Kant\ !\ eat\}$

$\longrightarrow^{SEND}\ [](\beta_K^{(denied)})Kant\{become(\beta_K^{(think)});\ e_{think};\ Kant\ !\ eat\}$

$\longrightarrow^{BECOME}\ [](\beta_K^{(think)})Kant\{e_{think};\ Kant\ !\ eat\}$

$\longrightarrow^{\cdots}\ [](\beta_K^{(think)})Kant\{Kant\ !\ eat\}$

$\longrightarrow^{SEND}\ [Kant \mapsto eat](\beta_K^{(think)})Kant\{\underline{0}\}$

e ricomincia il ciclo.

Caso 4: la prima bacchetta a rispondere è C_2 che è occupata. Poi risponde C_1, anch'essa occupata.

$[Kant \mapsto busy(C_2) \cdot busy(C_1)](\beta_K^{(hungry)})Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ [Kant \mapsto busy(C_1](\beta_K^{(hungry)})Kant\{become(\beta_K^{(denied)})\};$

$\longrightarrow^{BECOME}\ [Kant \mapsto busy(C_1](\beta_K^{(denied)})Kant\{\underline{0}\}$

$\longrightarrow^{RECEIVE}\ [](\beta_K^{(denied)})Kant\{become(\beta_K^{(think)});\ e_{think};\ Kant\ !\ eat\}$

$\longrightarrow^{BECOME}\ [](\beta_K^{(think)})Kant\{e_{think};\ Kant\ !\ eat\}$

$\longrightarrow^{\cdots}\ [](\beta_K^{(think)})Kant\{Kant\ !\ eat\}$

$\longrightarrow^{SEND}\ [Kant \mapsto eat](\beta_K^{(think)})Kant\{\underline{0}\}$

e ricomincia il ciclo.

Anche in questi ultimi due casi, se una delle due forchette dovesse tardare a rispondere, il filosofo si sospenderebbe in attesa.

Codice Scala Akka (riadattato da http://www.typesafe.com/activator/template/akka-sample-fsm-scala# code/src/main/scala/sample/become/DiningHakkersOnBecome.scala)

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5  import scala.concurrent.duration._
6
7  object DiningPhilosophers extends App{
8
9    final case class Busy(chopstick: ActorRef)
10   final case class Put(philosopher: ActorRef)
11   final case class Take(philosopher: ActorRef)
12   final case class Taken(chopstick: ActorRef)
13   final case class Eat()
14   final case class Think()
15
16   val s = ActorSystem()
17
18   /*Corpo delle bacchette*/
19   class Chopstick extends Actor{
20     import context._
21
22     def available: Receive = {
23       case Take(philosopher) =>
24         become(takenBy(philosopher))
25         println("%s taken by %s "
26             .format(self.path.name, philosopher.path.name))
27         philosopher ! Taken(self)
28     }
29
30     def takenBy(philosopher: ActorRef): Receive = {
31       case Take(otherPhilosopher) =>
32         otherPhilosopher ! Busy(self)
33       case Put(philosopher) =>
34         become(available)
35         println("%s released by %s "
36             .format(self.path.name, philosopher.path.name))
37     }
38
39     def receive = available
40   }
41
```

```
42  /*Corpo dei filosofi*/
43  class Philosopher(name: String, left: ActorRef, right: ActorRef)
44    extends Actor{
45
46    import context._
47
48    self ! Think
49
50    def thinking: Receive = {
51      case Eat =>
52        become(hungry)
53        left ! Take(self)
54        right ! Take(self)
55    }
56
57    def hungry: Receive = {
58      case Taken(`left`) =>
59        become(waiting_for(right, left))
60      case Taken(`right`) =>
61        become(waiting_for(left, right))
62      case Busy(chopstick) =>
63        become(denied_a_chopstick)
64        if(name.compareTo("Kant") == 0)
65          println("%s tried to pick up %s and failed"
66              .format(name, chopstick.path.name))
67    }
68
69    def waiting_for(chopstickToWaitFor: ActorRef,
70                otherChopstick: ActorRef): Receive = {
71
72      case Taken(`chopstickToWaitFor`) =>
73        println("%s has picked up %s and %s and starts to eat"
74            .format(name, left.path.name, right.path.name))
75        become(eating)
76        s.scheduler.scheduleOnce(3.seconds, self, Think)
77
78      case Busy(chopstick) =>
79        otherChopstick ! Put(self)
80        if(name.compareTo("Kant") == 0)
81          println("%s has picked up %s but not %s,
82                  put down %s and starts to think"
83              .format(name,
84                  otherChopstick.path.name,
85                  chopstickToWaitFor.path.name,
86                  otherChopstick.path.name))
```

```
87          startThinking(1.seconds)
88        }
89
90        def denied_a_chopstick: Receive = {
91          case Taken(chopstick) =>
92            chopstick ! Put(self)
93            if(name.compareTo("Kant") == 0)
94              println("%s failed picking up the first chopstick
95                       so put down %s and starts to think"
96                  .format(name, chopstick.path.name))
97            startThinking(1.seconds)
98          case Busy(chopstick) =>
99            if(name.compareTo("Kant") == 0)
100             println("%s failed picking up both chopstick")
101
102           startThinking(1.seconds)
103       }
104
105       def eating: Receive = {
106         case Think =>
107           left ! Put(self)
108           right ! Put(self)
109           println("%s puts down his chopsticks and starts to think"
110               .format(name))
111           startThinking(5.seconds)
112       }
113
114       def receive = {
115         case Think =>
116           println("%s starts to think".format(name))
117           startThinking(5.seconds)
118       }
119
120       private def startThinking(duration: FiniteDuration): Unit = {
121         become(thinking)
122         s.scheduler.scheduleOnce(duration, self, Eat)
123       }
124     }
125
126     val chopstick1 = s.actorOf(Props(new Chopstick), "C1")
127     val chopstick2 = s.actorOf(Props(new Chopstick), "C2")
128     val chopstick3 = s.actorOf(Props(new Chopstick), "C3")
129
130     val Kant = s.actorOf(Props(
131         new Philosopher("Kant", chopstick1, chopstick2)), "Kant")
```

```
132   val Hegel = s.actorOf(Props(
133       new Philosopher("Hegel", chopstick2, chopstick3)), "Hegel")
134   val Marx = s.actorOf(Props(
135       new Philosopher("Marx", chopstick3, chopstick1)), "Marx")
136
137 }
```

Facciamo alcune considerazioni sul codice. La prima cosa da notare è che la corrispondenza tra i comportamenti β del programma scritto in linguaggio A e i comportamenti (ovvero gli oggetti di tipo `Receive`) nel codice Scala Akka, è totale a meno delle stampe che servono nel codice Scala per tenere traccia dell'esecuzione. In particolare con le stampe tracciamo lo stato delle forchette (taken o released), i filosofi che iniziano a mangiare e il momento in cui ripongono le forchette al termine, e i fallimenti solo per il filosofo $Kant$, ovvero tutti i casi in cui $Kant$ riesce ad acquisire solo una delle due forchette o nessuna delle due.

Un'altra considerazione riguarda i confronti $if(x = a)$, che nel codice Scala non sono presenti. Ad esempio, nel comportamento

```
waiting_for(chopstickToWaitFor, otherChopstick)
```

il valore `chopstickToWaitFor` è già compreso nel pattern

```
Taken(‘chopstickToWaitFor‘) => ..
```

senza bisogno di un'espressione

```
Taken(x) => if (x == ‘chopstickToWaitFor‘) then ...
```

come abbiamo inserito nell'esempio scritto in linguaggio A. Questo perchè il pattern matching di Scala supporta case statements i cui parametri sono valori, mentre nel nostro linguaggio abbiamo definito pattern nella forma $m(\tilde{x})$ dove $\tilde{x}$ sono esclusivamente variabili. Per questo nel nostro linguaggio si rende necessario aggiungere il confronto $if(x = chopstickToWaitFor)$.

Notiamo anche che, mentre nel programma scritto in linguaggio A ogni filosofo ha associato un set di comportamenti, che contengono già i nomi delle bacchette a lui associate, nel programma Scala è stato scritto un unico set di comportamenti per i filosofi, che sono però parametrizzati e hanno quindi il riferimento al filosofo stesso e alle sue due bacchette.

Come ultima cosa notiamo che il metodo privato `startThinking` codifica quello che in A abbiamo rappresentato con l'espressione
$become(\beta_j^{(think)}); e_{think}; j \,!\, eat$ per un generico filosofo $j \in \{Kant, Hegel, Marx\}$.

Output 1

```
1 Kant starts to think
2 Hegel starts to think
3 Marx starts to think
```

```
C1 taken by Kant
C2 taken by Kant
C3 taken by Marx
C3 released by Marx
Kant has picked up C1 and C2 and starts to eat
C3 taken by Marx
C3 released by Marx
C3 taken by Marx
C3 released by Marx
C2 released by Kant
C1 released by Kant
Kant puts down his chopsticks and starts to think
C1 taken by Marx
C2 taken by Hegel
C3 taken by Marx
Marx has picked up C3 and C1 and starts to eat
C2 released by Hegel
C2 taken by Hegel
C2 released by Hegel
C2 taken by Hegel
C2 released by Hegel
Marx puts down his chopsticks and starts to think
C3 released by Marx
C1 released by Marx
C2 taken by Hegel
C3 taken by Hegel
Hegel has picked up C2 and C3 and starts to eat
Kant tried to pick up C2 and failed
C1 taken by Kant
C1 released by Kant
Kant failed picking up the first so put down C1 and starts to think
Kant tried to pick up C2 and failed
C1 taken by Kant
C1 released by Kant
Kant failed picking up the first so put down C1 and starts to think
Hegel puts down his chopsticks and starts to think
C3 released by Hegel
C2 released by Hegel
C2 taken by Kant
C1 taken by Kant
Kant has picked up C1 and C2 and starts to eat
C3 taken by Marx
C3 released by Marx
C3 taken by Marx
C3 released by Marx
```

```
49 Kant puts down his chopsticks and starts to think
50 C2 released by Kant
51 C1 released by Kant
52 C1 taken by Marx
53 C3 taken by Marx
54 Marx has picked up C3 and C1 and starts to eat
```

Focalizziamo l'attenzione sul comportamento di *Kant*. Alle righe 4 e 5 *Kant* riesce ad acquisire entrambe le bacchette. Ci ritroviamo nel caso 1, in cui il filosofo ha in mailbox due messaggi *taken*. Ci aspettiamo quindi la seguente sequenza:

$\hookrightarrow RECEIVE\ taken(C_1) \hookrightarrow BECOME\ wait(C_2, C_1) \hookrightarrow RECEIVE\ taken(C_2)$

$\hookrightarrow BECOME\ eat \hookrightarrow SEND\ Kant\ !\ think \hookrightarrow RECEIVE\ think \hookrightarrow SEND\ C_1\ !\ put$

$\hookrightarrow SEND\ C_2\ !\ put \hookrightarrow BECOME\ think \hookrightarrow SEND\ Kant\ !\ eat$

Alla riga 8 *Kant* inizia a mangiare, quindi analizzando il codice possiamo effettivamente vedere che ha eseguito i primi tre passaggi. Che l'attore cambi poi comportamento in $\beta^{(eat)}$ e abbia ricevuto un messaggio di tipo *think* lo vediamo dalla stampa alla riga 15, quando ripone le bacchette e torna a pensare. Alla riga 31 *Kant* prova ad acquisire una bacchetta, questo significa che ha effettivamente ricevuto un messaggio *eat* mentre si trovava ad avere un comportamento $\beta^{(think)}$ e quindi il comportamento del caso 1 descritto precedentemente è effettivamente un comportamento possibile in Akka.

Dalla riga 31 alla 34 vediamo l'esecuzione dal caso 3: la prima bacchetta è impegnata, mentre la seconda è libera e viene presa. In accordo con il sistema di regole presentato, ci aspettiamo che il filosofo assuma il comportamento $\beta^{(denied)}$, riceva poi un messaggio *taken* per la seconda bacchetta e riponga la bacchetta ($SEND\ C_1\ !\ put$) prima di tornare nuovamente a pensare. Analizzando il codice e l'output nelle righe indicate, vediamo che è esattamente quello che succede.

Output 2

```
1 Marx starts to think
2 Hegel starts to think
3 Kant starts to think
4 C2 taken by Hegel
5 C3 taken by Marx
```

```
6  C1 taken by Kant
7  C2 released by Hegel
8  C1 released by Kant
9  Kant has picked up C1 but not C2, put down C1 and starts to think
10 C3 released by Marx
11 C1 taken by Kant
12 C3 taken by Hegel
13 C2 taken by Hegel
14 Hegel has picked up C2 and C3 and starts to eat
15 C1 released by Kant
16 Kant has picked up C1 but not C2, put down C1 and starts to think
17 C1 taken by Kant
18 Kant tried to pick up C2 and failed
19 Kant failed picking up the first so put down C1 and starts to think
20 C1 released by Kant
21 C1 taken by Kant
```

Alla riga 6 $Kant$ acquisisce C_1 e, secondo le regole, passa ad avere comportamento $\beta^{(wait)}$, adatto ad acquisire la seconda bacchetta C_2. Questa era stata però acquisita da $Hegel$ alla riga 4. Ci aspettiamo quindi che $Kant$ riceva un messaggio di tipo *busy* che, se l'attore ha il comportamento corretto, gli permette di riporre la bacchetta e rimettersi a pensare. Alle righe 8-9 possiamo verificare che è esattamente così.

Osserviamo anche che questo codice é programmato per non avere deadlock. Alle righe 4-5-6, infatti, ognuno prende una forchetta ma non la seconda, ma questo non provoca deadlock in quanto la forchetta acquisita viene riposta dal filosofo che non riesce ad acquisire anche la seconda.

Capitolo 3

Il linguaggio formale A+

3.1 Sintassi e Semantica operazionale

Estendiamo il linguaggio formale A in modo da mettere in evidenza la gerarchia tra gli attori e quindi poter modellare il comportamento di stop. Ricordiamo che un attore può essere creato in Scala Akka attraverso il metodo `actorOf` applicato al sistema di attori o al contesto. Noi modelliamo un sistema con un solo ActorSystem. Questo non è restrittivo ed è inoltre quello che effettivamente va fatto per programmi non distribuiti. Modifichiamo la sintassi in modo da distinguere il tipo di attore che viene creato (i.e. top-level se creato dal sistema, figlio se viene creato dall'oggetto `context` dell'attore) e il meccanismo di terminazione:

Espressioni

$e ::= \underline{0} \mid u!m(\tilde{u}); e \mid val\ a = \gamma.Actor\{\beta; e\}; e \mid become(\beta); e \mid stop(a); e$
$\beta ::= \underline{0} \mid \{m_i(\tilde{x_i}) \Rightarrow e_i\}_{i \in I}$
$\gamma ::= \top \mid \bot$

dove $val\ a = \gamma.Actor\{\beta; e\}; e$ è una generica espressione di creazione di attore, con γ che assume il valore $\top$ per creare un attore top-level e $\bot$ per creare un attore dal contesto corrente (attore figlio).
L'espressione $stop(a); e$ permette di arrestare l'attore a dopo che questo ha finito di processare il messaggio corrente, in modo analogo al metodo `stop` di Akka. Ricordiamo che il metodo `stop` di Akka può essere invocato sull'oggetto `context` dell'attore o sul sistema e prende come parametro il riferimento (`ActorRef`) dell'attore da terminare. Scegliamo di arrestare gli attori solamente dal corpo di un altro attore. Questo non è limitante in quanto tutti gli attori hanno comunque la possibilità di essere arrestati.

Configurazioni

$F ::= \underline{0} \mid [a \to M]\ \mathcal{C}\ a\{e\} \mid F|F \mid (\nu a)F \mid d \mid [a \mapsto \emptyset]$
$d ::= \underline{0} \mid val\ a = \top.Actor\{\beta; e\}; d$
$M ::= \emptyset \mid m(\tilde{a}) \cdot M$
$\mathcal{C} ::= (\beta \uparrow \tilde{p} \downarrow \tilde{c})$

La rappresentazione runtime di un attore attivo si arricchisce di due elementi: il riferimento al padre e il riferimento ai figli, che vengono inseriti nel contesto $\mathcal{C}$ insieme al comportamento. Il riferimento al padre viene indicato con $\uparrow \tilde{p}$, dove $\tilde{p}$ è una tupla di lunghezza 0 o 1 di nomi di attori. Se l'attore è un top level actor la tupla del padre è la tupla vuota, se invece l'attore è stato creato come figlio di un attore p, la tupla contiene il solo elemento p. Analogamente il riferimento ai figli per un attore a è una tupla (possibilmente vuota) di nomi di attori che sono stati generati come figli di a.
Le configurazioni si arricchiscono con $[a \mapsto \emptyset]$. La mailbox vuota senza contesto né corpo dell'attore sta a rappresentare un attore terminato, il cui riferimento è ancora conosciuto dal sistema (quindi a cui possono essere spediti messaggi, che però non saranno gestiti).
Osserviamo inoltre che ora l'espressione d contiene solo attori top-level.

Avendo arricchito le configurazioni dobbiamo aggiornare la definizione di nomi liberi ($fn(F)$, vedi paragrafo 1, capitolo 2) con le nuove configurazioni. In particolare:

- $fn(\mathcal{C}) = fn(\beta) \cup \{\tilde{p}\} \cup \{\tilde{c}\}$
- $fn([a \mapsto \emptyset]) = a$

Semantica operazionale

$$\text{(PAR)}\ \frac{F_1 \longrightarrow F_1'}{F_1|F_2 \longrightarrow F_1'|F_2} \qquad \text{(RES)}\ \frac{F \longrightarrow F'}{(\nu a)F \longrightarrow (\nu a)F'} \qquad \text{(STRUCT)}\ \frac{F \equiv F' \quad F' \to F'' \quad F'' \equiv F'''}{F \longrightarrow F'''}$$

(1) $(\nu a)(\nu b)F \equiv (\nu b)(\nu a)F$
(2) $(\nu a)(F|F') \equiv F|(\nu a)F'\ a \notin fn(F)$
(3) $F|\underline{0} \equiv F$
(4) $F|F' \equiv F'|F$
(5) $(F_1|F_2)|F_3 \equiv F_1|(F_2|F_3)$
(6) $(\nu a)[a \mapsto \emptyset] \equiv \underline{0}$

(TOP-SPAWN)

$$val\ a = \top.Actor\{\beta; e\}; d \longrightarrow (\nu a)([a \mapsto \emptyset](\beta \uparrow \emptyset \downarrow \emptyset)a\{e\} \mid d)$$

(SPAWN)

$$\frac{a \notin (fn(\beta) \cup fn(M) \cup \tilde{p} \cup \tilde{c})}{\begin{array}{l}[b \mapsto M](\beta \uparrow \tilde{p} \downarrow \tilde{c})b\{val\ a = \top.Actor\{\beta'; e\}\ ; e'\} \longrightarrow \\ \qquad (\nu a)([b \to M](\beta \uparrow \tilde{p} \downarrow \tilde{c})b\{e'\} \mid [a \mapsto \emptyset](\beta' \uparrow \emptyset \downarrow \emptyset)a\{e\})\end{array}}$$

(SPAWN-CHILD)

$$\frac{a \notin (fn(\beta) \cup fn(M) \cup \tilde{p} \cup \tilde{c})}{\begin{array}{l}[b \mapsto M](\beta \uparrow \tilde{p} \downarrow \tilde{c})b\{val\ a = \bot.Actor\{\beta'; e\}\ ; e'\} \longrightarrow \\ \qquad (\nu a)([b \mapsto M](\beta \uparrow \tilde{p} \downarrow (\tilde{c}, a))b\{e'\} \mid [a \mapsto \emptyset](\beta' \uparrow b \downarrow \emptyset)a\{e\})\end{array}}$$

(BECOME)

$$[a \mapsto M](\beta \uparrow \tilde{p} \downarrow \tilde{c})a\{become(\beta');\ e\} \longrightarrow [a \mapsto M](\beta' \uparrow \tilde{p} \downarrow \tilde{c})a\{e\}$$

(SEND)

$$[a \mapsto M]\ \mathcal{C}\ a\{e\} \mid [b \mapsto M']\ \mathcal{C}'\ b\{a!m(\tilde{c}); e'\} \longrightarrow [a \mapsto M \cdot m(\tilde{c})]\ \mathcal{C}\ a\{e\} \mid [b \mapsto M']\ \mathcal{C}'\ b\{e'\}$$

(SEND-DEAD)

$$[a \mapsto \emptyset] \mid [b \mapsto M']\ \mathcal{C}'\ b\{a!m(\tilde{c}); e'\} \longrightarrow [a \mapsto \emptyset] \mid [b \mapsto M']\ \mathcal{C}'\ b\{e'\}$$

(RECEIVE)

$$\frac{}{\begin{array}{l}[a \mapsto m_j(\tilde{b}) \cdot M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I} \uparrow \tilde{p} \downarrow \tilde{c})a\{\underline{0}\} \longrightarrow \\ \qquad [a \mapsto M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I} \uparrow \tilde{p} \downarrow \tilde{c})a\{e_j\{\tilde{b}/\tilde{x_j}\}\}\end{array}} \quad j \in I$$

(JUNK)

$$\frac{}{\begin{array}{l}[a \mapsto m_j(\tilde{b}) \cdot M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I} \uparrow \tilde{p} \downarrow \tilde{c})a\{\underline{0}\} \longrightarrow \\ \qquad [a \mapsto M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i \in I} \uparrow \tilde{p} \downarrow \tilde{c})a\{\underline{0}\}\end{array}} \quad j \notin I$$

(STOP)

$$[a \mapsto M]\ \mathcal{C}\ a\{stop(b); e\} \mid [b \mapsto M']\ \mathcal{C}'\ b\{e'\} \longrightarrow [a \mapsto M]\ \mathcal{C}\ a\{e\} \mid [b \mapsto M']\ \mathcal{C}'\ b\{e'; stop(b)\}$$

(STOP-SELF)

$$[a \mapsto M](\beta \uparrow \tilde{p} \downarrow (c_i)_{i \in I})a\{stop(a); e\} \longrightarrow [a \mapsto M](\underline{0} \uparrow \tilde{p} \downarrow (c_i)_{i \in I})a\{e; (stop(c_i))_{i \in I}\}$$

(TOP-END)

$$[a \mapsto \emptyset](\underline{0} \uparrow \emptyset \downarrow \emptyset)a\{\underline{0}\} \longrightarrow [a \mapsto \emptyset]$$

(END)

$$[a \mapsto \emptyset](\underline{0} \uparrow p \downarrow \emptyset)a\{\underline{0}\} \mid [p \mapsto M](\beta \uparrow p' \downarrow (\tilde{c}))p\{e\} \longrightarrow [a \mapsto \emptyset] \mid [p \mapsto M](\beta \uparrow p' \downarrow (\tilde{c} \setminus \{a\}))p\{e\} \quad a \in \tilde{c}$$

Un attore può essere creato top-level attraverso la regola (TOP-SPAWN). Viene generato un nuovo attore in parallelo che ha come riferimento al padre la tupla vuota (i.e. attore user). Dall'interno di un attore, un altro attore può essere generato come figlio (SPAWN-CHILD), o come attore top-level (SPAWN).

La regola (SEND-DEAD) mostra cosa succede se un attore b invia un messaggio ad un attore terminato a. Nel nostro modello il messaggio viene semplicemente scartato. In Scala Akka, questo messaggio viene incapsulato in un oggetto `DeadLetter` e inviato all'`EventStream` che traccia l'evento. Dal momento che di default un messaggio inviato ad un attore terminato in Akka genera solamente una riga di log, nel nostro linguaggio scegliamo di scartarlo completamente. Questo rispetta la politica Akka "let it crash".

Un attore a può arrestare un altro attore di cui conosca il nome (regola (STOP)). L'attore su cui è stato chiamato lo stop finisce di eseguire il corpo corrente e infine richiama una *stop* su se stesso (STOP-SELF).

Un attore a che esegue una *stop* su se stesso si predispone a non ricevere ulteriori messaggi (il comportamento diventa $\underline{0}$), finisce di processare il rimanente corpo e (vedi esempio 3.2.1) e infine invia un comando di *stop* ai figli con la sequenza $stop(c_i)_{i \in I}$. Se l'attore non ha figli, ovvero $I = \emptyset$, l'espressione $stop(c_i)_{i \in I}$ equivale a $\underline{0}$. La regola (STOP-SELF) modella il costrutto `context.stop(self)` di Scala Akka, in cui un attore finisce di processare il messaggio corrente e scarta i messaggi rimasti in mailbox. Notiamo che, dopo aver effettuato una *stop* su se stesso, l'attore a rimane comunque attivo. Questo perché prima di terminare deve attendere la terminazione di tutti i figli (avere $I = \emptyset$) e scaricare la mailbox. Nel linguaggio A+ il procedimento di svuotamento della mailbox avviene applicando la regola (JUNK) dopo che l'attore ha cambiato il proprio comportamento in $\underline{0}$.

La regola (TOP-END) modella l'effettiva terminazione di un attore top-level. Quando l'attore non può ricevere messaggi (comportamento $\underline{0}$), ha svuotato la mailbox dai rimanenti, ha finito di processare il corpo e non ha più figli attivi,

può terminare. Questo genera la configurazione $[a \mapsto \emptyset]$, che sta a ricordare che anche se l'attore è terminato il suo nome è ancora conosciuto dal sistema e quindi, potenzialmente, possono essergli inviati messaggi (regola (SEND-DEAD)). Un attore non top-level che termina (regola (END)) deve avvisare il padre della propria terminazione. Formalmente questo avviene togliendo l'attore in questione dalla lista dei figli del padre.

3.2 Esempi

3.2.1 Terminazione

Consideriamo il seguente scenario: un attore *parent* genera un figlio *child* e gli assegna qualche compito. *child* inizia ad eseguire ciò che gli è stato comandato di fare e nel frattempo un attore a manda un messaggio al genitore che ne causa la terminazione. Chiamiamo gli attori *parent* e *child* rispettivamente p e c per semplicità di lettura e consideriamo la seguente configurazione:

$$[p \mapsto \emptyset](\beta \uparrow \emptyset \downarrow c)p\{\underline{0}\} \mid [c \mapsto \emptyset](\beta' \uparrow p \downarrow \emptyset)c\{e\}) \mid [a \mapsto \emptyset] \Subset a\{p!m;\}$$

$$\beta = \{m \Rightarrow stop(p); c \,!\, m'; e_p\}$$
$$\beta' = \{m' \Rightarrow e_c\}$$

Facciamo alcune considerazioni sulla configurazione da cui partiamo per l'analisi. Abbiamo un attore p con un figlio attivo c che sta eseguendo un corpo e. Abbiamo poi in parallelo un attore a che invia a p un messaggio m. Il comportamento β di p prevede che, alla ricezione di un messaggio m, l'attore esegua un comando *stop* su se stesso e successivamente invii un messaggio al figlio e prosegua con altre istruzioni e_p. Questo non ha molto senso, ma ci è utile per illustrare la nostra semantica. Notiamo inoltre che, quando c riceve m', inizia l'esecuzione di una serie di istruzioni e_c.

Vediamo ora come evolve runtime la configurazione illustrata:

$$\longrightarrow^{SEND} (\nu p)((\nu c)(\underline{[p \mapsto m](\beta \uparrow \emptyset \downarrow c)p\{\underline{0}\}} \mid (\beta' \uparrow p \downarrow \emptyset)c\{e_c\}) \mid (\nu a)([a \mapsto \emptyset] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{RECEIVE} (\nu p)((\nu c)(\underline{[](\beta \uparrow \emptyset \downarrow c)p\{stop(p); c!m'; e_p\}} \mid [](\beta' \uparrow p \downarrow \emptyset)c\{e\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{STOP-SELF} (\nu p)((\nu c)(\underline{[](\underline{0} \uparrow \emptyset \downarrow c)p\{c!m'; e_p; stop(c)\}} \mid [](\beta' \uparrow p \downarrow \emptyset)c\{e\}) \mid (\nu a)([] \Subset a\{\underline{0}\})$$

$$\longrightarrow^{SEND} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{e_p; stop(c)\} \mid [c \mapsto m'](\beta' \uparrow p \downarrow \emptyset)c\{e_c\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

Partendo da questa configurazione, che chiameremo F, mostriamo due possibili andamenti dell'esecuzione:

Caso 1

Supponiamo di trovarci nella configurazione F e ipotizziamo che l'esecuzione passi a c. c è ancora ignaro che il padre stia terminando, poiché verrà avvertito da p solo al termine dell'esecuzione di e_p. Può quindi ricevere il messaggio in mailbox e processarlo:

$$\longrightarrow^{\cdots} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{e_p; stop(c)\} \mid \underline{[c \mapsto m'](\beta' \uparrow p \downarrow \emptyset)c\{\underline{0}\}}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{RECEIVE} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{e_p; stop(c)\} \mid \underline{[](\beta' \uparrow p \downarrow \emptyset)c\{e_c\}}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{\cdots} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{e_p; stop(c)\} \mid [](\beta' \uparrow p \downarrow \emptyset)c\{\underline{0}\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

A questo punto l'unico attore della configurazione che può evolvere è p, che finirà di eseguire il corpo e_p e infine invierà un comando *stop* al figlio.

$$\longrightarrow^{\cdots} (\nu p)((\nu c)(\underline{[](\underline{0} \uparrow \emptyset \downarrow c)p\{stop(c)\}} \mid [](\beta' \uparrow p \downarrow \emptyset)c\{\underline{0}\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{STOP} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{\underline{0}\} \mid \underline{[](\beta' \uparrow p \downarrow \emptyset)c\{stop(c)\}}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

c nell'ultima configurazione non stava eseguendo nulla quindi riceve immediatamente il comando di *stop* dal padre ed esegue una stop su se stesso. c non ha figli e non ha messaggi in mailbox, quindi setta semplicemente il proprio comportamento a $\underline{0}$ e termina avvisando il padre.

$$\longrightarrow^{STOP-SELF} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{\underline{0}\} \mid \underline{[](\underline{0} \uparrow p \downarrow \emptyset)c\{\underline{0}\}}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{END} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow \emptyset)p\{\underline{0}\} \mid [c \mapsto \emptyset]) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

A questo punto p non ha più figli attivi, né corpo, né messaggi in mailbox da scartare, e ha comportamento vuoto, può quindi terminare:

$$\longrightarrow^{TOP-END} (\nu p)((\nu c)[p \mapsto \emptyset] \mid [c \mapsto \emptyset]) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

Dal momento che i nomi p e c non appartengono ai nomi liberi di $[] \Subset a\{\underline{0}\}$, Questa configurazione è

$$\equiv (\nu a)([] \Subset a\{\underline{0}\})$$

Scriviamo ora questo programma in Scala Akka e verifichiamo che i passaggi mostrati con le riduzioni nel linguaggio A+ si verificano effettivamente anche in Akka. In particolare, per ottenere l'interleaving descritto e quindi poterlo verificare, inseriamo alcuni `Thread.sleep` in modo da far prendere la strada desiderata all'esecuzione.

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5
6  object Stop extends App{
7
8    val s = ActorSystem()
9
10   val parent = s.actorOf(Props(new Actor{
11     val child = context.actorOf(Props(new Actor{
12         def receive = {
13           case "do" => println("child: ... doing something ...")
14                        Thread.sleep(6000)
15                        println("child: done")}
16         override def postStop() = {println("child: terminating myself")}
17     }));
18
19     child ! "do"
20
21     def receive = {
22        case "m" => println("parent: stopping myself")
23                    context.stop(self)
24                    println("parent: sending message to child")
25                    child ! "do"
26                    println("parent: waiting a little")
27                    Thread.sleep(12000)
28                    println("parent: finish")
29     }
30     override def postStop() = {println("parent: terminating myself")}
31
32   }))
33
34   val a = s.actorOf(Props(new Actor{
35      Thread.sleep(3000)
36      parent!"m"
37
38      def receive = {PartialFunction.empty}
```

```
39      }))
40  }
```

Notiamo che nel codice Scala Akka, per tenere traccia della terminazione degli attori, utilizziamo il metodo `postStop`, che viene invocato quando un'istanza di attore termina.

Output

```
1  child: ... doing something ...
2  parent: stopping myself
3  parent: sending message to child
4  parent: waiting a little
5  child: done
6  child: ... doing something ...
7  child: done
8  parent: finish
9  child: terminating myself
10 parent: terminating myself
```

Analizziamo l'output. La situazione è quella descritta precedentemente: *child* sta eseguendo il proprio corpo (e nel corrispondente programma A+), nel frattempo *parent* esegue il comando `context.stop(self)`. In accordo con la regola (STOP-SELF) *parent* finisce di processare il messaggio corrente, quindi invia un messaggio a *child* e poi continua ad eseguire mettendosi in sleep per un po' (corrispondente all'espressione e_p del programma A+). A questo punto l'esecuzione passa a *child* che, dal momento che *parent* non ha ancora finito di eseguire il suo corpo, non sa ancora che *parent* sta terminando, prende dalla mailbox un altro messaggio e lo processa.

Quando *parent* termina la sleep, invia il comando di `stop` a *child* che può procedere alla sua terminazione. Solo quando *child* è terminato anche *parent* termina.

L'output del programma Scala Akka conferma quanto mostrato dalla riduzione del programma A+: l'attore *parent*, dopo aver eseguito un comando `stop` su se stesso, finisce di processare il messaggio corrente prima di avviare effettivamente la propria terminazione, ovvero inviare ai figli un comando di `stop`. Questo è dimostrato dal fatto che *child* riesce a processare anche il secondo messaggio (inviatogli da *parent* dopo che questo si è stoppato) e spiega perché è corretto usare $e; (stop(c_i))_{i \in I}$ nella regola (STOP-SELF). Inoltre l'output conferma che *parent* non termina prima che tutti i suoi figli (in questo caso il solo figlio *child*) siano terminati.

Caso 2

Supponiamo di trovarci nella configurazione F sopra descritta e supponiamo questa volta che p finisca di processare tutto il suo corpo prima che c termini l'esecuzione di e_c. L'evoluzione del programma è la seguente:

$$(\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{e_p; stop(c)\} \mid [c \mapsto m'](\beta' \uparrow p \downarrow \emptyset)c\{e_c\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{\cdots} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{stop(c)\} \mid [c \mapsto m'](\beta' \uparrow p \downarrow \emptyset)c\{e_c\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{STOP} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{\underline{0}\} \mid [c \mapsto m'](\beta' \uparrow p \downarrow \emptyset)c\{e_c; stop(c)\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

A questo punto c è l'unico attore che può proseguire perché sia a che p hanno corpo nullo e mailbox vuota.

$$\longrightarrow^{\cdots} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{\underline{0}\} \mid [c \mapsto m'](\beta' \uparrow p \downarrow \emptyset)c\{stop(c)\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{STOP-SELF} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{\underline{0}\} \mid [c \mapsto m'](\underline{0} \uparrow p \downarrow \emptyset)c\{\underline{0}\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

Ora c non può ancora terminare perché ha dei messaggi in mailbox. Può però applicare la regola (JUNK) per svuotare la mailbox e quindi terminare.

$$\longrightarrow^{JUNK} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow c)p\{\underline{0}\} \mid [c \mapsto \emptyset](\underline{0} \uparrow p \downarrow \emptyset)c\{\underline{0}\}) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\longrightarrow^{END} (\nu p)((\nu c)([](\underline{0} \uparrow \emptyset \downarrow \emptyset)p\{\underline{0}\} \mid [c \mapsto \emptyset]) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

A questo punto p non ha più figli attivi e può terminare:

$$\longrightarrow^{TOP-END} (\nu p)((\nu c)([p \mapsto \emptyset] \mid [c \mapsto \emptyset]) \mid (\nu a)([] \Subset a\{\underline{0}\}))$$

$$\equiv (\nu a)([] \Subset a\{\underline{0}\})$$

Codice Scala Akka

Per provare che il comportamento descritto è quello del programma in Scala Akka apportiamo le seguenti modifiche al codice, in modo da far prendere la strada desiderata all'esecuzione. La funzione `receive` di *child* diventa:

```
1 def receive = {
2     case "do" => println("child: ... doing something ...")
3           Thread.sleep(12000)
4           println("child: done")
5 }
```

mentre quella di *parent*:

```
def receive = {
      case "m" => println("parent: stopping myself")
                  context.stop(self)
                  println("parent: sending message to child")
                  child ! "do"
    }
```

Output

```
child: ... doing something ...
parent: stopping myself
parent: sending message to child
child: done
child: terminating myself
parent: terminating myself
```

Dall'output possiamo vedere che *parent* inizia il processo di terminazione mentre *child* sta processando un messaggio. Come abbiamo detto questo comporta l'invio di un segnale di *stop* al figlio. Successivamente *parent* invia al figlio un messaggio. Quando *child* finisce di processare il messaggio corrente ha in mailbox il messaggio inviato dal padre, ma lo scarta e termina, in analogia con la riduzione descritta per il programma in linguaggio $A+$.

3.2.2 Alice Bob e Carl ping pong con goodbye

Estendiamo l'esempio del ping pong con il meccanismo degli stop. *Alice* inizia sempre due sessioni di gioco, *Carl* e *Bob* creano la loro e il gioco si svolge normalmente. Quando la sessione di gioco di *Bob* e *Carl* riceve un *pang*, questa invia un saluto ad *Alice* e arresta il genitore. Quando *Alice* ha ricevuto entrambi i messaggi di saluto chiude le sue sessioni e si arresta.

Il codice del programma rimane lo stesso dell'esempio 2.3.1 visto al capitolo 2 eccetto alcune modifiche: perché le sessione di *Bob* e *Carl* possano inviare un messaggio *goodbye* ad *Alice* devono conoscerne il riferimento. Facciamo quindi in modo che *Alice*, quando invia un messaggio *new* a uno dei due attori, supponiamo *Carl*, con parametro la sua sessione ac, gli invii anche il riferimento a se stessa. Il risultato sarà un messaggio $new(ac, Alice)$. In modo analogo ci sarà un messaggio $new(ab, Alice)$ per *Bob*. Inoltre cambia il comportamento iniziale di *Alice* e i comportamenti di ba e ca alla ricezione di un messaggio di tipo *pang*:

$\beta_A = \{goodbye() \Rightarrow become(\beta'_A)\}$

$\beta'_A = \{goodbye() \Rightarrow stop(Alice)\}$
$\beta_B = \{new(z,w) \Rightarrow val\ ba = Actor\{\beta_{ba}; \underline{0}\}; z!dest(ba)\}$
$\beta_{ab} = \{dest(y) \Rightarrow y!ping(); become\{\beta'_{ab}\}\}$
$\beta_{ba} = \{ping() \Rightarrow z!pong(); become\{\beta'_{ba}\}\}$
$\beta'_{ab} = \{pong() \Rightarrow y!pang()\}$
$\beta'_{ba} = \{pang() \Rightarrow w\ !\ goodbye(); stop(Bob);\}$

β_C, β_{ac}, β_{ca}, β'_{ac} e β'_{ca}, usati da $Carl$, si ottengono dai corrispettivi di Bob.

Con la nuova sintassi abbiamo introdotto due nuovi elementi nel contesto $\mathcal{C}$ dell'attore: il riferimento al padre e ai figli. Vediamo quindi la configurazione, una volta che tutti gli attori sono stati avviati, in linguaggio A+:

$(\nu ab)(\nu Bob)((\nu ba)([Bob \mapsto ...](\beta_B \uparrow \emptyset \downarrow ba)Bob\{...\} \mid [ba \mapsto ...](\beta_{ba} \uparrow Bob \downarrow \emptyset)ba\{...\}) \mid$
$(\nu ac)(\nu Carl)((\nu ca)[Carl \mapsto ...](\beta_C \uparrow \emptyset \downarrow ca)Carl\{...\} \mid [ca \mapsto ...](\beta_{ca} \uparrow Carl \downarrow \emptyset)ca\{...\}) \mid$
$(\nu Alice)[Alice \mapsto ...](\beta_A \uparrow \emptyset \downarrow (ab, ac))Alice\{...\} \mid [ab \mapsto ...](\beta_{ab} \uparrow Alice \downarrow \emptyset)ab\{...\} \mid [ac \mapsto ...](\beta_{ac} \uparrow Alice \downarrow \emptyset)ac\{...\}))$

Prendiamo come esempio $Alice$ e Bob e le loro sessioni di gioco ab e ba. Anche il ping pong tra $Alice$ e $Carl$ si svolgerà in modo analogo. Supponiamo quindi che il gioco con Bob sia il primo a concludersi. Dalla configurazione con gli attori avviati mostrata sopra fino all'arrivo del messaggio $pang$ l'esecuzione è la stessa descritta dall'esempio 2.3.1. Analizziamo quindi l'evoluzione del programma (limitandoci sempre al gioco tra $Alice$ e Bob) partendo da una situazione in cui la mailbox di ba contiene il messaggio $pang$. Chiamiamo F la configurazione di attori in parallelo che non sono tracciati esplicitamente. Usiamo inoltre $\mathcal{C}$ per indicare il contesto di $Alice$ e la notazione $F_1 \hookrightarrow F_2$ già utilizzata in precedenza per indicare che la riduzione da F_1 a F_2 può essere intervallata da operazioni di altri attori che eseguono in parallelo (nel nostro caso $Carl$ e le sessioni ab, ac, ca).

$(\nu ba)(\nu Bob)(\nu Alice)(\underline{[ba \mapsto pang()](\beta'_{ba} \uparrow B \downarrow \emptyset)ba\{\underline{0}\}} \mid [Bob \mapsto \emptyset](\beta_B \uparrow \emptyset \downarrow (ba))Bob\{\underline{0}\} \mid [Alice \mapsto \emptyset](\beta_A)Alice\{\underline{0}\} \mid F)$

$\hookrightarrow^{RECEIVE}$ $(\nu ba)(\nu Bob)(\nu Alice)(\underline{[ba \mapsto \emptyset](\beta'_{ba} \uparrow B \downarrow \emptyset)ba\{Alice\ !\ goodbye(); stop(Bob)\}} \mid [Bob \mapsto \emptyset](\beta_B \uparrow \emptyset \downarrow (ba))Bob\{\underline{0}\} \mid [Alice \mapsto \emptyset](\beta_A)Alice\{\underline{0}\} \mid F)$

$\hookrightarrow^{SEND}$ $(\nu ba)(\nu Bob)(\nu Alice)([ba \mapsto \emptyset](\beta'_{ba} \uparrow B \downarrow \emptyset)ba\{stop(Bob)\} \mid [Bob \mapsto \emptyset](\beta_B \uparrow \emptyset \downarrow (ba))Bob\{\underline{0}\} \mid [Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$

Mostriamo la terminazione di Bob prima di fare ricevere ad $Alice$ il suo messaggio $goodbye$, questo non è rilevante ai fini dell'esecuzione:

$\hookrightarrow^{STOP} (\nu ba)(\nu Bob)(\nu Alice)([ba \mapsto \emptyset](\beta'_{ba} \uparrow B \downarrow \emptyset)ba\{\underline{0}\} \mid \underline{[Bob \mapsto \emptyset](\beta_B \uparrow \emptyset \downarrow (ba))Bob\{stop(Bob)\}} \mid [Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$

$\hookrightarrow^{STOP-SELF} (\nu ba)(\nu Bob)(\nu Alice)([ba \mapsto \emptyset](\beta'_{ba} \uparrow B \downarrow \emptyset)ba\{\underline{0}\} \mid \underline{[Bob \mapsto \emptyset](\underline{0} \uparrow \emptyset \downarrow (ba))Bob\{stop(ba)\}} \mid [Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$

$\hookrightarrow^{STOP} (\nu ba)(\nu Bob)(\nu Alice)(\underline{[ba \mapsto \emptyset](\beta'_{ba} \uparrow B \downarrow \emptyset)ba\{stop(ba)\}} \mid [Bob \mapsto \emptyset](\underline{0} \uparrow \emptyset \downarrow (ba))Bob\{\underline{0}\} \mid [Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$

$\hookrightarrow^{STOP-SELF} (\nu ba)(\nu Bob)(\nu Alice)(\underline{[ba \mapsto \emptyset](\underline{0} \uparrow B \downarrow \emptyset)ba\{\underline{0}\}} \mid [Bob \mapsto \emptyset](\underline{0} \uparrow \emptyset \downarrow (ba))Bob\{\underline{0}\} \mid [Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$

$\hookrightarrow^{END} (\nu ba)(\nu Bob)(\nu Alice)([ba \mapsto \emptyset] \mid \underline{[Bob \mapsto \emptyset](\underline{0} \uparrow \emptyset \downarrow \emptyset)Bob\{\underline{0}\}} \mid [Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$

$\hookrightarrow^{TOP-END} (\nu ba)(\nu Bob)(\nu Alice)([ba \mapsto \emptyset] \mid [Bob \mapsto \emptyset] \mid [Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$

Dal momento che il gioco tra *Alice* e *Bob* è finito, $ba, Bob \notin fn([Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$. Pertanto la configurazione ottenuta è:

$\equiv (\nu Alice)([Alice \mapsto goodbye()](\beta_A)Alice\{\underline{0}\} \mid F)$

La sessione *ac* di Carl e l'attore *Carl* si comportano in modo analogo al termine del ping pong. Concentriamoci ora sull'attore *Alice*.

Alice ha comportamento iniziale β_A quindi, dopo aver avviato le sessioni, si mette in attesa dei messaggi di saluto. L'ordine di arrivo dei messaggi non è importante, dal momento che all'arrivo del primo messaggio *Alice* modifica il comportamento in modo da poter ricevere un messaggio identico. Possiamo quindi lavorare su uno solo dei casi, ad esempio quello in cui i due messaggi arrivano contemporaneamente. Analizziamo l'esecuzione attesa per *Alice* dopo l'arrivo dei due *goodbye*. Indichiamo anche in questo caso con F i rimanenti attori, che nello specifico sono gli attori *ac* e *ab* che sono gli unici ancora in esecuzione.

$[Alice \mapsto goodbye() \cdot goodbye()](\beta_A \uparrow \emptyset \downarrow (ab, ac))Alice\{\underline{0}\} \mid F$

$\hookrightarrow^{RECEIVE} [Alice \mapsto goodbye()](\beta_A \uparrow \emptyset \downarrow (ab, ac))Alice\{become(\beta'_A)\} \mid F$

$\rightarrow^{BECOME} [Alice \mapsto goodbye()](\beta'_A \uparrow \emptyset \downarrow (ab, ac))Alice\{\underline{0}\} \mid ; F$

$\hookrightarrow^{RECEIVE} [](\beta'_A \uparrow \emptyset \downarrow (ab, ac))Alice\{stop(Alice)\} \mid F$

$\hookrightarrow^{STOP-SELF} [](\underline{0} \uparrow \emptyset \downarrow (ab, ac))Alice\{stop(ab); stop(ac)\} \mid F$

A questo punto *Alice* effettua una stop sui figli che in questo caso sappiamo che non stanno processando nulla (entrambe le sessioni di gioco sono finite poiché sono arrivati due saluti).

$(\nu Alice)(\nu ab)(\nu ac)(\underline{[](\underline{0} \uparrow \emptyset \downarrow (ab, ac))A\{stop(ab); stop(ac)\}} \mid [](\beta_{ab} \uparrow A \downarrow \emptyset)ab\{\underline{0}\} \mid [](\beta_{ac} \uparrow A \downarrow \emptyset)ac\{\underline{0}\})$

$\hookrightarrow^{STOP} (\nu Alice)(\nu ab)(\nu ac)(\underline{[](\underline{0} \uparrow \emptyset \downarrow (ab, ac))A\{stop(ac)\}} \mid [](\beta_{ab} \uparrow A \downarrow \emptyset)ab\{stop(ab)\} \mid [](\beta_{ac} \uparrow A \downarrow \emptyset)ac\{\underline{0}\})$

$\hookrightarrow^{STOP} (\nu Alice)(\nu ab)(\nu ac)([](\underline{0} \uparrow \emptyset \downarrow (ab, ac))A\{\underline{0}\} \mid \underline{[](\beta_{ab} \uparrow A \downarrow \emptyset)ab\{stop(ab)\}} \mid [](\beta_{ac} \uparrow A \downarrow \emptyset)ac\{stop(ac)\})$

$\hookrightarrow^{STOP-SELF} (\nu Alice)(\nu ab)(\nu ac)([](\underline{0} \uparrow \emptyset \downarrow (ab, ac))A\{\underline{0}\} \mid \underline{[](\underline{0} \uparrow A \downarrow \emptyset)ab\{\underline{0}\}} \mid [](\beta_{ac} \uparrow A \downarrow \emptyset)ac\{stop(ac)\})$

$\hookrightarrow^{END} (\nu Alice)(\nu ab)(\nu ac)([](\underline{0} \uparrow \emptyset \downarrow (ac))A\{\underline{0}\} \mid [ab \mapsto \emptyset] \mid \underline{[](\beta_{ac} \uparrow A \downarrow \emptyset)ac\{stop(ac)\}})$

$\hookrightarrow^{STOP-SELF} (\nu Alice)(\nu ab)(\nu ac)([](\underline{0} \uparrow \emptyset \downarrow (ac))A\{\underline{0}\} \mid [ab \mapsto \emptyset] \mid \underline{[](\underline{0} \uparrow A \downarrow \emptyset)ac\{\underline{0}\}})$

$\hookrightarrow^{END} (\nu Alice)(\nu ab)(\nu ac)(\underline{[](\underline{0} \uparrow \emptyset \downarrow \emptyset)A\{\underline{0}\}} \mid [ab \mapsto \emptyset] \mid [ac \mapsto \emptyset])$

$\hookrightarrow^{TOP-END} (\nu Alice)(\nu ab)(\nu ac)([Alice \mapsto \emptyset] \mid [ab \mapsto \emptyset] \mid [ac \mapsto \emptyset]) \equiv \underline{0}$

Runtime ci aspettiamo quindi:

- che *Alice* termini solo dopo aver ricevuto entrambi i saluti e dopo aver terminato le sue sessioni di gioco
- che *Bob* e *Carl* terminino dopo aver terminato le loro sessioni di gioco *ab* e *ac*

Codice Scala Akka

```
1 import akka.actor.Actor
2 import akka.actor.ActorRef
3 import akka.actor.Props
4 import akka.actor.ActorSystem
```

```
5
6
7  object PingPongStop extends App{
8
9
10     case class dest(y:ActorRef)
11     case class new_(xy: ActorRef, who: ActorRef)
12     case class ping()
13     case class pong()
14     case class pang()
15     case class ended()
16     case class goodbye()
17
18     val s = ActorSystem()
19
20     val Bob = s.actorOf(Props(new Actor{
21       def receive = {
22         case new_(session, who) =>
23           val ba : ActorRef = context.actorOf(Props(new Actor{
24             def receive = {
25               case ping() =>
26                 session ! pong(); Thread.sleep(1000)
27                 context.become({case pang() =>
28                     println("ba: RECEIVE pang")
29                     println("ba: sending goodbye to Alice")
30                     who ! goodbye()
31                     println("ba: stopping Bob")
32                     context.stop(context.parent)})
33             }
34             override def postStop = {println("ba: terminating")}
35           }))
36
37           session ! dest(ba)
38
39
40       }
41       override def postStop = {println("Bob: terminating");}
42     }))
43
44     val Carl = s.actorOf(Props(new Actor{
45       def receive = {
46         case new_(session, who) =>
47           val ca : ActorRef = context.actorOf(Props(new Actor{
48             def receive = {
49               case ping() =>
```

```
50                session ! pong();
51                context.become({case pang() =>
52                     println("ca: RECEIVE pang")
53                     println("ca: sending goodbye to Alice")
54                     who ! goodbye()
55                     println("ca: stopping Carl")
56                     context.stop(context.parent)})
57              }
58            }))
59            session!dest(ca)
60
61
62        }
63        override def postStop = {println("Carl: terminating"); }
64      }))
65
66      val Alice = s.actorOf(Props(new Actor{
67        val ab = context.actorOf(Props(new Actor{
68          def receive = {
69            case dest(session) =>
70              session ! ping()
71              context.become({case pong() =>
72                session ! pang();
73              })
74          }
75          override def postStop = {println("ab: terminating")}
76        }))
77
78        Bob ! new_(ab, self)
79
80
81         val ac = context.actorOf(Props(new Actor{
82          def receive = {
83            case dest(session) =>
84              session ! ping()
85              context.become({case pong() =>
86               session ! pang()
87              })
88          }
89          override def postStop = {println("ac: terminating")}
90
91        }))
92
93        Carl ! new_(ac, self)
94
```

```
95
96        def receive = {
97          case goodbye() =>
98            println("Alice: RECEIVE goodbye #1")
99            context.become({case goodbye() =>
100             println("Alice: RECEIVE goodbye #2")
101             context.stop(self)
102             })
103       }
104       override def postStop = {println("Alice: terminating")}
105     }))
106
107
108 }
```

Output 1

```
1  ca: RECEIVE pang
2  ca: sending goodbye to Alice
3  ca: stopping Carl
4  Alice: RECEIVE goodbye #1
5  Carl: terminating
6  ba: RECEIVE pang
7  ba: sending goodbye to Alice
8  ba: stopping Bob
9  Alice: RECEIVE goodbye #2
10 ba: terminating
11 ab: terminating
12 ac: terminating
13 Bob: terminating
14 Alice: terminating
```

il comportamento è quello atteso: *Alice* termina dopo aver ricevuto i due messaggi *goodbye* e dopo che i figli *ab* e *ac* sono terminati, in accordo con le regole descritte. Anche *Bob* e *Carl* prima di terminare attendono la terminazione dei figli.

Output 2

```
1  ca: RECEIVE pang
2  ca: sending goodbye to Alice
3  ca: stopping Carl
4  Alice: RECEIVE goodbye #1
5  Carl: terminating
```

```
ba: RECEIVE pang
ba: sending goodbye to Alice
ba: stopping Bob
Alice: RECEIVE goodbye #2
ba: terminating
ab: terminating
Bob: terminating
ac: terminating
Alice: terminating
```

3.2.3 Alice Bob e Carl ping pong con stop forzato

Nell'esempio precedente abbiamo scritto il programma in modo che le sessioni di gioco *ba* e *ca* salutassero *Alice* al termine del gioco e poi avviassero la terminazione dei genitori *Bob* e *Carl* (e quindi a cascata anche la loro). *Alice* chiamava uno stop su se stessa solo dopo aver ricevuto il saluto da entrambe le sessioni.

Supponiamo ora invece che, all'arrivo di un messaggio *pang*, le sessioni *ba* e *ca*, eseguano uno stop sul genitore e poi direttamente su *Alice*. *Alice* ha due figli, che sono le due sessioni di gioco aperte, una con *Bob* e una con *Carl*. La sessione di gioco, tra quelle di *Bob* e *Carl*, che finisce per prima provoca uno stop dell'attore *Alice*, che a sua volta richiede ai figli di arrestarsi per poter terminare e quindi alle sessioni *ab* e *ac* di concludersi. Ovviamente una delle due sarà già conclusa, ma l'altra potrebbe essere potenzialmente ancora in esecuzione. Supponiamo che a concludersi sia stata la sessione *ca*. Questa arresta il genitore *Carl* e arresta *Alice*. *Alice* per terminare arresta i figli *ab* e *ac*. Se la sessione di gioco di *Bob* non è ancora concluso il programma evolverà in un termine stuck (un termine che non può più evolvere con nessuna delle regole date).

Modelliamo questo nuovo programma in A+ e vediamo cosa succede runtime. Come prima il messaggio *new* ha due parametri: la sessione e l'attore (nel nostro caso *Alice*) che la richiede. Il comportamento di *Alice* torna ad essere vuoto, inoltre cambiano i comportamenti β'_{ba} e β'_{ca}:

$\beta_A = \underline{0}$
$\beta_B = \{new(z, w) \Rightarrow val\ ba = Actor\{\beta_{ba}; \underline{0}\}; z!dest(ba)\}$
$\beta_{ab} = \{dest(y) \Rightarrow y!ping(); become\{\beta'_{ab}\}\}$
$\beta_{ba} = \{ping() \Rightarrow z!pong(); become\{\beta'_{ba}\}\}$
$\beta'_{ab} = \{pong() \Rightarrow y!pang()\}$
$\beta'_{ba} = \{pang() \Rightarrow stop(Bob); stop(w)\}$

β_C, β_{ac}, β_{ca}, β'_{ac} e β'_{ca}, usati da *Carl*, si ottengono dai corrispettivi di *Bob*.

Facciamo evolvere per ora solo gli attori *Alice*, *Carl*, *ac*, *ca* e chiamiamo i restanti attori in parallelo F. Supponiamo che la sessione tra *Alice* e *Carl* stia

per terminare, ovvero che la mailbox di ca contenga il messaggio $pang$, mentre la sessione tra $Alice$ e Bob sia solo all'inizio.

$(\nu Alice)(\nu ac)(\nu ca)(\nu Carl)([Alice \mapsto \emptyset](\beta_A \uparrow \emptyset \downarrow (ab, ac))Alice\{\underline{0}\} \mid [ac \mapsto \emptyset](\beta'_{ac} \uparrow Alice \downarrow \emptyset)ac\{\underline{0}\} \mid [ca \mapsto pang()](\beta'_{ca} \uparrow Carl \downarrow \emptyset)ca\{\underline{0}\} \mid [Carl \mapsto \emptyset](\beta_C \uparrow \emptyset \downarrow ca)Carl\{\underline{0}\} \mid F)$

con

$F = [Bob \mapsto \emptyset](\beta_B \uparrow \emptyset \downarrow ba)Bob\{\underline{0}\} \mid [ba \mapsto \emptyset](\beta_{ba} \uparrow Bob \downarrow \emptyset)ba\{ab!pong(); become(\beta'_{ba}); \} \mid [ab \mapsto \emptyset](\beta'_{ab} \uparrow Alice \downarrow \emptyset)ab\{\underline{0}\}$

Questa configurazione evolve nel seguente modo:

$\hookrightarrow^{RECEIVE} (\nu Alice)(\nu ac)(\nu ca)(\nu Carl)([Alice \mapsto \emptyset](\beta_A \uparrow \emptyset \downarrow (ab, ac))Alice\{\underline{0}\} \mid [ac \mapsto \emptyset](\beta'_{ac} \uparrow Alice \downarrow \emptyset)ac\{\underline{0}\} \mid \underline{[ca \mapsto \emptyset](\beta'_{ca} \uparrow Carl \downarrow \emptyset)ca\{stop(Carl); stop(Alice)\}} \mid [Carl \mapsto \emptyset](\beta_C \uparrow \emptyset \downarrow ca)Carl\{\underline{0}\} \mid F)$

$\hookrightarrow^{STOP} (\nu Alice)(\nu ac)(\nu ca)(\nu Carl)([Alice \mapsto \emptyset](\beta_A \uparrow \emptyset \downarrow (ab, ac))Alice\{\underline{0}\} \mid [ac \mapsto \emptyset](\beta'_{ac} \uparrow Alice \downarrow \emptyset)ac\{\underline{0}\} \mid [ca \mapsto \emptyset](\beta'_{ca} \uparrow Carl \downarrow \emptyset)ca\{stop(Alice)\} \mid [Carl \mapsto \emptyset](\beta_C \uparrow \emptyset \downarrow ca)Carl\{stop(Carl)\} \mid F)$

A questo punto possono evolvere $Carl$ o ca, scegliamo di far evolvere prima ca (l'ordine è indifferente):

$\hookrightarrow^{STOP} (\nu Alice)(\nu ac)(\nu ca)(\nu Carl)([Alice \mapsto \emptyset](\beta_A \uparrow \emptyset \downarrow (ab, ac))Alice\{stop(Alice)\} \mid [ac \mapsto \emptyset](\beta'_{ac} \uparrow Alice \downarrow \emptyset)ac\{\underline{0}\} \mid [ca \mapsto \emptyset](\beta'_{ca} \uparrow Carl \downarrow \emptyset)ca\{\underline{0}\} \mid [Carl \mapsto \emptyset](\beta_C \uparrow \emptyset \downarrow ca)Carl\{stop(Carl)\} \mid F)$

Ora $Carl$ eseguirà la sua terminazione come già più volte descritto, chiamando uno $stop$ sul figlio ca e poi terminando se stesso. Allo stesso modo $Alice$ chiederà ai suoi due figli ab e ac di terminare e poi terminerà. Ci ritroviamo quindi nella seguente situazione:

$(\nu Alice)(\nu ac)(\nu ab)(\nu ca)(\nu Carl)([Alice \mapsto \emptyset] \mid [Carl \mapsto \emptyset] \mid [ac \mapsto \emptyset] \mid [ab \mapsto \emptyset] \mid [ca \mapsto \emptyset] \mid F')$

con

$F' = [Bob \mapsto \emptyset](\beta_B \uparrow \emptyset \downarrow ba)Bob\{\underline{0}\} \mid$
$[ba \mapsto \emptyset](\beta_{ba} \uparrow Bob \downarrow \emptyset)ba\{ab!pong(); become(\beta'_{ba}); \}.$

$Alice$ ha terminato la sua sessione ab. Dal momento che $ac, ca \notin fn(F')$, mentre ba ha ancora sia il riferimento ad ab che ad $Alice$, la configurazione risulta

$\equiv (\nu Alice)(\nu ab)([Alice \mapsto \emptyset] \mid [ab \mapsto \emptyset] \mid F')$

L'unico attore che può evolvere ora è *ba*, che cerca di inviare un messaggio *pong* ad *ab* che però è terminato, e poi passa ad avere un comportamento atto a gestire messaggi *pang*:

$\longrightarrow^{SEND-DEAD} (\nu Alice)(\nu ab)(\nu Bob)(\nu ba)([Alice \mapsto \emptyset] \mid [ab \mapsto \emptyset] \mid [Bob \mapsto \emptyset](\beta_B \uparrow \emptyset \downarrow ba)Bob\{\underline{0}\} \mid [ba \mapsto \emptyset](\beta'_{ba} \uparrow Bob \downarrow \emptyset)ba\{become(\beta'_{ba});\})$

$\longrightarrow^{BECOME} (\nu Alice)(\nu ab)(\nu Bob)(\nu ba)([Alice \mapsto \emptyset] \mid [ab \mapsto \emptyset] \mid [Bob \mapsto \emptyset](\beta_B \uparrow \emptyset \downarrow ba)Bob\{\underline{0}\} \mid [ba \mapsto \emptyset](\beta'_{ba} \uparrow Bob \downarrow \emptyset)ba\{\underline{0}\})$

A questo punto *ba* si mette in attesa di un messaggio *pang* che non può arrivare in quanto la sessione *ab* è terminata. La configurazione a cui siamo arrivati è una configurazione stuck.

Verifichiamo che questo si verifica anche nel programma scritto in Scala Akka.

Codice Scala Akka

```
1  import akka.actor.Actor
2  import akka.actor.ActorRef
3  import akka.actor.Props
4  import akka.actor.ActorSystem
5
6
7  object PingPongCruelStop extends App{
8
9      case class dest(y:ActorRef)
10     case class new_(xy: ActorRef, who:ActorRef)
11     case class ping()
12     case class pong()
13     case class pang()
14
15     val s = ActorSystem()
16
17     val Bob = s.actorOf(Props(new Actor{
18       def receive = {
19         case new_(session, who) =>
20
21           val ba  = context.actorOf(Props(new Actor{
22             def receive = {
```

```
23              case ping() =>
24                Thread.sleep(12000)
25                println("ba: SEND ab ! PONG")
26                session!pong();
27                println("ba: BECOME case PANG")
28                context.become({case pang() =>
29                      println("ba: RECEIVE pang")
30                      println("ba: stopping Bob")
31                      context.stop(context.parent)
32                      println("ba: stopping Alice")
33                      context.stop(who)})
34            }
35            override def postStop = {println("ba: terminating")}
36          }))
37
38          session!dest(ba)
39
40      }
41      override def postStop = {println("Bob: terminating");}
42    }))
43
44    val Carl  = s.actorOf(Props(new Actor{
45      def receive = {
46        case new_(session, who) =>
47          val ca = context.actorOf(Props(new Actor{
48            def receive = {
49              case ping() =>
50                session!pong();
51                context.become({case pang() =>
52                     println("ca: RECEIVE pang")
53                     println("ca: stopping Carl")
54                     context.stop(context.parent)
55                     println("ca: stopping Alice")
56                     context.stop(who)})
57              }
58          override def postStop = {println("ca: terminating")}
59          }))
60          session!dest(ca)
61
62      }
63      override def postStop = {println("Carl: terminating"); }
64    }))
65
66    val Alice  = s.actorOf(Props(new Actor{
67      val ab = context.actorOf(Props(new Actor{
```

```
68          def receive = {
69            case dest(y) =>
70              y ! ping()
71              context.become({case pong() =>
72                y ! pang();
73              })
74          }
75          override def postStop = {println("ab: terminating")}
76        }))
77
78        Bob ! new_(ab, self)
79
80         val ac = context.actorOf(Props(new Actor{
81          def receive = {
82            case dest(session) =>
83              session ! ping()
84              context.become({case pong() =>
85               session ! pang()
86              })
87          }
88          override def postStop = {println("ac: terminating")}
89
90        }))
91
92        Carl ! new_(ac, self)
93
94
95        def receive = {PartialFunction.empty }
96        override def postStop = {println("Alice: terminating")}
97      }))
98
99
100 }
```

Nel codice, al fine di avere l'interleaving descritto sopra, è stato modificato il comportamento dell'attore *ba* alla ricezione di un messaggio *ping*, in modo che si metta in pausa per un po' prima di inviare il messaggio di risposta *pong* ad *ab*. Questo per permettere al ping pong tra *Alice* e *Carl* di finire e quindi di mandare la terminazione ad *Alice* (e quindi ai suoi figli *ac*, *ab*), prima che finisca il ping pong tra *Alice* e *Bob*.

Output

```
1 ca: RECEIVE pang
2 ca: stopping Carl
```

```
ca: stopping Alice
ac: terminating
ca: terminating
ab: terminating
Alice: terminating
Carl: terminating
ba: SEND ab ! PONG
ba: BECOME case PANG
[INFO] [01/26/2016 20:21:01.430]
[default-akka.actor.default-dispatcher-7]
[akka://default/user/$c/$a] Message [PingPongCruelStop$pong]
from Actor[akka://default/user/$a/$a#-494843951]
to Actor[akka://default/user/$c/$a#1435377659]
was not delivered. [1] dead letters encountered.
```

Guardando l'output del programma possiamo vedere che, con la stampa `ca: RECEIVE pang`, termina il ping pong tra *Alice* e *Carl*. Dal codice vediamo che questo implica uno `stop` su *Alice* che infatti possiamo vedere dall'output che subito dopo termina i suoi figli *ac* e *ab* e termina. Dopo la terminazione di *Alice*, *ba* cerca di inviare un messaggio *pong* che, esattamente come descritto dalle regole di semantica date, non arriva all'attore terminato. Possiamo vedere questo nell'output dal messaggio di log `message was not delivered`. Infine, anche nel programma scritto in Scala Akka si giunge ad una configurazione stuck. Infatti gli unici attori attivi sono *Bob* e *ba*, entrambi senza corpo da eseguire né messaggi in mailbox. Come si vede dalle stampe *ba* resta in attesa di un messaggio *pang* che, come detto, non gli può arrivare.

Capitolo 4

Definizione del type system

I sistemi di tipi rappresentano un tool per ragionare sui programmi, garantendo che il comportamento di un programma rispetti una specifica data o un comportamento atteso. In particolare lo scopo di questo capitolo è mostrare che, se un programma sorgente è ben tipato secondo il sistema di tipi dato, allora ogni attore sarà in grado di ricevere tutti i messaggi che gli vengono inviati.

Il sistema di tipi che andiamo a introdurre vuole avvicinarsi al sistema di tipi sviluppato da Akka Typed per lo stesso scopo: garantire staticamente la ricezione dei messaggi.

Estendiamo il linguaggio A assegnando i tipi ai nomi degli attori. Un tipo indica l'insieme di messaggi a cui l'attore è in grado di rispondere:

$$T \ ::= \ \langle m_1(\tilde{T_1}), \ldots, m_k(\tilde{T_k}) \rangle$$

con $k \geq 0$ dove con $k = 0$ indichiamo il tipo $\langle\rangle$ di un attore che non è in grado di ricevere alcun messaggio.
Assegnare un tipo ai nomi degli attori nel nostro linguaggio A_T equivale all'assegnazione di un tipo agli `ActorRef` di Akka Typed. Questo significa che se in Akka ho un tipo di messaggi `A`, allora il tipo `ActorRef[A]` corrisponde al nostro tipo $\langle A \rangle$.
Il linguaggio su cui lavoriamo, chiamato A_T ha una sintassi tipata:

Espressioni	$e ::= \underline{0} \mid u!m(\tilde{u}); e \mid val\ a{:}T = Actor\{\beta; e\}; e \mid become(\beta); e$
Comportamenti	$\beta ::= \{m_i(\tilde{x_i}) \Rightarrow e_i\}_{i \in I}$

Configurazioni	$F ::= \underline{0} \mid [a \to M](\beta)a\{e\} \mid F \vert F \mid (\nu a{:}T)F \mid d$
	$d ::= \underline{0} \mid val\ a{:}T = Actor\{\beta; e\}; d$
	$M ::= \emptyset \mid m(\tilde{a}) \cdot M$

Osserviamo che non è necessario tipare anche le variabili nei pattern contenuti nei comportamenti, poiché le informazioni sul loro tipo si ricostruiscono facilmente dal tipo dell'attore.

Semantica operazionale

Le regole di riduzione, per il linguaggio A_T rimangono le stesse del linguaggio A, riscritte però per il linguaggio che usa la sintassi tipata. In particolare quindi vengono riscritte (TOP-SPAWN) e (SPAWN).

(TOP-SPAWN)

$$\frac{}{val\ a{:}T = Actor\{\beta; e\}; d \longrightarrow (\nu a{:}T)([a \mapsto \emptyset](\beta)a\{e\} \mid d)}$$

(SPAWN)

$$\frac{a \notin (fn(\beta) \cup fn(M))}{[b \mapsto M](\beta)b\{val\ a{:}T = Actor\{\beta'; e\}\ ; e'\} \rightarrow (\nu a{:}T)([b \rightarrow M](\beta)b\{e'\} \mid [a \mapsto \emptyset](\beta')a\{e\})}$$

Cambiano inoltre gli assiomi:

(1) $(\nu a{:}T)(\nu b{:}T')F \equiv (\nu b{:}T')(\nu a{:}T)F$
(2) $(\nu a{:}T)(F|F') \equiv F|(\nu a{:}T)F' \quad a \notin fn(F)$
(6) $(\nu a{:}T)\underline{0} \equiv \underline{0}$

Lasciamo per ora la regola (JUNK). Dimostreremo in questo capitolo che, per gli attori ben tipati, la regola (JUNK) non viene mai utilizzata, e quindi si potrebbe eliminare dalle regole di riduzione.

Contesti e giudizi di tipo I contesti assegnano tipi a nomi e a variabili, cioè:

$$\Gamma \ ::= \ \emptyset \mid u : T, \Gamma$$

Useremo i seguenti giudizi di tipo:

- $\Gamma \vdash \diamond$ il contesto Γ è ben formato
- $\Gamma \vdash u : T$ il nome/variabile u ha tipo T in Γ
- $\Gamma \vdash_a e$ l'espressione e è ben tipata all'interno dell'attore a secondo Γ
- $\Gamma \vdash_a \beta$ il comportamento β è ben tipato per l'attore a secondo Γ
- $\Gamma \vdash [a \mapsto M]$ la mailbox $[a \mapsto M]$ dell'attore a è ben tipata in Γ
- $\Gamma \vdash F$ la configurazione F è ben tipata in Γ

Subtyping La relazione di subtyping per i tipi degli attori è definita sia in larghezza che in profondità. Il subtyping in larghezza è in accordo con la definizione di controvarianza `ActorRef[-T]` data in Akka Typed.

$$\text{(Reflex)}\ \frac{}{T <: T} \qquad \text{(Trans)}\ \frac{S <: U \quad U <: T}{S <: T}$$

$$\text{(Sub Width)}\ \frac{}{\langle m_1(\tilde{T_1}), .., m_{k+n}(\tilde{T}_{k+n})\rangle <: \langle m_1(\tilde{T_1}), .., m_k(\tilde{T_k})\rangle}$$

$$\text{(Sub Depth)}\ \frac{\tilde{S_i} <: \tilde{T_i} \quad \forall i = 1, .., k}{\langle m_1(\tilde{T_1}), .., m_k(\tilde{T_k})\rangle <: \langle m_1(\tilde{S_1}), .., m_k(\tilde{S_k})\rangle}$$

4.1 Regole di tipo

Typing dei contesti e dei nomi/variabili

$$\text{(Empty Ctx)}\ \frac{}{\emptyset \vdash \diamond} \qquad \text{(Ctx)}\ \frac{\Gamma \vdash \diamond \quad u \notin Dom(\Gamma)}{\Gamma, u : T \vdash \diamond} \qquad \text{(Project)}\ \frac{\Gamma, u : T \vdash \diamond}{\Gamma, u : T \vdash u : T} \qquad \text{(Subsumption)}\ \frac{\Gamma \vdash u : T \quad T <: S}{\Gamma \vdash u : S}$$

Typing delle espressioni

$$\text{(Empty Expr)}\ \frac{\Gamma \vdash \diamond}{\Gamma \vdash_a \underline{0}} \qquad \text{(Type Send)}\ \frac{\Gamma \vdash u : \langle m(T_1, .., T_k)\rangle \quad \Gamma \vdash u_i : T_i\ \forall i{=}1, .., k \quad \Gamma \vdash_a e}{\Gamma \vdash_a u!m(u_1, \ldots, u_k); e}$$

$$\text{(Type Become)}\ \frac{\Gamma \vdash_a \beta \quad \Gamma \vdash_a e}{\Gamma \vdash_a become(\beta); e} \qquad \text{(Type Actor)}\ \frac{\Gamma, a : T \vdash_a \beta \quad \Gamma, a : T \vdash_a e \quad \Gamma, a : T \vdash_b e'}{\Gamma \vdash_b val\ a : T = Actor\{\beta; e\}; e'}$$

Typing dei comportamenti

(TYPE BEHAVE)
$$\frac{a : \langle m_1(\tilde{T_1}), \ldots, m_k(\tilde{T_k})\rangle \in \Gamma \quad \Gamma, \tilde{x_i} : \tilde{T_i} \vdash_a e_i \quad \forall i = 1, .., k}{\Gamma \vdash_a \{m_1(\tilde{x_1}) => e_1, \ldots, m_k(\tilde{x_k}) => e_k\}}$$

Typing delle mailbox e delle configurazioni

(TYPE EMPTY MAIL)
$$\frac{\Gamma \vdash a : T}{\Gamma \vdash [a \mapsto \emptyset]}$$

(TYPE MAILBOX)
$$\frac{\Gamma \vdash a : \langle m(T_1, .., T_k)\rangle \quad \Gamma \vdash b_i : T_i \ \forall i = 1, .., k \quad \Gamma \vdash [a \mapsto M]}{\Gamma \vdash [a \mapsto m(b_1, \ldots, b_k) \cdot M]}$$

(TYPE EMPTY CONF)
$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \underline{0}}$$

(TYPE CONF)
$$\frac{\Gamma \vdash [a \mapsto M] \quad \Gamma \vdash_a \beta \quad \Gamma \vdash_a e}{\Gamma \vdash [a \mapsto M](\beta)a\{e\}}$$

(TYPE PAR)
$$\frac{\Gamma \vdash F_1 \quad \Gamma \vdash F_2}{\Gamma \vdash F_1 \mid F_2}$$

(TYPE RES)
$$\frac{\Gamma, a : T \vdash F}{\Gamma \vdash (\nu a{:}T)F}$$

(TYPE TOP ACTOR)
$$\frac{\Gamma, a : T \vdash_a \beta \quad \Gamma, a : T \vdash_a e \quad \Gamma, a : T \vdash d}{\Gamma \vdash val\ a : T = Actor\{\beta; e\}; d}$$

Descriviamo brevemente le regole principali.

La regola (TYPE SEND) si applica quando un attore a invia all'attore u un messaggio $m(u_1, .., u_k)$. Perché l'espressione di invio del messaggio sia ben tipata in un contesto Γ occorre che in Γ u abbia un tipo adatto a ricevere un messaggio m con parametri $u_1, .., u_k$ che in Γ hanno un sottotipo di $T_1, .., T_k$. Inoltre il rimanente corpo di a deve essere ben tipato in Γ.

(TYPE BECOME) dichiara che un attore a che esegue una $become(\beta)$ e prosegue come e è ben tipato in un contesto Γ se nel contesto sono ben tipati il nuovo comportamento β e la continuazione e.

L'espressione di creazione di un nuovo attore $a : T$ dal corpo di un attore b, che successivamente prosegue con e' è ben tipata in Γ, secondo la regola (TYPE ACTOR), se nel contesto con l'aggiunta dell'assegnazione $a : T$ sono ben tipati il comportamento e il corpo del nuovo attore a e la continuazione e' di b.

La regola (TYPE BEHAVE) si applica per definire un comportamento ben tipato. Un comportamento $\{m_1(\tilde{x}_1) \Rightarrow e_1, .., m_k(\tilde{x}_k) \Rightarrow e_k\}$ all'interno del corpo di a è ben tipato in Γ se in Γ il tipo di a è $\langle m_1(\tilde{T_1}), .., m_k(\tilde{T_k})\rangle$, ovvero è esattamente il set di messaggi che il comportamento è in grado di gestire. In questo modo si assicura che un attore ben tipato possa modificare il proprio comportamento senza modificare il set di messaggi a cui risponde. Inoltre per ogni case statement del comportamento l'espressione e_j deve essere ben tipata nel contesto con in aggiunta la corrispondente variabile $x_j : T_j$.

(TYPE MAILBOX) afferma che la mailbox $[a \mapsto m(b_1, .., b_k) \cdot M]$ dell'attore a è ben tipata in Γ se in Γ a ha un sottotipo di $\langle m(T_1, .., T_k)\rangle$ che permette di

ricevere il primo messaggio, tutti i parametri del messaggio sono ben tipati ed è ben tipato il rimanente insieme di messaggi M.

(TYPE CONF), (TYPE PAR) e (TYPE RES) indicano rispettivamente quando sono ben tipati un attore attivo, una coppia di configurazioni in parallelo e una configurazione con un nome a legato.

Infine la regola (TYPE TOP ACTOR) definisce quando è ben tipata in Γ un'espressione di creazione di attore $a : T$ top-level che prosegue con un'espressione d. Le condizioni richiedono che nel contesto Γ con l'aggiunta dell'assegnazione $a : T$ siano ben tipati il comportamento del nuovo attore, il suo corpo e la sequenza di espressioni di creazione di attori top-level d.

4.2 Proprietà del sistema di tipi

Lemma 1 (Lemma di inversione)**.**

1. *se* $\Gamma, u{:}T \vdash \diamond$ *è derivabile, allora* $\Gamma \vdash \diamond$, $u \notin Dom(\Gamma)$.
2. *se* $\Gamma \vdash u{:}T$ *è derivabile, allora* $\Gamma = \Gamma', u{:}T'$ *con* $T' <: T$ *e* $\Gamma \vdash \diamond$.
3. *se* $\Gamma \vdash u{:}S$ *è derivabile e* $S = \langle m_1(\tilde{S_1}), ..., m_k(\tilde{S_k})\rangle$ *allora* $u{:}T \in \Gamma$, $\exists T$ *con* $T = \langle m_1(\tilde{T_1}), .., m_{k+n}(\tilde{T}_{k+n})\rangle$ *e* $\tilde{S_i} <: \tilde{T_i} \quad \forall i = 1, .., k$
4. *se* $\Gamma \vdash_a \underline{0}$ *è derivabile, allora* $\Gamma \vdash \diamond$.
5. *se* $\Gamma \vdash_a u!m(u_1, ..., u_k); e$ *è derivabile, allora* $\exists T_1, .., T_k$ *tali che* $\Gamma \vdash u{:}\langle m(T_1, .., T_k)\rangle$, $\Gamma \vdash u_i{:}T_i \quad \forall i = 1, .., k$, $\Gamma \vdash_a e$.
6. *se* $\Gamma \vdash_a become(\beta); e$ *è derivabile, allora* $\Gamma \vdash_a \beta$, $\Gamma \vdash_a e$.
7. *se* $\Gamma \vdash_b val\ a{:}T = Actor\{\beta; e\}; e'$ *è derivabile, allora* $\Gamma, a{:}T \vdash_a \beta$, $\Gamma, a{:}T \vdash_a e$, $\Gamma, a{:}T \vdash_b e'$.
8. *se* $\Gamma \vdash_a \{m_1(\tilde{x}_1) \Rightarrow e_1, .., m_k(\tilde{x}_k) \Rightarrow e_k\}$ *è derivabile, allora* $\Gamma \ni a{:}\langle m_1(\tilde{T_1}), .., m_k(\tilde{T_k})\rangle$, $\Gamma, \tilde{x}_i{:}\tilde{T_i} \vdash_a e_i \quad \forall i = 1, .., k$.
9. *se* $\Gamma \vdash [a \mapsto \emptyset]$ *è derivabile, allora* $\exists T$ *tale che* $\Gamma \vdash a{:}T$.
10. *se* $\Gamma \vdash [a \mapsto m(b_1, ..., b_k) \cdot M]$ *è derivabile, allora* $\exists T_1, .., T_k$ *tali che* $\Gamma \vdash a{:}\langle m(T_1, ..., T_k)\rangle$, $\Gamma \vdash b_i{:}T_i \quad \forall i = 1, .., k$, $\Gamma \vdash [a \mapsto M]$.
11. *se* $\Gamma \vdash \underline{0}$ *è derivabile, allora* $\Gamma \vdash \diamond$.
12. *se* $\Gamma \vdash [a \mapsto M](\beta)a\{e\}$ *è derivabile, allora* $\Gamma \vdash [a \mapsto M]$, $\Gamma \vdash_a \beta$, $\Gamma \vdash_a e$.
13. *se* $\Gamma \vdash F_1 \mid F_2$ *è derivabile, allora* $\Gamma \vdash F_1$, $\Gamma \vdash F_2$.
14. *se* $\Gamma \vdash (\nu a{:}T)F$ *è derivabile, allora* $\Gamma, a{:}T \vdash F$.
15. *se* $\Gamma \vdash val\ a{:}T = Actor\{\beta; e\}; d$ *è derivabile, allora* $\Gamma, a{:}T \vdash_a \beta$, $\Gamma, a{:}T \vdash_a e$, $\Gamma, a{:}T \vdash d$.

Dimostrazione. Immediata dalle definizioni delle regole di tipo e di subtyping. □

Lemma 2. *Se* $\Gamma \vdash F$ *allora* $\Gamma \vdash \diamond$.

Proposizione 1 (Lemma di sostituzione)**.**

- *Se* $\Gamma, x{:}S \vdash u : T$ *e* $\Gamma \vdash b : S$ *allora* $\Gamma \vdash u\{b/x\}{:}T$.
- *Se* $\Gamma, x{:}S \vdash_a \beta$ *e* $\Gamma \vdash b : S$ *allora* $\Gamma \vdash_a \beta\{b/x\}$
- *Se* $\Gamma, x{:}S \vdash_a e$ *e* $\Gamma \vdash b : S$ *allora* $\Gamma \vdash_a e\{b/x\}$.

Dimostrazione. Per induzione simultanea sull'altezza della derivazione dei giudizi $\Gamma, x{:}S \vdash u{:}T$, $\Gamma, x{:}S \vdash_a \beta$, $\Gamma, x{:}S \vdash_a e$. I casi base sono gli assiomi (PROJECT) e (EMPTY EXPR):

- l'ipotesi è $\Gamma, x{:}S \vdash x{:}T$, $\Gamma \vdash b{:}S$. La tesi $\Gamma \vdash x\{b/x\}{:}T$.

  Dall'ipotesi $\Gamma, x{:}S \vdash x{:}T$, per il lemma di inversione, si ha $S <: T$. Per la definizione di sostituzione $x\{b/x\} = b$. Dall'ipotesi $\Gamma \vdash b{:}S$ e da $S <: T$ si ottiene la tesi per la regola (SUBSUMPTION).

- l'ipotesi è $\Gamma, x{:}S \vdash u{:}T$ con $u \neq x$, $\Gamma \vdash b{:}S$. La tesi $\Gamma \vdash u\{b/x\}{:}T$.

  Per il lemma di inversione si ha $\Gamma = \Gamma', u{:}T'$ con $T' <: T$. Quindi per (PROJECT) si ha $\Gamma', u{:}T', x{:}S \vdash u{:}T'$. Da questo, per il lemma di strenghtening si ha $\Gamma', u : T' \vdash u : T'$. Da questo giudizio, per (SUBSUMPTION) si ha $\Gamma', u : T' \vdash u : T$ cioè $\Gamma \vdash u : T$ che, per la definizione di sostituzione, è la tesi.

- l'ipotesi è $\Gamma, x{:}S \vdash_a \underline{0}$, $\Gamma \vdash b{:}S$. La tesi $\Gamma \vdash_a \underline{0}\{b/x\}$.

  Dal giudizio in ipotesi, per il lemma di inversione, si ha $\Gamma, x{:}S \vdash \diamond$ e quindi $\Gamma \vdash \diamond$. Per la regola (EMPTY EXPR) si ha $\Gamma \vdash_a \underline{0}$ che, per la definizione di sostituzione $\underline{0}\{b/x\} = \underline{0}$ è esattamente la tesi.

I casi induttivi sono quelli per cui il giudizio in ipotesi è stato derivato con una derivazione di altezza $k+1$. Distinguiamo i casi a seconda dell'ultima regola di tipo usata:

- l'ipotesi è $\Gamma, x{:}S \vdash_a u!m(u_1, .., u_k); e, \Gamma \vdash b{:}S$. La tesi $\Gamma \vdash_a (u!m(u_1, .., u_k); e)\{b/x\}$.

  Il giudizio in ipotesi è stato derivato da $\Gamma, x{:}S \vdash u{:}\langle m(T_1, .., T_k)\rangle$, $\Gamma, x{:}S \vdash u_i{:}T_i \ \forall i = 1, .., k$, $\Gamma, x{:}S \vdash_a e$. Per ipotesi induttiva si ha $\Gamma \vdash u\{b/x\}{:}\langle m(T_1, .., T_k)\rangle$, $\Gamma \vdash u_i\{b/x\}{:}T_i \ \forall i = 1, .., k$, $\Gamma \vdash_a e\{b/x\}$. Da questi giudizi, per la regola (TYPE SEND), si ha $\Gamma \vdash_a u\{b/x\} \ ! \ m((u_1, .., u_k)\{b/x\}); e\{b/x\}$, che per la definizione di sostituzione è esattamente la tesi.

- l'ipotesi è $\Gamma, x{:}S \vdash_a become(\beta); e, \Gamma \vdash b{:}S$. La tesi $\Gamma \vdash_a (become(\beta); e)\{b/x\}$.

  Il giudizio in ipotesi è stato derivato da $\Gamma, x{:}S \vdash_a \beta$, $\Gamma, x{:}S \vdash_a e$. Per ipotesi induttiva si ha $\Gamma \vdash (\beta)\{b/x\}$, $\Gamma \vdash_a e\{b/x\}$. Da questi giudizi, per la regola (TYPE BECOME), si ha $\Gamma \vdash_a become(\beta\{b/x\}); e\{b/x\}$, che per la definizione di sostituzione è esattamente la tesi.

- l'ipotesi è $\Gamma, x : S \vdash_{b'} val\ a{:}T = Actor\{\beta; e\}; e'$, $\Gamma \vdash b{:}S$. La tesi $\Gamma \vdash_{b'} (val\ a{:}T = Actor\{\beta; e\}; e')\{b/x\}$.

  Il giudizio in ipotesi è stato derivato da $\Gamma, x{:}S, a{:}T \vdash_a \beta$, $\Gamma, x{:}S, a{:}T \vdash_a e$, $\Gamma, x{:}S, a{:}T \vdash_{b'} e'$. Per ipotesi induttiva si ha $\Gamma, a{:}T \vdash_a (\beta)\{b/x\}$, $\Gamma, a{:}T \vdash_a e\{b/x\}$, $\Gamma, a{:}T \vdash_{b'} e'\{b/x\}$. Da questi giudizi, per la regola (TYPE ACTOR), si ha $\Gamma \vdash_{b'} val\ a{:}T = Actor\{\beta\{b/x\}; e\{b/x\}\}; e'\{b/x\}$, che per la definizione di sostituzione è esattamente la tesi.

- l'ipotesi è $\Gamma, x{:}S \vdash_a \{m_1(\tilde{x}_1) \Rightarrow e_1, .., m_k(\tilde{x}_k) \Rightarrow e_k\}$, $\Gamma \vdash b{:}S$. La tesi $\Gamma \vdash_a \{m_1(\tilde{x}_1) \Rightarrow e_1, .., m_k(\tilde{x}_k) \Rightarrow e_k\}\{b/x\}$.

  Il giudizio in ipotesi è stato derivato da $\Gamma, x{:}S \ni a{:}\langle m_1(\tilde{T}_1), .., m_k(\tilde{T}_k)\rangle$, $\Gamma, x{:}S, \tilde{x}_i{:}\tilde{T}_i \vdash_a e_i \quad \forall i = 1, .., k$. Per ipotesi induttiva si ha $\Gamma, \tilde{x}_i{:}\tilde{T}_i \vdash_a e_i\{b/x\} \quad \forall i = 1, .., k$. Inoltre, poichè $a \neq x$ si ha $\Gamma \ni a{:}\langle m_1(\tilde{T}_1), .., m_k(\tilde{T}_k)\rangle$. Da questi due giudizi si deriva, per la regola (TYPE BEHAVE), $\Gamma \vdash_a \{m_1(\tilde{x}_1) \Rightarrow e_1, .., m_k(\tilde{x}_k) \Rightarrow e_k\{b/x\}\}$ che per definizione di sostituzione è esattamente la tesi.

- l'ipotesi è $\Gamma, x{:}S \vdash u{:}T$, poiché $\Gamma, x{:}S \vdash u{:}T'$ con $T' <: T$. Per ipotesi induttiva $\Gamma \vdash u\{b/x\}{:}T'$. Da questo giudizio e da $T' <: T$, per (SUBSUMPTION), otteniamo $\Gamma \vdash u\{b/x\}{:}T$ che è la tesi.

□

Lemma 3 (Permutazione)**.** *Se* $\Gamma, a{:}T, b{:}T' \vdash F$ *allora* $\Gamma, b{:}T', a{:}T \vdash F$.

Lemma 4 (Strenghtening)**.**

- *Se* $\Gamma, a{:}T \vdash F$ *e* $a \notin fn(F)$ *allora* $\Gamma \vdash F$.
- *Se* $\Gamma, x{:}T \vdash F$ *e* $a \notin fv(F)$ *allora* $\Gamma \vdash F$.

Lemma 5 (Weakening)**.**

- *Se* $\Gamma \vdash \diamond$ *è un giudizio derivabile e* $w{:}S \notin Dom(\Gamma)$ *allora* $\Gamma, w{:}S \vdash \diamond$ *è derivabile.*
- *Se* $\Gamma \vdash u{:}T$ *è un giudizio derivabile e* $w{:}S \notin Dom(\Gamma)$ *allora* $\Gamma, w{:}S \vdash u{:}T$ *è derivabile.*
- *Se* $\Gamma \vdash_a e$ *è un giudizio derivabile e* $w{:}S \notin Dom(\Gamma)$ *allora* $\Gamma, w{:}S \vdash_a e$ *è derivabile.*
- *Se* $\Gamma \vdash_a \beta$ *è un giudizio derivabile e* $w{:}S \notin Dom(\Gamma)$ *allora* $\Gamma, w{:}S \vdash_a \beta$ *è derivabile.*
- *Se* $\Gamma \vdash [a \mapsto M]$ *è un giudizio derivabile e* $w{:}S \notin Dom(\Gamma)$ *allora* $\Gamma, w{:}S \vdash [a \mapsto M]$ *è derivabile.*
- *Se* $\Gamma \vdash F$ *è un giudizio derivabile e* $w{:}S \notin Dom(\Gamma)$ *allora* $\Gamma, w{:}S \vdash F$ *è derivabile.*

Dimostrazione. Per induzione simultanea sull'altezza della derivazione del giudizio in ipotesi. I casi base sono quelli per cui il giudizio di tipo viene da assioma:

- l'ipotesi è $\emptyset \vdash \diamond$, $w{:}S \notin Dom(\emptyset)$, la tesi $w{:}S \vdash \diamond$.

  Dalle ipotesi per la regola (CTX) si ha $w{:}S \vdash \diamond$.

I casi induttivi sono quelli in cui il giudizio in ipotesi è stato derivato con una derivazione di altezza $k+1$. Distinguiamo i casi a seconda dell'ultima regola di tipo usata:

- l'ipotesi è $\Gamma, u{:}T \vdash \diamond$, $w{:}S \notin Dom(\Gamma, u{:}T)$, la tesi $\Gamma, u{:}T, w{:}S \vdash \diamond$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash \diamond$, $u \notin Dom(\Gamma)$. Per ipotesi induttiva $\Gamma, w{:}S \vdash \diamond$. Dall'ipotesi si ha inoltre $u \neq w$ e quindi anche $u{:}T \notin Dom(\Gamma, w{:}S)$. Quindi per la regola di tipo (CTX) $\Gamma, w{:}S, u{:}T \vdash \diamond$. La tesi si ottiene per il lemma di permutazione.

- l'ipotesi è $\Gamma, u{:}T \vdash u{:}T$, $w{:}S \notin Dom(\Gamma, u{:}T)$, la tesi $\Gamma, u{:}T, w{:}S \vdash u{:}T$.

  Il giudizio in ipotesi è stato derivato da $\Gamma, u{:}T \vdash \diamond$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S, u{:}T \vdash \diamond$ da cui si ottiene la tesi per la regola (PROJECT), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash u{:}S$, $w{:}S \notin Dom(\Gamma)$. La tesi $\Gamma, w{:}S \vdash u{:}S$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash u{:}T$, $T <: S$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash u{:}T$. Da questo e dall'ipotesi $T <: S$ si ottiene la tesi per la regola (SUBSUMPTION), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash_a \underline{0}$, $w{:}S \notin Dom(\Gamma)$, la tesi $\Gamma, w{:}S \vdash_a \underline{0}$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash \diamond$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash \diamond$ da cui si ottiene la tesi per la regola (EMPTY EXPR), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash_a u!m(u_1, ..., u_k); e$, $w{:}S \notin Dom(\Gamma)$, la tesi $\Gamma, w{:}S \vdash_a u!m(u_1, ..., u_k); e$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash u{:}\langle m(T_1, .., T_k)\rangle$, $\Gamma \vdash u_i{:}T_i\ \forall i = 1, .., k$, $\Gamma \vdash_a e$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash u{:}\langle m(T_1, .., T_k)\rangle$, $\Gamma, w{:}S \vdash u_i{:}T_i\ \forall i = 1, .., k$, $\Gamma, w{:}S \vdash_a e$ da cui si ottiene la tesi per la regola (TYPE SEND), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash_a become(\beta); e$, $w{:}S \notin Dom(\Gamma)$, la tesi $\Gamma, w{:}S \vdash_a become(\beta); e$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash_a \beta$, $\Gamma \vdash_a e$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash_a \beta$, $\Gamma, w{:}S \vdash_a e$ da cui si ottiene la tesi per la regola (TYPE BECOME), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash_b val\ a{:}T = Actor\{\beta; e\}; e'$, $w{:}S \notin Dom(\Gamma)$, la tesi $\Gamma, w{:}S \vdash_b val\ a{:}T = Actor\{\beta; e\}; e'$.

  Il giudizio in ipotesi è stato derivato da $\Gamma, a{:}T \vdash_a \beta$, $\Gamma, a{:}T \vdash_a e$, $\Gamma, a{:}T \vdash_b e'$ con una derivazione di altezza k. Essendo a un nome legato in $val\ a{:}T = Actor\{\beta; e\}; e'$, a meno di rinomina possiamo assumere $a \neq w$. Quindi, per ipotesi induttiva $\Gamma, a{:}T, w{:}S \vdash_a \beta$, $\Gamma, a{:}T, w{:}S \vdash_a e$, $\Gamma, a{:}T, w{:}S \vdash_b e'$ da cui si ottiene la tesi per la regola (TYPE ACTOR), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash_a \{m_1(\tilde{x}_1) \Rightarrow e_1, ..., m_k(\tilde{x}_k)\}$, $w{:}S \notin Dom(\Gamma)$, la tesi $\Gamma, w{:}S \vdash_a \{m_1(\tilde{x}_1) \Rightarrow e_1, ..., m_k(\tilde{x}_k)\}$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \ni a{:}\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle$, $\Gamma, \tilde{x}_i{:}\tilde{T}_i \vdash_a e_i\ \forall i = 1, .., k$ con una derivazione di altezza k. Da $\Gamma \ni a{:}\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle$ si ha $\Gamma, w{:}S \ni a{:}\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle$. Per ipotesi induttiva $\Gamma, \tilde{x}_i{:}\tilde{T}_i, w{:}S \vdash_a e_i\ \forall i = 1, .., k$. Da questo e da $\Gamma, w{:}S \ni a{:}\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle$ si ottiene la tesi per la regola (TYPE BEHAVE), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash [a \mapsto \emptyset]$, $w{:}S \notin Dom(\Gamma)$, la tesi $\Gamma, w{:}S \vdash [a \mapsto \emptyset]$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash a{:}T$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash a{:}T$ da cui si ottiene la tesi per la regola (TYPE EMPTY MAIL), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash [a \mapsto m(b_1, .., b_k) \cdot M]$, $w{:}S \notin Dom(\Gamma)$, la tesi $\Gamma, w{:}S \vdash [a \mapsto m(b_1, .., b_k) \cdot M]$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash a{:}\langle m(T_1, .., T_k)\rangle$, $\Gamma \vdash b_i{:}T_i\ \forall i = 1, .., k$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash a{:}\langle m(T_1, .., T_k)\rangle$, $\Gamma, w{:}S \vdash b_i{:}T_i\ \forall i = 1, .., k$ da cui si ottiene la tesi per la regola (TYPE MAILBOX), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash \underline{0}$, la tesi $\Gamma, w{:}S \vdash \underline{0}$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash \diamond$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash \diamond$ da cui si ottiene la tesi per la regola (TYPE EMPTY CONF), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash [a \mapsto M](\beta)a\{e\}$, la tesi $\Gamma, w{:}S \vdash [a \mapsto M](\beta)a\{e\}$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash a{:}T$, $\Gamma \vdash_a \beta$, $\Gamma \vdash_a e$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash a{:}T$, $\Gamma, w{:}S \vdash_a \beta$, $\Gamma, w{:}S \vdash_a e$ da cui si ottiene la tesi per la regola (TYPE CONF), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash F_1 \mid F_2$, la tesi $\Gamma, w{:}S \vdash F_1 \mid F_2$.

  Il giudizio in ipotesi è stato derivato da $\Gamma \vdash F_1$, $\Gamma \vdash F_2$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, w{:}S \vdash F_1$, $\Gamma, w{:}S \vdash F_2$ da cui si ottiene la tesi per la regola (TYPE PAR), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash (\nu a{:}T)F$, la tesi $\Gamma, w{:}S \vdash F_1 \mid F_2(\nu a{:}T)F$.

  Il giudizio in ipotesi è stato derivato da $\Gamma, a{:}T \vdash F$ con una derivazione di altezza k. Per ipotesi induttiva $\Gamma, a{:}T, w{:}S \vdash F$ da cui si ottiene la tesi per la regola (TYPE RES), con derivazione di altezza $k+1$.

- l'ipotesi è $\Gamma \vdash \mathit{val}\ a{:}T = \mathit{Actor}\{\beta, e\}; d$, la tesi $\Gamma, w{:}S \vdash \mathit{val}\ a{:}T = \mathit{Actor}\{\beta, e\}; d$.

  Il giudizio in ipotesi è stato derivato da $\Gamma, a{:}T \vdash_a \beta$, $\Gamma, a{:}T \vdash_a e$. $\Gamma, a{:}T \vdash d$ con una derivazione di altezza k. Poiché a è legato, a meno di rinomina possiamo assumere $a \neq w$ e quindi $w{:}S \notin \mathit{Dom}(\Gamma, a{:}T)$. Per ipotesi induttiva $\Gamma, a{:}T, w{:}S \vdash_a \beta$, $\Gamma, a{:}T, w{:}S \vdash_a e$. $\Gamma, a{:}T, w{:}S \vdash d$ da cui si ottiene la tesi per la regola (TYPE TOP ACTOR), con derivazione di altezza $k+1$. □

Proposizione 2 (Subject Congruence)**.**

1. *Se* $\Gamma \vdash F$ *e* $F \equiv F'$ *allora* $\Gamma \vdash F'$
2. *Se* $\Gamma \vdash F'$ *e* $F \equiv F'$ *allora* $\Gamma \vdash F$.

Dimostrazione. Per induzione simultanea sull'altezza delle derivazioni di $F \equiv F'$ e $F' \equiv F$.
Dimostriamo 1. I casi base sono quelli in cui $F \equiv F'$ ha derivazione di altezza 1 cioè viene da uno degli assiomi che definiscono la relazione $\equiv$. Distinguiamo questi casi base:

(RIFLESSIVA) in questo caso $F' = F$. La tesi è $\Gamma \vdash F$ che viene direttamente dall'ipotesi.

(ASSIOMA 1) in questo caso $F = (\nu a{:}T_1)(\nu b{:}T_2)F_1$, $F' = (\nu b{:}T_2)(\nu a{:}T_1)F_1$.

Dall'ipotesi $\Gamma \vdash F$, per il lemma di inversione, $\Gamma, a{:}T_1, b{:}T_2 \vdash F_1$. Per il lemma di permutazione $\Gamma, b{:}T_2, a{:}T_1 \vdash F_1$. La tesi si ottiene da questo giudizio applicando due volte la regola (TYPE RES).

(ASSIOMA 2) in questo caso $F = (\nu a{:}T)(F_1|F_2)$, $F' = F_1|(\nu a{:}T)F_2$ con $a \notin fn(F_1)$.

Dall'ipotesi per il lemma di inversione, si ha $\Gamma, a{:}T \vdash F_1|F_2$. Di nuovo per il lemma di inversione

(1) $\Gamma, a{:}T \vdash F_1$

(2) $\Gamma, a{:}T \vdash F_2$

Da (1) e dall'ipotesi $a \notin fn(F_1)$, per il lemma di Strengthening si ha $\Gamma \vdash F_1$. Applicando la regola (TYPE RES) a (2) si ottiene $\Gamma \vdash (\nu a{:}T)F_2$. Da quest'ultimo giudizio e da $\Gamma \vdash F_1$ si ottiene la tesi per la regola (TYPE PAR).

(ASSIOMA 3) in questo caso $F = F_1|\underline{0}$, $F' = F_1$.

Dall'ipotesi, per il lemma di inversione, si ha $\Gamma \vdash F_1$ che è anche la tesi.

(ASSIOMA 4) in questo caso $F = F_1|F_2$, $F' = F_2|F_1$.

Dall'ipotesi, per il lemma di inversione, si ha $\Gamma \vdash F_1$, $\Gamma \vdash F_2$, da cui si ottiene la tesi per la regola (TYPE PAR).

(ASSIOMA 5) in questo caso $F = (F_1|F_2)|F_3$, $F' = F_1|(F_2|F_3)$.

Dall'ipotesi, per il lemma di inversione, si ha $\Gamma \vdash F_1|F_2$, $\Gamma \vdash F_3$. Dal primo dei due giudizi, di nuovo per il lemma di inversione, si ha $\Gamma \vdash F_1$, $\Gamma \vdash F_2$. Da $\Gamma \vdash F_2$ e $\Gamma \vdash F_3$, per la regola (TYPE PAR), si ottiene $\Gamma \vdash (F_2|F_3)$. Da questa e da $\Gamma \vdash F_1$ si ottiene la tesi di nuovo per la regola (TYPE PAR).

(ASSIOMA 6) in questo caso $F = (\nu a{:}T)\underline{0}$, $F' = \underline{0}$.

Dall'ipotesi, per il lemma di inversione, si ha $\Gamma, a{:}T \vdash \underline{0}$. La tesi si ottiene da questo giudizio per il lemma di Strengthening.

I casi induttivi sono quelli in cui $F \equiv F'$ ha derivazione di altezza $k+1$. Distinguiamo i casi in base all'ultima regola usata:

(SIMMETRICA) in questo caso $F \equiv F'$ poichè $F' \equiv F$, che è derivabile con altezza k. Per ipotesi induttiva del punto 2 $\Gamma \vdash F'$ che è la tesi.

(TRANSITIVA) in questo caso $\exists F_2$ t.c. $F \equiv F_2$ e $F_2 \equiv F_1$, entrambe di altezza $\leq k$. Per ipotesi induttiva si ha $\Gamma \vdash F_2$ e da questa e da $F_2 \equiv F'$, ancora per ipotesi induttiva, si ha $\Gamma \vdash F'$.

Dimostriamo 2. I casi base sono quelli per cui $F' \equiv F$ viene da assioma.

(RIFLESSIVA) in questo caso $F = F'$ e la tesi viene direttamente dall'ipotesi

(ASSIOMA 1) in questo caso $F = (\nu b : T_2)(\nu a{:}T_1)F_1$ e $F' = (\nu a{:}T_1)(\nu b{:}T_2)F_1$.

Dall'ipotesi, per il lemma di inversione, si ha $\Gamma, b{:}T_2, a{:}T_1 \vdash F_1$. Da questo, per il lemma di permutazione si ottiene $\Gamma, a{:}T_1, b{:}T_2 \vdash F_1$. Da quest'ultimo giudizio si ottiene la tesi applicando due volte la regola (TYPE REF).

(ASSIOMA 2) in questo caso $F = F_1|(\nu a{:}T)F_2$ con $a \notin fn(F_1)$, $F' = (\nu a{:}T)(F_1|F_2)$.

Dall'ipotesi, per il lemma di inversione, si ottiene

(1) $\Gamma \vdash F_1$

(2) $\Gamma \vdash (\nu a{:}T)F_2$

Da (2), per il lemma di inversione, si ha $\Gamma, a{:}T \vdash F_2$. Sempre da (2) si ha $\Gamma, a{:}T \vdash \diamond$. Da (1) e da quest'ultimo giudizio, per il lemma di weakening, si ottiene $\Gamma, a{:}T \vdash F_1$. Da questi e da $\Gamma, a{:}T \vdash F_2$, per la regola (PAR), si ha $\Gamma, a{:}T \vdash F_1|F_2$, da cui si ottiene la tesi per la regola (PAR).

(ASSIOMA 3) in questo caso $F = F_1$, $F' = F_1|\underline{0}$.

Dall'ipotesi si ha $\Gamma \vdash \diamond$ per il lemma 2. Da questo, per la regola (EMPTY EXPR) si ha $\Gamma \vdash \underline{0}$. Da quest'ultimo giudizio e dall'ipotesi, si ottiene la tesi per la regola (PAR).

(ASSIOMA 4) in questo caso $F = F_2|F_1$, $F' = F_1|F_2$.

Dall'ipotesi, per il lemma di inversione, si ha $\Gamma \vdash F_2$, $\Gamma \vdash F_1$, da cui si ottiene la tesi per la regola (PAR).

(ASSIOMA 5) in questo caso $F = F_1|(F_2|F_3)$, $F' = (F_1|F_2)|F_3$.

Dall'ipotesi, per il lemma di inversione, si ha $\Gamma \vdash F_1$, $\Gamma \vdash F_2|F_3$. Applicando il lemma di inversione a quest'ultimo giudizio si ottiene $\Gamma \vdash F_2$, $\Gamma \vdash F_3$. Da $\Gamma \vdash F_1$ e $\Gamma \vdash F_2$, per la regola (PAR), si ha $\Gamma \vdash F_1|F_2$. Da questo e da $\Gamma \vdash F_3$ si ottiene la tesi per la regola (PAR).

(ASSIOMA 6) in questo caso $F = \underline{0}$, $F' = (\nu a{:}T)\underline{0}$.

Se $a{:}T \in Dom(\Gamma)$ allora la tesi viene direttamente per la regola (TYPE RES). Se $a \notin Dom(\Gamma)$, da questo e dall'ipotesi, per il lemma di weakening si ottiene $\Gamma, a{:}T \vdash \underline{0}$. Da cui si ottiene la tesi per la regola (TYPE RES).

I casi induttivi sono quelli in cui $F' \equiv F$ ha derivazione di altezza $k+1$.

(SIMMETRICA) in questo caso $F' \equiv F$ poiché $F \equiv F'$, che è derivabile con altezza k. La tesi si ottiene dall'ipotesi induttiva del punto 1.

(TRANSITIVA) in questo caso $\exists F_2$ tale che $F' \equiv F_2$ e $F_2 \equiv F$, entrambi di altezza $\leq k$. Per ipotesi induttiva si ha $\Gamma \vdash F_2$. Da questa e da $F_2 \equiv F$, si ottiene la tesi ancora per ipotesi induttiva. □

Lemma 6. *Se* $\Gamma \vdash [a \mapsto M]$, $\Gamma \vdash a{:}\langle m(\tilde{T}_c)\rangle$, $\Gamma \vdash \tilde{c}{:}\tilde{T}_c$ *allora* $\Gamma \vdash [a \mapsto M \cdot m(\tilde{c})]$

Dimostrazione. Per induzione sulla lunghezza di M.

- $(M = \emptyset)$ l'ipotesi è $\Gamma \vdash [a \mapsto \emptyset]$, $\Gamma \vdash a{:}\langle m(\tilde{T}_c)\rangle$, $\Gamma \vdash \tilde{c}{:}\tilde{T}_c$. La tesi si deriva direttamente dalle ipotesi per la regola (TYPE MAILBOX).
- $(M = m'(\tilde{b}) \cdot M')$ l'ipotesi è $\Gamma \vdash [a \mapsto m'(\tilde{b}) \cdot M']$, $\Gamma \vdash a{:}\langle m(\tilde{T}_c)\rangle$, $\Gamma \vdash \tilde{c}{:}\tilde{T}_c$. La tesi $\Gamma \vdash [a \mapsto m'(\tilde{b}), M', m(\tilde{c})]$.

  Dall'ipotesi $\Gamma \vdash [a \mapsto m'(b_1, .., b_k) \cdot M']$, per il lemma di inversione, $\exists T_1, .., T_k$ tali che $\Gamma \vdash a{:}\langle m'(T_1, .., T_k)\rangle$, $\Gamma \vdash b_i{:}T_i \quad \forall i = 1, .., k$, $\Gamma \vdash [a \mapsto M']$. Per ipotesi induttiva $\Gamma \vdash [a \mapsto M' \cdot m(\tilde{c})]$. Da questo e dalle ipotesi $\Gamma \vdash a{:}\langle m'(T_1, .., T_k)\rangle$, $\Gamma \vdash b_i{:}T_i \quad \forall i = 1, .., k$, si ha tesi per la regola (TYPE MAILBOX). □

Teorema 1 (Subject Reduction)**.** *Se* $\Gamma \vdash F$ *e* $F \longrightarrow F'$ *allora* $\Gamma \vdash F'$

Dimostrazione. Per induzione sull'altezza della derivazione $F \longrightarrow F'$. I casi base sono quelli per cui $F \longrightarrow F'$ viene da assioma:

- caso in cui $F = (val\ a{:}T = Actor\{\beta; e\}; d)$, $F' = (\nu a{:}T)([a \mapsto \emptyset](\beta)a\{e\} \mid d)$. Dall'ipotesi $\Gamma \vdash F$, per il lemma di inversione, si ha:

  (1) $\Gamma, a{:}T \vdash_a \beta$

  (2) $\Gamma, a{:}T \vdash_a e$

(3) $\Gamma, a{:}T \vdash d$

da questi giudizi, applicando più volte il lemma di inversione, abbiamo $\Gamma, a{:}T \vdash \diamond$, da cui, per la regola (PROJECT), deriviamo $\Gamma, a{:}T \vdash a{:}T$. Da quest'ultimo giudizio, per la regola (TYPE EMPTY MAIL), otteniamo:

(4) $\Gamma, a{:}T \vdash [a \mapsto \emptyset]$

Da (1),(2),(4), per la regola (TYPE CONF) deriviamo:

$$\Gamma, a{:}T \vdash [a \mapsto \emptyset](\beta)a\{e\}$$

Da quest'ultimo giudizio e da (3) si ottiene, per la regola (TYPE PAR): $\Gamma, a{:}T \vdash [a \mapsto \emptyset](\beta)a\{e\} \mid d$, da cui per la regola (TYPE RES) ricaviamo:

$$\Gamma \vdash (\nu a{:}T)([a \mapsto \emptyset](\beta)a\{e\} \mid d)$$

- caso in cui $F = [b \mapsto M](\beta')b\{val\ a{:}T = Actor\{\beta; e\}; e'\}$, $F' = (\nu a{:}T)([b \mapsto M](\beta')b\{e'\} \mid [a \mapsto \emptyset](\beta)a\{e\})$, con $a \notin (fn(\beta') \cup fn(M))$. Dall'ipotesi $\Gamma \vdash F$, per il lemma di inversione, si ha

$$(a)\ \Gamma \vdash [b \mapsto M], \quad (b)\ \Gamma \vdash_b \beta', \quad (c)\ \Gamma \vdash_b val\ a{:}T = Actor\{\beta; e\}; e'$$

in particolare dall'ultimo giudizio, per il lemma di inversione, si ha

(1) $\Gamma, a{:}T \vdash_a \beta$

(2) $\Gamma, a{:}T \vdash_a e$

(3) $\Gamma, a{:}T \vdash_b e'$

da (1),(2),(3) applicando più volte il lemma di inversione, si ha $\Gamma, a{:}T \vdash \diamond$. Per la regola (PROJECT) allora si ha $\Gamma, a{:}T \vdash a{:}T$, da cui si deriva, per (TYPE EMPTY MAIL):

(4) $\Gamma, a{:}T \vdash [a \mapsto \emptyset]$

da $\Gamma, a{:}T \vdash \diamond$, per il lemma di inversione si ha $\Gamma \vdash \diamond$, $a \notin Dom(\Gamma)$. Da (a) e (b) possiamo quindi scrivere, per il lemma di weakening:

(5) $\Gamma, a{:}T \vdash [b \mapsto M]$

(6) $\Gamma, a{:}T \vdash_b \beta'$

Da (3),(5),(6) per la regola (TYPE CONF) otteniamo (i) $\Gamma, a{:}T \vdash [b \mapsto M](\beta')b\{e'\}$, mentre da (1),(2),(4), per la stessa regola, abbiamo (ii) $\Gamma, a{:}T \vdash [a \mapsto \emptyset](\beta)a\{e\}$. Da (i) e (ii), per la regola (TYPE PAR) otteniamo:

$$\Gamma, a{:}T \vdash [b \mapsto M](\beta')b\{e'\} \mid [a \mapsto \emptyset](\beta)a\{e\}$$

da cui, per la regola (TYPE RES) ricaviamo:

$$\Gamma \vdash (\nu a{:}T)([b \mapsto M](\beta')b\{e'\} \mid [a \mapsto \emptyset](\beta)a\{e\}) = F'$$

- caso in cui $F = [a \mapsto M](\beta)a\{become(\beta'); e\}$, $F' = [a \mapsto M](\beta')a\{e\}$. Dall'ipotesi $\Gamma \vdash F$ si ha, per il lemma di inversione:

  (1) $\Gamma \vdash [a \mapsto M]$

  (2) $\Gamma \vdash_a become(\beta'); e$

  Da (2), per il lemma di inversione, otteniamo:

  (3) $\Gamma \vdash_a \beta'$

  (4) $\Gamma \vdash_a e$

  Da (1),(3),(4) per la regola (TYPE CONF) deriviamo $\Gamma \vdash [a \mapsto M](\beta')a\{e\} = F'$

- caso in cui $F = [a \mapsto M](\beta)a\{e\} \mid [b \mapsto M'](\beta')b\{a \,!\, m(\tilde{c}); e'\}$, $F' = [a \mapsto M \cdot m(\tilde{c})](\beta)a\{e\} \mid [b \mapsto M'](\beta')b\{e'\}$. Dall'ipotesi $\Gamma \vdash F$, per il lemma di inversione, abbiamo (i) $\Gamma \vdash [a \mapsto M](\beta)a\{e\}$ e (ii) $\Gamma \vdash [b \mapsto M'](\beta')b\{a \,!\, m(\tilde{c}); e'\}$. Da (i), sempre per il lemma di inversione, otteniamo:

  (1) $\Gamma \vdash [a \mapsto M]$

  (2) $\Gamma \vdash_a \beta$

  (3) $\Gamma \vdash_a e$

  Da (ii), per il lemma di inversione, deriviamo:

  (4) $\Gamma \vdash [b \mapsto M']$

  (5) $\Gamma \vdash_b \beta'$

  (6) $\Gamma \vdash_b a \,!\, m(c_1, .., c_k); e'$

  Da (6), per il lemma di inversione, ricaviamo:

  (7) $\exists T_1, .., T_k$ tali che $\Gamma \vdash a{:}\langle m(T_1, .., T_k)\rangle$, $\Gamma \vdash c_i : T_i \; \forall i = 1, .., k$

  (8) $\Gamma \vdash_b e'$

  Da (7) e (1) ricaviamo, per il lemma 6, $\Gamma \vdash [a \mapsto M \cdot m(\tilde{c})]$. Da quest'ultima, insieme con (2) e (3) si ha $\Gamma \vdash [a \mapsto M \cdot m(\tilde{c})](\beta)a\{e\}$ per la regola (TYPE CONF).
  Da (8), (4), (5) si ricava, sempre per la regola (TYPE CONF) $[b \mapsto M'](\beta')b\{e'\}$.
  Da questi ultimi due giudizi, per la regola (TYPE PAR), si ricava:

$$\Gamma \vdash [a \mapsto M \cdot m(\tilde{c})](\beta)a\{e\} \mid [b \mapsto M'](\beta')b\{a!m(\tilde{c}); e'\} = F'$$

- caso in cui $F = [a \mapsto m_j(\tilde{c}) \cdot M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I})a\{\underline{0}\}$, $F' = [a \mapsto M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I})a\{e_j\{\tilde{c}/\tilde{x}_j\}\}$ con $j \in I$, $I = 1, .., k$. Dall'ipotesi $\Gamma \vdash F$, per il lemma di inversione, si ha:

  (1) $\Gamma \vdash [a \mapsto m_j(\tilde{c}) \cdot M]$

  (2) $\Gamma \vdash_a \{m_1(\tilde{x}_1) \Rightarrow e_1, .., m_k(\tilde{x}_k) \Rightarrow e_k\}$

  Da (1), per il lemma di inversione, $\exists \tilde{S}_j$ tale che $\Gamma \vdash a{:}\langle m_j(\tilde{S}_j)\rangle$, (i) $\Gamma \vdash \tilde{c}{:}\tilde{S}_j$. Da (2), per il lemma di inversione, si ottiene:

  $$(*)\ \Gamma \ni a{:}\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle, \quad (**)\ \Gamma, \tilde{x}_i{:}\tilde{T}_i \vdash_a e_i\ \forall i = 1, .., k$$

  da $\Gamma \vdash a{:}\langle m_j(\tilde{S}_j)\rangle$ poiché $\Gamma \ni a{:}\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle$ si ha, per il lemma di inversione, $\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle <: \langle m_j(\tilde{S}_j)\rangle$ ovvero $j \in I$ e $\tilde{S}_j <: \tilde{T}_j$. Da questo e da (i), per (SUBSUMPTION), abbiamo $\Gamma \vdash \tilde{c}{:}\tilde{T}_j$. Da questo giudizio, insieme con $\Gamma, \tilde{x}_j{:}\tilde{T}_j \vdash_a e_j$ che viene da (**), per il lemma di sostituzione, otteniamo:

  (3) $\Gamma \vdash_a e_j\{\tilde{c}/\tilde{x}_j\}$

  Da (1), per il lemma di inversione, otteniamo anche

  (4) $\Gamma \vdash [a \mapsto M]$

  Da (4),(3),(2), per la regola (TYPE CONF), otteniamo

  $$\Gamma \vdash [a \mapsto M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I})a\{e_j\{\tilde{c}/\tilde{x}_j\}\} = F'$$

- caso in cui $F = [a \mapsto m_j(\tilde{c}) \cdot M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I})a\{\underline{0}\}$ con $j \notin I$, $I = 1, .., k$, $F' = [a \mapsto M](\{m_i(\tilde{x}_i) \Rightarrow e_i\}_{i\in I})a\{\underline{0}\}$. Dall'ipotesi $\Gamma \vdash F$, per il lemma di inversione, si ha:

  (1) $\Gamma \vdash [a \mapsto m_j(\tilde{c}) \cdot M]$

  (2) $\Gamma \vdash_a \{m_1(\tilde{x}_1) \Rightarrow e_1, .., m_k(\tilde{x}_k) \Rightarrow e_k\}$

  Da (1), per il lemma di inversione, $\exists \tilde{S}_j$ tale che $\Gamma \vdash a{:}\langle m_j(\tilde{S}_j)\rangle$, (i) $\Gamma \vdash \tilde{c}{:}\tilde{S}_j$. Da (2), per il lemma di inversione, si ottiene:

  $$(*)\ \Gamma \ni a{:}\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle, \quad (**)\ \Gamma, \tilde{x}_i{:}\tilde{T}_i \vdash_a e_i\ \forall i = 1, .., k$$

  da $\Gamma \vdash a{:}\langle m_j(\tilde{S}_j)\rangle$ poiché $\Gamma \ni a{:}\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle$ si ha, per il lemma di inversione, $\langle m_1(\tilde{T}_1), ..., m_k(\tilde{T}_k)\rangle <: \langle m_j(\tilde{S}_j)\rangle$ ovvero $j \in I$. Pertanto questo è un caso vacuo.

I casi induttivi sono quelli in cui $F \longrightarrow F'$ è stato derivato con una derivazione di altezza $k + 1$. Distinguiamo i casi a seconda dell'ultima regola usata:

- caso in cui $F = F_1 \mid F_2$ e $F' = F_1' \mid F_2$, poichè $F_1 \longrightarrow F_1'$ derivato con un'altezza k. Dall'ipotesi $\Gamma \vdash F$, per il lemma di inversione, otteniamo:

  (1) $\Gamma \vdash F_1$

  (2) $\Gamma \vdash F_2$

  Valendo (1), per ipotesi induttiva, si ha $\Gamma \vdash F_1'$. Da quest'ultimo giudizio e da (2), per la regola (TYPE PAR), si ha:

  $$\Gamma \vdash F_1' \mid F_2 = F'$$

- caso in cui $F = (\nu a{:}T)F_1$, $F' = (\nu a{:}T)F_1'$, poiché $F_1 \longrightarrow F_1'$ derivato con un'altezza k. Dall'ipotesi $\Gamma \vdash F$, per il lemma di inversione, si ha $\Gamma, a : T \vdash F_1$. Quindi, per ipotesi induttiva, $\Gamma, a{:}T \vdash F_1'$. Da quest'ultimo giudizio, per la regola (TYPE RES), si deriva $\Gamma \vdash (\nu a{:}T)F_1' = F'$

- caso in cui $F = F_1$, $F' = F_4$, poiché $F_2 \longrightarrow F_3$ con $F_1 \equiv F_2$ e $F_2 \equiv F_3$, derivato con un'altezza k. Dalle ipotesi $\Gamma \vdash F = F_1$ e $F_1 \equiv F_2$, per la subject congruence, si ha $\Gamma \vdash F_2$. Da quest'ultimo giudizio, per ipotesi induttiva, si ottiene $\Gamma \vdash F_3$. Da questo e dall'ipotesi $F_3 \equiv F_4$, di nuovo per la subject congruence, si ottiene $\Gamma \vdash F_4 = F'$. □

Corollario 1. *Se* $\emptyset \vdash F$ *e* $F \longrightarrow^* F'$ *allora* $\emptyset \vdash F'$

Dimostrazione. Per induzione sulla lunghezza di $F \longrightarrow^* F'$. Se la lunghezza è 0 allora $F = F'$ e la tesi è immediata. Sia invece $F \longrightarrow^* F'$ in $k+1$ passi, cioè $F \longrightarrow^* F_1$ in k passi e $F_1 \longrightarrow F'$. Da $\emptyset \vdash F$ per ipotesi induttiva si ha $\emptyset \vdash F_1$ e per il terorema subject reduction si conclude $\emptyset \vdash F'$. □

Indichiamo con P un programma sorgente, cioè un'espressione chiusa della forma $val\ a{:}T = Actor\{\beta; e\}; d$. Inoltre, data la riduzione $F \longrightarrow F'$, indichiamo con $\Re_{F \longrightarrow^* F'}$ l'insieme di regole applicate per ottenere F' a partire da F.

Teorema 2 (Soundness)**.** *Se* $\emptyset \vdash P$ *e* $P \longrightarrow^* F$ *allora* $(JUNK) \notin \Re_{P \longrightarrow^* F}$

Dimostrazione. Procediamo per induzione sulla lunghezza l della riduzione:

- $(l = 0)$ $F = P$. $\Re = \emptyset$ quindi la tesi è verificata.

- $(l \geq 1)$ $P \longrightarrow^{(l-1)} F' \longrightarrow F$ con $F' \equiv (\nu a_1{:}T_1)...(\nu a_n{:}T_n)(\Pi_{i=1,..,n}[a_i \mapsto M_i](\beta_i)a_i\{e_i\} \mid d)$. Inoltre $\Re_{P \longrightarrow^{(l)} F} = \Re_{P \longrightarrow^{(l-1)} F'} \cup \Re_{F' \longrightarrow F}$.

  Per il corollario 1, $\emptyset \vdash F'$. Inoltre, per ipotesi induttiva $(JUNK) \notin \Re_{P \longrightarrow^{(l-1)} F'}$.

  Dimostro che $(JUNK) \notin \Re_{F' \longrightarrow F}$. Da $\emptyset \vdash F'$, per il lemma di inversione, si ottiene $\forall a \in a_1, .., a_n$ (chiamando $\Gamma = a_1{:}T_1, .., a_n{:}T_n$):

(1) $\Gamma \vdash [a \mapsto M]$ con $M = m_1(\tilde{c}_1), .., m_t(\tilde{c}_t)$

(2) $\Gamma \vdash_a \beta$ con $\beta = \{m_1(\tilde{x}_1) \Rightarrow e_1, .., m_k(\tilde{x}_k) \Rightarrow e_k\}$

(3) $\Gamma \vdash_a e$

Da (2), per il lemma di inversione, ricaviamo $\Gamma \ni a : \langle m_1(\tilde{T}_1), .., m_k(\tilde{T}_k)\rangle$ per $\exists T_1, .., T_k$. Da (1) abbiamo che la mailbox è ben tipata quindi, per il lemma di inversione, $\exists \tilde{S}_j$ tali che $\Gamma \vdash a{:}\langle m_j(\tilde{S}_j)\rangle \ \forall j = 1, .., t$. Per il lemma di inversione, poiché $a : \langle m_1(\tilde{T}_1), .., m_k(\tilde{T}_k)\rangle \in \Gamma$ si ha $\langle m_1(\tilde{T}_1), .., m_k(\tilde{T}_k)\rangle <: \langle m_j(\tilde{S}_j)\rangle \forall j = 1, .., t$. Cioè $j \in \{1, .., k\} \quad \forall j = 1, .., t$. Questo significa che tutti i messaggi nelle mailbox sono gestibili dai corrispondenti attori, ovvero la condizione della regola (JUNK) non è soddisfatta per nessun attore attivo nella configurazione attuale e pertanto $(JUNK) \notin \Re_{F' \longrightarrow F}$. Da questo e dall'ipotesi induttiva si ottiene la tesi $(JUNK) \notin \Re_{P \longrightarrow^{(l)} F}$. □

4.3 Esempi

4.3.1 Lock usa e getta

Riprendiamo l'esempio del lock già visto al capitolo 2. Il lock che abbiamo analizzato ha una sintassi ricorsiva che richiede un tipo ricorsivo che ci riserviamo di trattare come lavoro futuro, ma che non viene trattato in questo elaborato. Studiamo quindi a fondo un lock usa e getta finito che possa essere utilizzato una sola volta. Vedremo poi in seguito come generalizzarlo.

Scriviamo un programma P in A_T che modelli tale lock.

$$
\begin{aligned}
P = \quad & val\ L{:}T_L = Actor\{\beta_{FREE}; \underline{0}\};\\
& val\ C{:}T_C = Actor\{\beta_C; L\ !\ acquire(C); \underline{0}\};\\
& \underline{0}
\end{aligned}
$$

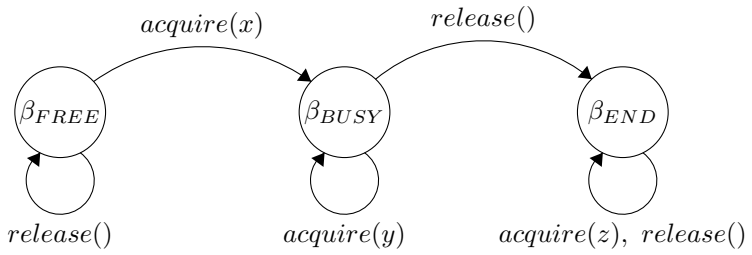
$$
\begin{aligned}
\beta_{FREE} = \{ & acquire(x) \Rightarrow x\ !\ here(L); become(\beta_{BUSY}); \underline{0}\\
& release() \Rightarrow \underline{0}\}
\end{aligned}
$$

$$
\begin{aligned}
\beta_{BUSY} = \{ & acquire(y) \Rightarrow \underline{0}\\
& release() \Rightarrow become(\beta_{END}); \underline{0}\}
\end{aligned}
$$

$$
\begin{aligned}
\beta_{END} = \{ & acquire(z) \Rightarrow \underline{0}\\
& release() \Rightarrow \underline{0}\}
\end{aligned}
$$

$$\beta_C = \{here(x') \Rightarrow release()\}$$

Il lock può acquisire tre comportamenti: inizialmente si trova in uno stato di lock non acquisito (comportamento β_{FREE}); in seguito ad una richiesta passa ad uno stato di lock acquisito (comportamento β_{BUSY}); infine quando viene rilasciato passa ad uno stato di lock terminato (comportamento β_{END}).



Notiamo che tutti e tre i comportamenti assunti dal lock gestiscono sia messaggi *acquire* che messaggi *release*. Questo in accordo con Akka Typed, in cui tutti i comportamenti assunti da un attore devono avere lo stesso tipo. Vedremo in seguito che questo è un requisito fondamentale per poter garantire staticamente che tutti i messaggi inviati a quel determinato attore saranno ricevuti, ovvero perchè valga il teorema di soundeness dimostrato sopra.

Il lock può essere acquisito o rilasciato da un client. Il tipo che ci aspettiamo per il lock è quindi:

$$T_L = \langle acquire(T_C), release() \rangle$$

dove T_C è il tipo del client.

Un client può ricevere un lock, ma vogliamo che sul lock acquisito possa eseguire solo una *release*. Pertanto vogliamo che il client abbia tipo:

$$T_C = \langle here(\langle release() \rangle) \rangle$$

Mostriamo la derivazione di tipo per questo programma e dimostriamo che, assegnando al lock e al client i tipi T_L e T_C indicati, il programma è ben tipato secondo le regole di tipo date. Se il programma è ben tipato, per il teorema di soundness, l'esecuzione non prevederà mai l'uso della regola (JUNK).

Derivazione

Consideriamo $d = val\ C{:}T_C = Actor\{\beta_C; L\ !\ acquire(C)\}; \underline{0}$

$$\dfrac{\dfrac{*}{L{:}T_L \vdash_L \beta_{FREE}} \qquad \dfrac{\dfrac{\dfrac{\checkmark}{\emptyset \vdash \diamond}\,(E\ CTX) \qquad L \notin Dom(\emptyset)}{L{:}T_L \vdash \diamond}\,(CTX)}{L{:}T_L \vdash_L \underline{0}}\,(E\ EXP) \qquad \dfrac{**}{L{:}T_L \vdash d}}{\emptyset \vdash val\ L{:}T_L = Actor\{\beta_{FREE}; \underline{0}\}; d}\,(T\ TOP\ ACTOR)$$

Proseguiamo con la derivazione di $*$. Definiamo $e = x\ !\ here(L); become(\beta_{BUSY}); \underline{0}$.

$$\dfrac{\dfrac{\checkmark}{L{:}T_L \ni L{:}\langle acquire(T_C), release()\rangle} \quad \dfrac{\bullet\bullet}{L{:}T_L, x : T_C \vdash_L e} \quad \dfrac{\bullet\bullet\bullet}{L{:}T_L \vdash_L \underline{0}}}{L{:}T_L \vdash_L \{acquire(x) \Rightarrow e \quad release() \Rightarrow \underline{0}\}}\,(T\ BEHAVE)$$

Chiudiamo la derivazione di $\bullet\bullet\bullet$.

$$\dfrac{\dfrac{\dfrac{\checkmark}{\emptyset \vdash \diamond}\,(E\ CTX) \quad L \notin Dom(\emptyset)}{L{:}T_L \vdash \diamond}\,(CTX)}{L{:}T_L \vdash_L \underline{0}}\,(E\ EXP)$$

E proseguiamo con la derivazione di $\bullet\bullet$. Definiamo $\Gamma = L{:}T_L, x{:}T_C$

$$\dfrac{\dfrac{\circ}{\Gamma \vdash x : \langle here(T)\rangle \quad T = \langle release()\rangle} \quad \dfrac{\circ\circ}{\Gamma \vdash L : \langle release()\rangle} \quad \dfrac{\circ\circ\circ}{\Gamma \vdash_L become(\beta_{BUSY}); \underline{0}}}{\Gamma \vdash_L x\ !\ here(L); become(\beta_{BUSY}); \underline{0}}\,(T\ SEND)$$

Chiudiamo le derivazioni di $\circ$ e $\circ\circ$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\emptyset \vdash \diamond}\,(E\ CTX) \quad L \notin Dom(\emptyset)}{L{:}T_L \vdash \diamond}\,(CTX) \quad x{:}T_C \notin Dom(L{:}T_L)}{\Gamma \vdash \diamond}\,(CTX)}{\Gamma \vdash x : \langle here(\langle release()\rangle)\rangle}\,(PROJECT)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\emptyset \vdash \diamond}\,(E\ CTX) \quad L \notin Dom(\emptyset)}{L{:}T_L \vdash \diamond}\,(CTX) \quad x{:}T_C \notin Dom(L{:}T_L)}{\Gamma \vdash \diamond}\,(CTX)}{\Gamma \vdash L{:}T_L}\,(PROJECT) \quad \dfrac{\checkmark}{T_L <: \langle release()\rangle}\,(SUB\ W)}{\Gamma \vdash L{:}\langle release()\rangle}\,(SUBSUMPTION)$$

Proseguiamo ora con la derivazione di $\circ\circ\circ$. Abbiamo già mostrato nelle derivazioni precedenti sia $L{:}T_L \vdash \diamond$ che $\Gamma \vdash \diamond$ quindi da adesso in poi non ne daremo nuovamente la derivazione, ma ci limiteremo a chiudere la derivazione quando incontriamo questi tipi di giudizio.

$$\dfrac{\dfrac{\triangle}{\Gamma \vdash_L \beta_{BUSY}} \quad \dfrac{\dfrac{\checkmark}{\Gamma \vdash \diamond}}{\Gamma \vdash_L \underline{0}}\,(E\ EXP)}{\Gamma \vdash_L become(\beta_{BUSY}); \underline{0}}\,(T\ BECOME)$$

Proseguiamo con la derivazione di $\triangle$. Definiamo $e' = become(\beta_{END}); \underline{0}$.

$$\dfrac{\dfrac{\checkmark}{\Gamma \ni L{:}\langle acquire(T_C), release()\rangle}\,(PRJ) \qquad \dfrac{\dfrac{\dfrac{\checkmark}{\Gamma \vdash \diamond} \quad y \notin Dom(L{:}T_L, x{:}T_C)}{\Gamma, y{:}T_C \vdash \diamond}\,(CTX)}{\Gamma, y : T_C \vdash_L \underline{0}}\,(E\ EXP) \qquad \dfrac{\triangle\triangle}{\Gamma \vdash_L e'}}{\Gamma \vdash_L \{acquire(y) \Rightarrow \underline{0} \quad release() \Rightarrow e'\}}\,(T\ BEHAVE)$$

Deriviamo $\triangle\triangle$.

$$\dfrac{\dfrac{\blacktriangleright}{\Gamma \vdash_L \beta_{END}} \quad \dfrac{\dfrac{\checkmark}{\Gamma \vdash \diamond}}{\Gamma \vdash_L \underline{0}}\,(E\ EXP)}{\Gamma \vdash_L become(\beta_{END}); \underline{0}}\,(T\ BECOME)$$

Concludiamo con la derivazione di $\blacktriangleright$.

$$\dfrac{\dfrac{}{\Gamma \ni L{:}\langle acquire(T_C), release()\rangle}\,(PJ) \qquad \dfrac{\dfrac{\dfrac{\checkmark}{\Gamma \vdash \diamond} \quad z \notin Dom(\Gamma)}{\Gamma, z{:}T_C \vdash \diamond}\,(CTX)}{\Gamma, z{:}T_C \vdash \underline{0}}\,(E\ EXP) \qquad \dfrac{\dfrac{\checkmark}{\Gamma \vdash \diamond}}{\Gamma \vdash \underline{0}}\,(E\ EXP)}{\Gamma \vdash_L \{acquire(z) \Rightarrow \underline{0} \quad release() \Rightarrow \underline{0}\}}\,(T\ BEHAVE)$$

Infine mostriamo la derivazione di $**$. Definiamo $\Gamma' = L{:}T_L, C{:}T_C$.

$$\dfrac{\dfrac{\circledast}{\Gamma' \vdash_C \beta_C} \quad \dfrac{\circledast\circledast}{\Gamma' \vdash_C L\ !\ acquire(C); \underline{0}} \qquad \dfrac{\dfrac{\dfrac{\checkmark}{L{:}T_L \vdash \diamond} \quad C \notin Dom(L{:}T_L)}{\Gamma' \vdash \diamond}\,(CTX)}{\Gamma' \vdash \underline{0}}\,(E\ EXP)}{L{:}T_L \vdash val\ C{:}T_C = Actor\{\beta_C; L\ !\ acquire(C); \underline{0}\}; \underline{0}}\,(T\ TOP\ ACTOR)$$

Chiudiamo le derivazione di $\circledast$.

$$\frac{\dfrac{}{\Gamma' \ni C{:}\langle here(T'_x)\rangle \quad T'_x = \langle release()\rangle}\checkmark \qquad \dfrac{}{\Gamma', x' : \langle release()\rangle \vdash x' \,!\, release(); \underline{0}}\boxdot}{\Gamma' \vdash_C \{here(x') \Rightarrow x' \,!\, release(); \underline{0}\}}\,(T\ BEHAVE)$$

con $\boxdot$ con derivazione:

$$\frac{\dfrac{\dfrac{}{\Gamma' \vdash \diamond}\checkmark \quad x' \notin Dom(\Gamma')}{\Gamma', x' : \langle release()\rangle \vdash \diamond}\,(CTX)}{\Gamma', x' : \langle release()\rangle \vdash x' : \langle release()\rangle}\,(PJ) \qquad \frac{\dfrac{}{\Gamma', x' : \langle release()\rangle \vdash \diamond}\checkmark}{\Gamma', x' : \langle release()\rangle \vdash_C \underline{0}}\,(E\ EXP)$$
$$\Gamma', x' : \langle release()\rangle \vdash x' \,!\, release(); \underline{0}$$

Infine chiudiamo la derivazione di $\circledast\circledast$.

$$\frac{\dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \diamond}\checkmark}{\Gamma' \vdash L{:}T_L}\,(PRJ) \qquad \dfrac{}{T_L <: \langle acquire(T_C)\rangle}\checkmark\,(SUB\ W)}{\Gamma' \vdash L{:}\langle acquire(T_C)\rangle}\,(SUBSUMP) \qquad \dfrac{\dfrac{}{\Gamma' \vdash \diamond}\checkmark}{\Gamma' \vdash C : T_C}\,(PRJ) \qquad \dfrac{\dfrac{}{\Gamma' \vdash \diamond}\checkmark}{\Gamma' \vdash_C \underline{0}}\,(E\ EXP)}{\Gamma' \vdash_C L \,!\, acquire(C); \underline{0}}\,(T\ SEND)$$

Possiamo quindi affermare che il programma è ben tipato utilizzando i tipi T_L e T_C dati. Per il teorema di soundness questo programma, durante l'esecuzione, non applicherà mai la regola (JUNK).

Codice Akka Typed

Riscriviamo lo stesso programma utilizzando Akka Typed.

```
1  import akka.typed._
2  import akka.typed.ScaladSL._
3
4
5  object TypedOnceLock extends App{
6
7  sealed trait LockMsg
8    case class Acquire(client: ActorRef[ClientMsg]) extends LockMsg
9    case class Release(client: ActorRef[ClientMsg]) extends LockMsg
10
```

```
11 sealed trait ClientMsg
12   case class Here(lock: ActorRef[Release]) extends ClientMsg
13
14 def bBusy(who: ActorRef[ClientMsg]): Behavior[LockMsg] =
15   ContextAware[LockMsg] { ctx =>
16   Total[LockMsg] {
17   case Acquire(client) =>
18     println(client.path.name+" non ottiene il lock ")
19     Same
20   case Release(client) if client == who =>
21     println(client.path.name+" rilascia il lock ")
22     bEnd
23   case Release(client) if client != who =>
24     println(client.path.name+" cerca di rilasciare un lock che non ha ")
25     Same
26   }
27 }
28
29 def bEnd(): Behavior[LockMsg] =
30   ContextAware[LockMsg] { ctx =>
31   Total[LockMsg] {
32   case Acquire(client) =>
33     println(client.path.name+" non ottiene il lock")
34     Same
35   case Release(client)  =>
36     println(client.path.name+" cerca di rilasciare un lock che non ha")
37     Same
38   }
39 }
40
41 val bFree : Behavior[LockMsg] = ContextAware[LockMsg] { ctx =>
42   Total[LockMsg] {
43   case Acquire(client) =>
44     println(client.path.name+" acquisisce il lock ")
45     client ! Here(ctx.self)
46     bBusy(client)
47   case Release(client) =>
48     println(client.path.name+" cerca di rilasciare un lock che non ha ")
49     Same
50 }
51 }
52
53 val bClient = ContextAware[ClientMsg] { ctx =>
54   Total[ClientMsg] {
55   case Here(lock) =>
```

```
56      println(ctx.self.path.name+" inizia a lavorare ...")
57      Thread.sleep(3000)
58      println("... " +ctx.self.path.name+" finisce di lavorare")
59      lock ! Release(ctx.self)
60      Same
61  }
62  }
63
64  case class Spawn(name: String, l:ActorRef[Acquire])
65
66  val bGenerator:Behavior[Spawn] = ContextAware[Spawn] { ctx =>
67    Static[Spawn] {
68      case Spawn(name: String, l:ActorRef[Acquire]) =>
69        val c = ctx.spawn(Props(bClient), name)
70        println(name+ " tenta di acquisire il lock")
71        l ! Acquire(c)
72    }
73  }
74
75  val lock : ActorRef[Acquire] = ActorSystem("Lock", Props(bFree))
76  val generator = ActorSystem("Generator", Props(bGenerator))
77
78  generator ! Spawn("Alice", lock)
79  generator ! Spawn("Bob", lock)
80  generator ! Spawn("Eve", lock)
81
82  }
```

Facciamo alcune considerazioni sul codice. Notiamo come prima cosa che i tre comportamenti assumibili dal lock hanno tutti tipo `LockMsg`, che è supertipo dei messaggi *acquire* e *release*. Questo significa che tutti i comportamenti possono gestire entrambi questi tipi di messaggi, in modo analogo a quanto abbiamo implementato nel nostro programma A_T. Notiamo che il lock nell'esempio Akka Typed viene dichiarato di tipo `Acquire`: `Acquire <: LockMsg`, quindi in accordo con la definizione `ActorRef[-T] ActorRef[LockMsg] <: ActorRef[Acquire]`. La riga 75 è quindi un assegnamento corretto in quanto l'espressione a destra ritorna un `ActorRef[LockMsg]`. Questo consente di restringere il set di messaggi che possono essere inviati al lock a soli messaggi *acquire*, come in effetti ci aspetteremmo da un lock nello stato iniziale di non acquisito. Nel nostro linguaggio A_T definire una variabile analoga a `lock` non è possibile, in quanto non è stato definito l'assegnamento. Nell'esempio del lock usa e getta scritto in A_T quindi è possibile inviare un messaggio *release* ad un lock in uno stato libero; è poi il comportamento a definire la sequenza di istruzioni che segue ad un messaggio di questo tipo, ovvero $\underline{0}$ (il messaggio viene semplicemente scartato).

Osserviamo infine che, nell'esempio scritto in Akka Typed, abbiamo mantenuto il controllo che il lock possa essere rilasciato solo dall'attore che effetti-

vamente l'aveva acquisito, controllo che nel programma in linguaggio A_T non abbiamo inserito per semplicità.

4.3.2 Polling Lock

Consideriamo il polling lock visto al capitolo 2. Su questo esempio non possiamo fare type checking con le regole di tipo date per il linguaggio A_T, in quanto la sintassi necessiterebbe di gestire tipi ricorsivi con l'aggiunta di variabili di tipo che ci riserviamo di aggiungere come lavoro futuro. Tuttavia possiamo dedurre informalmente il tipo associato al lock e al client e verificare che il corrispondente programma in Akka Typed tipa correttamente con i tipi dedotti dal nostro programma A_T.

Ricordiamo brevemente i comportamenti assunti dal lock nel caso di un polling lock, ovvero di un lock che in seguito ad una richiesta mentre si trova in uno stato busy, risponde al richiedente con un messaggio *busy*.

$$\beta_{FREE} = \{acquire(x) \Rightarrow become(\beta_{BUSY}(x)); x \,!\, here(Lock);\}$$
$$release(x) \Rightarrow \underline{0}\}$$

$$\beta_{BUSY}(y) = \{acquire(x) \Rightarrow x \,!\, busy(Lock)$$
$$release(x) \Rightarrow if(x = y)\ become(\beta_{FREE})\}$$

Il client, di conseguenza, ha comportamento

$$\beta_C = \{here(x) \Rightarrow e_C; x \,!\, release(C)$$
$$busy(x) \Rightarrow x \,!\, acquire(C)\}$$

dove e_C è un'espressione qualsiasi che utilizza il lock.

I tipi che ci aspettiamo per i due attori *Lock* e C del programma scritto in A_T sono:

$$T_L = \langle acquire(T_C), release(T_C)\rangle$$
$$T_C = \langle here(\langle release(T_C)\rangle), busy(\langle acquire(T_C)\rangle)\rangle$$

ovvero vorremmo che il lock gestisse messaggi *acquire* e *release* e inoltre che un lock acquisito possa solo essere rilasciato, mentre ad un lock occupato possano essere inviati soli messaggi *acquire*, come indicato dal tipo T_C del client.

Vediamo l'esempio scritto in Akka Typed e verifichiamo che, con questi tipi, il codice compila ed esegue correttamente.

Codice Akka Typed

```
1 import akka.typed._
2 import akka.typed.ScalaDSL._
3
```

```
4  object TypedPollingLock extends App{
5
6  sealed trait LockMsg
7    case class Acquire(client: ActorRef[ClientMsg]) extends LockMsg
8    case class Release(client: ActorRef[ClientMsg]) extends LockMsg
9
10 sealed trait ClientMsg
11   case class Here(lock: ActorRef[Release]) extends ClientMsg
12   case class Busy(lock: ActorRef[Acquire]) extends ClientMsg
13
14 def bBusy(who: ActorRef[ClientMsg]): Behavior[LockMsg] =
15   ContextAware[LockMsg] { ctx =>
16   Total[LockMsg] {
17   case Acquire(client) =>
18     println(client.path.name+" non ottiene il lock ")
19     client ! Busy(ctx.self)
20     Same
21   case Release(client) if client == who =>
22     println(client.path.name+" rilascia il lock ")
23     bFree
24   case Release(client) if client != who =>
25     println(client.path.name+" cerca di rilasciare un lock che non ha ")
26     Same
27   }
28 }
29
30 val bFree : Behavior[LockMsg] = ContextAware[LockMsg] { ctx =>
31   Total[LockMsg] {
32   case Acquire(client) =>
33     println(client.path.name+" acquisisce il lock ")
34     client ! Here(ctx.self)
35     bBusy(client)
36   case Release(client) =>
37     println(client.path.name+" cerca di rilasciare un lock che non ha ")
38     Same
39 }
40 }
41
42 val bClient = ContextAware[ClientMsg] { ctx =>
43   Total[ClientMsg] {
44   case Here(lock) =>
45     println(ctx.self.path.name+" inizia a lavorare ...")
46     Thread.sleep(3000)
47     println("... " +ctx.self.path.name+" finisce di lavorare")
48     lock ! Release(ctx.self)
```

```
49      Same
50    case Busy(lock) =>
51      println(ctx.self.path.name+" ritenta tra poco ")
52      Thread.sleep(6000)
53      println(ctx.self.path.name+" tenta nuovamente di acquisire il lock ")
54      lock ! Acquire(ctx.self)
55      Same
56  }
57  }
58
59  case class Spawn(name: String)
60
61  val bGenerator:Behavior[Spawn] = ContextAware[Spawn] { ctx =>
62    Static[Spawn] {
63      case Spawn(name: String) =>
64        val c = ctx.spawn(Props(bClient), name)
65        println(name+ " tenta di acquisire il lock")
66        lock ! Acquire(c)
67    }
68  }
69
70  val lock : ActorRef[Acquire] = ActorSystem("Lock", Props(bFree))
71  val generator = ActorSystem("Generator", Props(bGenerator))
72
73  generator ! Spawn("Alice")
74  generator ! Spawn("Bob")
75  generator ! Spawn("Eve")
76
77  }
```

Il fulcro del sistema di tipi sta nelle righe 6-12, dove vengono definiti i messaggi di tipo `LockMsg`, ovvero quelli ricevuti dai comportamenti del lock che hanno appunto lo stesso tipo, e i messaggi di tipo `ClientMsg`, che definiscono il tipo dei comportamenti del client. Ricordiamo che in Akka Typed il tipo del comportamento è il tipo dell'attore che ha quel comportamento, quindi gli attori che istanziano il lock e il client hanno rispettivamente tipo `[Acquire[ClientMsg], Release[ClientMsg]]` e `[Here[Release], Busy[Acquire]]`, che corrispondono esattamente ai tipi che avevamo individuato a partire dal programma A_T per i due attori. Con questi tipi il programma Akka Typed tipa correttamente.

4.4 Tipo dei comportamenti e soundness

La scelta di Akka typed di far cambiare comportamento ad un attore a patto che il nuovo comportamento abbia esattamente il tipo di quello precedente sembra restrittiva. Tuttavia questo vincolo è necessario per poter garantire

staticamente che tutti i messaggi in arrivo ad un attore saranno anche gestiti (teorema di soundness). Per darne prova prendiamo il nostro linguaggio A_T e cambiamo la regola di tipo (TYPE BEHAVE), in modo da permettere comportamenti più specifici, ovvero in grado di gestire solo un sottoinsieme dei messaggi a cui può rispondere l'attore.

$$\frac{\Gamma \vdash a : \langle m_1(\tilde{T_1}), \ldots, m_k(\tilde{T_k})\rangle \quad \Gamma, \tilde{x_i} : \tilde{T_i} \vdash_a e_i \;\; \forall i = 1, .., k}{\Gamma \vdash_a \{m_1(\tilde{x_1}) => e_1, \ldots, m_k(\tilde{x_k}) => e_k\}} \text{(Type Behave 2)}$$

Consideriamo l'esempio del lock usa e getta riscritto in questo modo:

$P = val\ L{:}T_L = Actor\{\beta_{FREE}; \underline{0}\};$
$val\ C_1{:}T_C = Actor\{\beta_C; L\ !\ acquire(C_1); \underline{0}\};$
$val\ C_2{:}T_C = Actor\{\beta_C; L\ !\ acquire(C_2); \underline{0}\}; \underline{0}$

$\beta_{FREE} = \{acquire(x) \Rightarrow become(\beta_{BUSY}); x\ !\ here(L); \underline{0}\}$
$\beta_{BUSY} = \{release() \Rightarrow \underline{0}\}$

$\beta_C = \{here(x) \Rightarrow x\ !\ release(); \underline{0}\}$

con $T_L = \langle acquire(T_C), release()\rangle$, $T_C = \langle here(\langle release()\rangle)\rangle$.

Il lock ha ora due comportamenti β_{FREE} e β_{BUSY} che rispondono rispettivamente a soli messaggi *acquire* e a soli messaggi *release*. Mostriamo che il programma P è ben tipato utilizzando le regole di tipo con la variante (TYPE BEHAVE 2). Tuttavia runtime questo programma può evolvere utilizzando la regola (JUNK), i.e. non si garantisce staticamente che tutti i messaggi inviati al lock possano essere ricevuti.

Consideriamo $d = val\ C_1{:}T_C = Actor\{\beta_C; L\ !\ acquire(C_1); \underline{0}\}; val\ C_2{:}T_C = Actor\{\beta_C; L\ !\ acquire(C_2); \underline{0}\}; \underline{0}$

$$\frac{\dfrac{*}{L{:}T_L \vdash_L \beta_{FREE}} \qquad \dfrac{\dfrac{\dfrac{\checkmark}{\emptyset \vdash \diamond \;\; L \notin Dom(\emptyset)}\,(CTX)}{L{:}T_L \vdash \diamond}\,(E\ EXP)}{L{:}T_L \vdash_L \underline{0}} \qquad \dfrac{**}{L{:}T_L \vdash d}}{\emptyset \vdash val\ L{:}T_L = Actor\{\beta_{FREE}; \underline{0}\}; d}\,(T\ TOP\ ACTOR)$$

Proseguiamo con la derivazione di * considerando $e = become(\beta_{BUSY}); x\ !\ here(L); \underline{0}$. Da ora in poi non daremo più la derivazione di $L{:}T \vdash \diamond$ già data, ci limiteremo a chiudere la derivazione quando incontriamo questo giudizio:

$$\dfrac{\dfrac{\dfrac{\checkmark}{L{:}T_L \vdash \diamond}}{L{:}T_L \vdash L{:}T_L}\,(PROJECT) \quad \dfrac{\checkmark}{T_L <: acquire(T_C)}\,(SUB\,W)}{\dfrac{L{:}T_L \vdash L{:}\langle acquire(T_C)\rangle}{L{:}T_L \vdash_L \{acquire(x) \Rightarrow e\}}\,(SUBSUMP) \quad \dfrac{\bullet}{L{:}T_L, x{:}T_C \vdash_L e}}\,(T\,BEHAVE\,2)$$

Proseguiamo con la derivazione di $\bullet$. Definiamo $\Gamma = L{:}T_L, x{:}T_C$.

$$\dfrac{\dfrac{\bullet\bullet}{\Gamma \vdash_L \beta_{BUSY}} \quad \dfrac{\overset{\circ}{\overline{\Gamma \vdash x{:}\langle here(T)\rangle}} \quad \overset{\circ\circ}{\overline{T = \langle release()\rangle}} \quad \overset{\circ\circ\circ}{\overline{\Gamma \vdash L{:}\langle release()\rangle}} \quad \overline{\Gamma \vdash_L \underline{0}}}{\Gamma \vdash_L x\,!\,here(L);\underline{0}}\,(T\,BECOME)}{\Gamma \vdash become(\beta_{BUSY});x\,!\,here(L);\underline{0}}$$

Chiudiamo le derivazioni di $\circ$, $\circ\circ$, $\circ\circ\circ$:

$$\dfrac{\dfrac{\dfrac{\checkmark}{L{:}T_L \vdash \diamond} \quad x \notin Dom(L{:}T_L)}{\Gamma \vdash \diamond}\,(CTX)}{\Gamma \vdash x{:}\langle here(\langle release()\rangle)\rangle}\,(PROJECT)$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{L{:}T_L \vdash \diamond} \quad x \notin Dom(L{:}T_L)}{\Gamma \vdash \diamond}\,(CTX)}{\Gamma \vdash L{:}T_L}\,(PROJECT) \quad \dfrac{\checkmark}{T_L <: \langle release()\rangle}\,(SUB\,W)}{\Gamma \vdash L{:}\langle release()\rangle}\,(SUBSUMPTION)$$

$$\dfrac{\dfrac{\dfrac{\checkmark}{L{:}T_L \vdash \diamond} \quad x \notin Dom(L{:}T_L)}{\Gamma \vdash \diamond}\,(CTX)}{\Gamma \vdash_L \underline{0}}\,(E\,EXP)$$

Proseguiamo con la derivazione di $\bullet\bullet$. Da questo momento in poi non daremo più la derivazione di $\Gamma \vdash \diamond$, già data sopra.

$$\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma \vdash \diamond}}{\Gamma \vdash L{:}T_L}\,(PROJECT) \quad \dfrac{\checkmark}{T_L <: \langle release() \rangle}\,(SUB\ W)}{\Gamma \vdash L{:}\langle release() \rangle}\,(SUBSUMP) \quad \dfrac{\dfrac{\checkmark}{\Gamma \vdash \diamond}}{\Gamma \vdash_L \underline{0}}\,(E\ EXP)}{\Gamma \vdash_L \{release() \Rightarrow \underline{0}\}}\,(T\ BEHAVE\ 2)$$

Infine deriviamo $**$. Consideriamo $d' = val\ C_2{:}T_C = Actor\{\beta_C; L\ !\ acquire(C_2); \underline{0}\}; \underline{0}$ e $\Gamma' = L{:}T_L, C_1{:}T_C$

$$\dfrac{\dfrac{\triangle}{\Gamma' \vdash_{C_1} \beta_C} \quad \dfrac{\triangle\triangle}{\Gamma' \vdash_{C_1} L\ !\ acquire(C_1); \underline{0}} \quad \dfrac{\triangle\,\triangle\,\triangle}{\Gamma' \vdash d'}}{L{:}T_L \vdash val\ C_1{:}T_C = Actor\{\beta_C; L\ !\ acquire(C_1); \underline{0}\}; d'}\,(T\ TOP\ ACTOR)$$

Deriviamo $\triangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{L{:}T_L \vdash \diamond}\ C_1 \notin Dom(L{:}T_L)}{\Gamma' \vdash \diamond}\,(CTX)}{\Gamma' \vdash C_1{:}\langle here(T_x) \rangle \quad T_x = \langle release() \rangle}\,(PROJECT) \quad \dfrac{\star}{\Gamma', x{:}\langle release() \rangle \vdash_{C_1} x\ !\ release(); \underline{0}}}{\Gamma' \vdash_{C_1} \{here(x) \Rightarrow x\ !\ release(); \underline{0}\}}\,(T\ BEHAVE\ 2)$$

Chiudiamo la derivazione di $\star$. Di nuovo, avendo già dato la derivazione di $\Gamma' \vdash \diamond$, ci limitiamo a chiudere la derivazione quando troviamo un giudizio di questo tipo.

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma' \vdash \diamond}\ x \notin Dom(\Gamma')}{\Gamma', x{:}\langle release() \rangle \vdash \diamond}\,(CTX)}{\Gamma', x{:}\langle release() \rangle \vdash x{:}\langle release() \rangle}\,(PROJECT) \quad \dfrac{\dfrac{\dfrac{\checkmark}{\Gamma' \vdash \diamond}\ x \notin Dom(\Gamma')}{\Gamma', x{:}\langle release() \rangle \vdash \diamond}\,(CTX)}{\Gamma', x{:}\langle release() \rangle \vdash_{C_1} \underline{0}}\,(E\ EXP)}{\Gamma', x{:}\langle release() \rangle \vdash_{C_1} x\ !\ release(); \underline{0}}\,(T\ SEND)$$

Deriviamo ora $\triangle\triangle$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\checkmark}{\Gamma' \vdash \diamond}}{\Gamma' \vdash L{:}T_L}\,(PROJECT) \quad \dfrac{\checkmark}{T_L <: \langle acquire(T_C) \rangle}\,(SUB\ W)}{\Gamma' \vdash L{:}\langle acquire(T_C) \rangle} \quad \dfrac{\dfrac{\checkmark}{\Gamma' \vdash \diamond}}{\Gamma' \vdash C_1{:}T_C}\,(PROJECT) \quad \dfrac{\dfrac{\checkmark}{\Gamma' \vdash \diamond}}{\Gamma' \vdash_{C_1} \underline{0}}\,(E\ EXPR)}{\Gamma' \vdash_{C_1} L\ !\ acquire(C_1); \underline{0}}\,(T\ SEND)$$

La derivazione di $\triangle\triangle\triangle$ è esattamente analoga quindi la tralasciamo. Possiamo quindi affermare che, con i tipi T_L e T_C rispettivamente per il lock e per i client, questo programma è ben tipato. Tuttavia vi è almeno un interleaving possibile in cui l'esecuzione richiede l'uso della regola (JUNK).

Prendiamo per esempio il seguente interleaving:

$$[L \mapsto \emptyset](\beta_{FREE})L\{\underline{0}\} \mid [C_1 \mapsto \emptyset](\beta_C)C_1\{L!acquire(C_1)\} \mid [C_2 \mapsto \emptyset](\beta_C)C_2\{L!acquire(C_2)\}.$$

Secondo la regola (PAR) della semantica si può far evolvere uno degli attori in parallelo. L non ha corpo né messaggi in mailbox quindi possiamo far evolvere C_1 oppure C_2. Scegliamo di far evolvere C_1:

$$[L \mapsto acquire(C_1)](\beta_{FREE})L\{\underline{0}\} \mid [C_1 \mapsto \emptyset](\beta_C)C_1\{\underline{0}\} \mid [C_2 \mapsto \emptyset](\beta_C)C_2\{L!acquire(C_2)\}$$

A questo punto supponiamo che ad evolvere sia L, che riceve il messaggio $acquire(C_1)$, passa ad avere comportamento β_{BUSY} e concede il lock a C_1.

$$\begin{aligned}\longrightarrow^{RECEIVE}\ & [L \mapsto \emptyset](\beta_{FREE})L\{become(\beta_{BUSY}); C_1\ !\ here(Lock); \underline{0}\} \mid \\ & [C_1 \mapsto \emptyset](\beta_C)C_1\{\underline{0}\}\ ||; [C_2 \mapsto \emptyset](\beta_C)C_2\{L!acquire(C_2)\}\end{aligned}$$

$$\begin{aligned}\longrightarrow^{BECOME}\ & [L \mapsto \emptyset](\beta_{BUSY})L\{C_1\ !\ here(L); \underline{0}\} \mid [C_1 \mapsto \emptyset](\beta_C)C_1\{\underline{0}\} \mid \\ & [C_2 \mapsto \emptyset](\beta_C)C_2\{L!acquire(C_2)\}\end{aligned}$$

$$\begin{aligned}\longrightarrow^{SEND}\ & [L \mapsto \emptyset](\beta_{BUSY})L\{\underline{0}\} \mid [C_1 \mapsto here(L)](\beta_C)C_1\{\underline{0}\} \mid \\ & [C_2 \mapsto \emptyset](\beta_C)C_2\{L!acquire(C_2)\}\end{aligned}$$

Supponiamo ora che evolva C_2. C_2 manda un messaggio di *acquire* al lock.

$$\begin{aligned}\longrightarrow^{SEND}\ & [L \mapsto acquire(C_2)](\beta_{BUSY})L\{\underline{0}\} \mid [C_1 \mapsto here(L)](\beta_C)C_1\{\underline{0}\} \mid \\ & [C_2 \mapsto \emptyset](\beta_C)C_2\{\underline{0}\}\end{aligned}$$

A questo punto il lock ha in mailbox un messaggio *acquire*, ha inoltre corpo vuoto e comportamento β_{BUSY} che può gestire solo messaggi *release*. Può quindi essere applicata la regola (JUNK).

Abbiamo quindi dimostrato, portando un controesempio, che se considerassimo ben tipati attori che cambiano dinamicamente il set di messaggi che gestiscono, non saremmo più in grado di garantire staticamente che un attore ben tipato riceverà tutti i messaggi che gli saranno inviati nel corso dell'esecuzione del programma.

Conclusioni

Abbiamo presentato un calcolo formale per gli attori Scala Akka. In particolare abbiamo descritto un linguaggio A che modella la creazione degli attori, lo scambio di messaggi e il cambio di comportamento degli attori Akka. Abbiamo presentato diversi esempi di programmi scritti in A e in Akka e abbiamo evidenziato come effettivamente le esecuzioni coincidessero.

Abbiamo poi esteso il linguaggio A aggiungendo la gerarchia di attori e modellando la terminazione e abbiamo chiamato il nuovo linguaggio $A+$. Anche i programmi scritti in $A+$ hanno mostrato un'esecuzione coincidente con quella degli stessi programmi scritti in Scala Akka. $A+$ è la base per modellare in futuro il sistema di supervisione, che si basa proprio sulla gerarchia che questo linguaggio traccia.

Infine abbiamo assegnato dei tipi agli attori del linguaggio A e, prendendo spunto dal modulo sperimentale Akka Typed, abbiamo creato un sistema di tipi per il nuovo linguaggio tipato A_T. Con il sistema di tipi presentato abbiamo dimostrato che possiamo garantire staticamente che in un programma ben tipato un attore a tempo di esecuzione sarà in grado di gestire tutti i messaggi che gli verranno inviati, analogamente a quanto si propone di fare Akka Typed.

Questo lavoro può essere esteso in futuro introducendo nel sistema di tipi i tipi ricorsivi, in modo da poter tipare staticamente esempi di attori con comportamenti ciclici. Con un sistema di tipi ricorsivo possiamo garantire staticamente che, se l'esempio del polling lock presentato è ben tipato, nessun messaggio verrà scartato runtime.

Bibliografia

[1] *http://www.typesafe.com/activator/template/akka-sample-fsm-scala#code/src/main/scala/sample/become/DiningHakkersOnBecome.scala*.

[2] Pypl popularity of programming language. *http://pypl.github.io/PYPL.html*.

[3] Scala akka documentation. *http://doc.akka.io/docs/akka/2.4.1*.

[4] Scala overview. *http://www.scala-lang.org/what-is-scala.html*.

[5] http://www.reactivemanifesto.org/. 2014.

[6] Philip Wadler Atsushi Igarashi, Benjamin C. Pierce. Featherweight java: A minimal core calculus for java and gj. 2002.

[7] Leszek Gruchala. Overview of akka typed. *scalac Official Team Blog*, 2015.

[8] Carl Hewitt. Actor model of computation: Scalable robust information systems. 2010.

[9] Aleksandar Prokopec. *Learning Concurrent Programming in Scala*. PACKT, 2014.

[10] Gul Agha Rajesh K. Karmani. Actors. *Encyclopedia of Parallel Computing*, 2011.

[11] Derek Wyatt. *Akka Concurrency*. 2013.