Alma Mater Studiorum · Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
Corso di Laurea Magistrale in Informatica

Analisi statica dei Deadlock in Featherweight Java con Futuri

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Correlatori:
Elena Giachino
Michael Lienhardt

Presentata da:
Carlo Augusto Grazia

Sessione II
Anno Accademico 2011/2012

Abstract

Abbiamo studiato `ABSFJf`, un linguaggio ad oggetti concorrente con tipi di dato futuro ed operazioni per acquisire e rilasciare il controllo delle risorse. I programmi `ABSFJf` possono manifestare lock (deadlock e livelock) a causa degli errori del programmatore. Per individuare staticamente possibili comportamenti non voluti abbiamo studiato e implementato una tecnica per l'analisi dei lock basata sui contratti, che sono una descrizione astratta del comportamento dei metodi. I contratti si utilizzano per formare un automa i cui stati racchiudono informazioni di dipendenza di tipo chiamante-chiamato; vengono derivati automaticamente da un algoritmo di type inference e modellati da un analizzatore che sfrutta la tecnica del punto fisso con saturazione. Il fine è quello di individuare dipendenze circolari all'interno di uno stato dell'automa per ottenere staticamente informazioni sui diversi lock.

Indice

Capitolo 1

Introduzione

La programmazione concorrente ad oggetti è un modello di programmazione che risale agli anni 80' [35, 3] ed attualmente è utilizzato dai più diffusi linguaggi di programmazione (`Java`, `C#`. `C++`, `Objective C`, etc.). La necessità che ha portato alla definizione di questo modello è stata la combinazione di programmazione distribuita e programmazione orientata agli oggetti (come gli Agha's Actors [2]). In questo contesto, le chiamate di metodo sono state modellate come *scambio di messaggi asincrono*, dove il chiamante continua l'esecuzione *in parallelo al* metodo chiamato. Più recentemente, l'invocazione standard dei metodi è stata integrata con aspetti di "basso accoppiamento" utilizzando *tipi futuri* [22, 27] ed esplicite *operazioni per recuperare il valore di ritorno e rilasciare il controllo*. Infatti numerose estensioni che includono tali caratteristiche sono state progettate, spesso come librerie, per i famosi e già citati linguaggi `C++` [21], `Java` [34], `C#`, Visual Basic e .NET [32], così come nuovi prototipi sono stati proposti [16]. Mentre questi meccanismi di scheduling espliciti (richiedere il valore, rilasciare il controllo ecc.) permettono metodologie flessibili di sincronizzazione, fondamentali per ottimizzazioni o per evitare attese inutili, il debugging dei programmi che utilizzano questi meccanismi risulta molto difficile a causa delle inconsistenze tra i punti di rilascio delle risorse in metodi separati, ma cooperanti. Perseguendo lo scopo di definire un linguaggio "leggero", sufficientemente piccolo per facilitare la dimostrazione di proprietà basilari, è stato realizzato un linguaggio orientato agli oggetti con tipi di dato futuro e con operazioni per rilasciare il controllo. È stata inoltre sviluppata ed implementata una tecnica per l'analisi dei deadlock. Tale linguaggio orientato agli oggetti, chiamato *Featherweight Java con futuri*, `FJf` in breve, è una estensione di Featherweight Java [14]. In `FJf`, gli oggetti hanno diversi task in esecuzione ed al più un task per ogni oggetto è attivo in qualsivoglia momento. I task attivi possono restituire esplicitamente il controllo per consentire ad altri task dello stesso

oggetto di proseguire l'esecuzione. I task sono creati tramite l'invocazione di metodi: l'attività del chiamante prosegue dopo l'invocazione ed il codice del metodo chiamato viene eseguito in un task differente. La sincronizzazione tra il chiamante ed il metodo chiamato viene realizzata quando il risultato è strettamente necessario. Per "disaccoppiare" l'invocazione di metodo con il valore di ritorno, `FJf` usa *variabili future* che sono puntatori ai valori che non sono ancora disponibili. Chiaramente, l'accesso ai valori delle variabili future richiede l'attesa del valore di ritorno.

In un modello ad oggetti con operazioni di scheduling esplicite, compaiono i tipici lock (deadlock o livelock) quando uno o più task sono in attesa di terminazione l'uno dell'altro per restituire un valore. Una semplice dipendenza circolare che coinvolge un solo task è visibile nel metodo:

```
Int fatt(Int n){ return    if (n==0) then 1 ;
                           else n*(this!fatt(n-1).get)   ; }
```

tale metodo definisce la funzione fattoriale (nel caso dell'esempio sono stati inclusi tipi di dato primitivo come `Int` e lo statement condizionale nella sintassi di `FJf`). Nel corpo di `fatt`, la chiamata ricorsiva `this!fatt(n-1)` è seguita da una operazione `get` che richiede il valore restituito dall'invocazione. Siccome l'operazione di `get` non rilascia il lock dall'oggetto chiamante, il task che valuta `this!fact(n-1)` è destinato a non essere mai eseguito perché il suo oggetto è lo stesso del chiamante. Lo scopo principale è quello di implementare una tecnica per l'analisi statica dei deadlock e livelock in un programma `FJf`. La tecnica proposta è basata sui *contratti*, che sono una descrizione astratta dei comportamenti e racchiudono le informazioni necessarie per individuare i lock [19, 20]. Per esempio, il contratto di `fatt` (assumendo che il metodo appartenga alla classe `Maths` senza campi) è a`[ ](){ Maths.fact` a`[ ]().(`a, a`) }`. Questo contratto dichiara che l'invocazione di `fatt` sull'oggetto a chiamerà ricorsivamente `fatt` sullo stesso oggetto a e l'invocazione introduce una *dipendenza tra nomi di oggetto* (a, a). La dipendenza specifica che l'oggetto chiamante, *contenuto nella coppia in prima posizione*, viene rilasciato non appena la chiamata ottiene e rilascia il suo oggetto, che è *contenuto nella coppia in seconda posizione*. È quindi stato definito ed implementato un sistema di inferenza per associare un contratto a ciascun metodo del programma e ciascuna espressione da valutare.

Per individuare staticamente potenziali lock in programmi `FJf`, è stato introdotto ed implementato un modello che calcola un automa a stati finiti partendo dai contratti, che raccoglie le informazioni sulle dipendenze chiamante-chiamato all'interno degli stati. Tale automa prende il nome di *lamp* (da *Lock Analysis Model with Pairs*). In particolare quindi, ogni stato

dell'automa è una relazione sui nomi di oggetto e potenziali comportamenti non voluti come deadlock o livelock vengono segnalati dalla presenza di circolarità di dipendenze. Viene quindi anche presentata la semantica dei *lamp* per i contratti che, dato un sistema di tipi per metodi ed espressioni, permette di associare un *lamp* ad ogni programma `FJf`.

Per esempio, il modello del metodo `fatt` definito sopra è un *lamp* formato dal singolo stato $\{(a, a)\}$. La coppia (a, a) in uno stato segnala *dipendenza circolare tra nomi di oggetto*, che permette di concludere che il metodo `fatt` manifesta un lock, in tal caso un deadlock. Infatti tale metodo per il calcolo del fattoriale è una implementazione sbagliata in `FJf`, una corretta implementazione si vedrà nelle conclusioni.

Il punto chiave del contributo della tesi è l'implementazione del sistema di inferenza dei contratti e dell'analizzatore statico dei lock, oltre che l'analisi e il riadattamento della teoria di `FJf` ad un caso pratico come `ABSFJf` (un linguaggio già esistente con sintassi differente ed eguale semantica).

La tesi è organizzata come segue. A seguire in questo capitolo viene mostrato lo stato dell'arte dell'argomento di tesi. Il capitolo 2. introduce il linguaggio `ABSFJf`, la sintassi, la semantica ed alcuni esempi. Il capitolo 3. definisce il sistema di tipi dei contratti e l'algoritmo di inferenza per calcolarli automaticamente. Il capitolo 4. definisce l'analizzatore dei deadlock, come viene svolta tale analisi, come è stata implementata ed alcuni esempi per vedere l'intero tool all'opera. Il capitolo 5. presenta le conclusioni e i lavori futuri.

1.1 Stato dell'arte

Analisi statica dei deadlock. L'analisi dei deaclock basata sui tipi è stata fortemente studiata. Alcune proposte riguardano calcoli di processi [19, 31, 30, 33], ma alcuni contributi riguardano anche i deadlock nei programmi orientati agli oggetti [4, 8, 1].

Kobayashi, in lavori sul pi-calcolo [18, 13, 19], definisce dipendenze tra i canali utilizzando "capacità" ed "obblighi" dei tipi e verifica l'assenza di circolarità. Così come la tecnica implementata nella tesi, è in grado di trattare comportamenti ricorsivi e creazioni di nuovi canali.

In ogni caso la sua tecnica è diversa da quella presentata ed un completo confronto richiederebbe l'applicazione della nostra tecnica al pi-calcolo, che ha un differente modello di sincronizzazione rispetto a quello che verrà descritto in seguito.

Il lavoro di Suenaga [31, 30] applica la tecnica di Kobayashi ai linguaggi concorrenti con risorse. Risulta facile estrarre il *lamp* da un programma in un linguaggio con interrupt di [31] e, pertanto, essere in grado di verificare la presenza di deadlock altrettanto facilmente. In ogni caso, un preciso confronto con [31] non è stato fatto. Il linguaggio di [30] utilizza referenze mutabili, che sono un noto aspetto che, insieme alla concorrenza, raggiunge un comportamento non deterministico.

In altri contributi [4, 17, 10], un sistema di tipi calcola un ordinamento parziale dei lock in un programma e con un teorema si dimostra che i task seguono quell'ordine. Al contrario, la nostra tecnica non calcola alcun ordinamento sui lock, garantendo una maggiore flessibilità: un programma può acquisire due lock in ordine differente in diverse fasi, che è corretto nel nostro caso, ma non corretto con l'altra tecnica.

La maggiore flessibilità si può chiarire con un esempio applicativo che coinvolge la classe `C` e i metodi `m` ed `n` della Tabella 1.1.

```
class C {
     C m(C x) { return x ;}
     C n(C y, C z) { return y!m(z).get; z!m(y).get ;}
```

Tabella 1.1: Classe per accesso ordinato alle risorse in FJf

Nella Tabella 1.1 il metodo `n` accede alle due risorse y e z, che ottiene come parametri, in uno specifico ordine: prima y e poi z. In `FJf` e quindi in `ABSFJf` è possibile scrivere un `main` che valuti l'espressione

$$\mathtt{a!n(b,c)};\ \mathtt{a!n(c,b)}$$

costringendo il programma ad acquisire i lock sulle risorse b e c, prima in un ordine e poi nell'opposto (seguendo appunto il pattern di acquisizione definito nel metodo `n`). Questo programma non produce alcun deadlock e non può essere trattato con le tecniche basate su ordinamento di nomi, in quando non è possibile definire un ordinamento parziale in cui le risorse b e c compaiano nei due ordini opposti.

Una ulteriore differenza con i lavori sopra citati è l'utilizzo dei contratti. I contratti sono termini in un processo algebrico [20]. L'utilizzo di un semplice processo algebrico per descrivere protocolli (comunicazione o sincronizzazione) non è nuovo. Questo è il caso di soluzioni di scambio in SSDL [29], che si basano su CSP [5] e il pi-calcolo [25], o sui tipi comportamentali in [28] ed in [7], che usano CCS [24].

Altri approcci statici, che non si basano sui tipi, sono [23, 6] dove le dipendenze circolari all'interno dei processi sono rilevate come configurazioni erronee, ma la creazione dinamica di nomi non è trattata. Vediamo con un esempio come la nostra tecnica sia più potente e permetta di analizzare correttamente programmi che creano un insieme infinito di nomi.

```
class C {
     C m() { return new C() ;}
     C n() { return while(true){ (new C())!m().get } ;}
```

Tabella 1.2: Classe per creazione di risorse infinite in FJf

Nella Tabella 1.2 si producono lock su infinite risorse, la creazione infinita di nomi di oggetto porta ad infiniti accessi a risorse nuove (nell'esempio è stato incluso lo statement `while` per facilitare la lettura del codice, ma non è necessario, infatti, per realizzare un esempio pratico, si potrebbe usare un metodo ricorsivo). In `FJf` e quindi in `ABSFJf` è possibile scrivere un `main` che valuti l'espressione:

`a!n()`;

costringendo l'oggetto a ad acquisire i lock sulle risorse b_1, b_2 , ... (Figura 1.1). Questo programma, a seguito della saturazione, non produce, correttamente, alcun deadlock. Lo stesso programma non può essere trattato con le tecniche [23, 6] in quando non è possibile definire un ordinamento parziale su un numero illimitato di risorse.

Linguaggio e Contratti. Il linguaggio che è stato studiato nella tesi è inspirato al linguaggio Creol [16], che ha ulteriori primitive di scheduling ed operazione per l'update dei campi. Contratti simili a quelli utilizzati nella tesi sono stati studiati in [9] per un linguaggio della famiglia di Creol [15] con

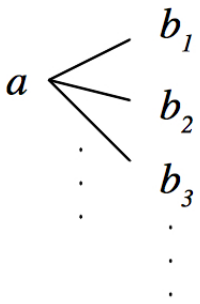

Figura 1.1: Accesso ad un numero illimitato di risorse

lo stesso obiettivo di analisi dei deadlock. A parte questa fonte di ispirazione, `FJf` è maggiormente una estensione di Featherweight Java [14] con tipi futuri e l'operazione di `get` è descritta in [34].

Modello. Il modello presentato in questa tesi è basato sul modello descritto in [9], in cui si adopera la tecnica standard del punto fisso, tramite l'utilizzo di approssimanti finiti per catturare la generazione infinita di nuovi nomi.

Capitolo 2

Il linguaggio

2.1 Presentazione

ABSFJf è un linguaggio concorrente ad oggetti con tipi futuri ed operazioni per ottenere valori e rilasciare il controllo. Tale linguaggio deriva dal suo predecessore **FJf** [9] una estensione di Featherweight Java [14].

Il motivo per cui l'argomento di tesi è `ABSFJf` e non direttamente `FJf` è pura questione di praticità; `ABSFJf` infatti è un nucleo di un linguaggio già esistente il cui nome è **ABS**(Abstract Behavioral Specification) creato per realizzare ed eseguire modelli di sistemi object-oriented distribuiti [15], tale linguaggio fa parte di un progetto europeo di cui fa parte anche `FJf`. Essendo all'interno dello stesso progetto europeo i linguaggi `ABS` ed `FJf` condividono la semantica operazionale delle espressioni. Se da una parte `ABS` è un linguaggio di programmazione "completo" (provvisto quindi di compilatore, type-checker, esecutore ecc...), d'altra parte `FJf` viene presentato solo a livello teorico e non dispone quindi di un tool già esistente in grado di compilarlo ed eseguirlo. Per non cominciare da zero è stato scelto quindi di lavorare a partire dal tool già esistente di `ABS`, tool già perfettamente funzionante, introducendo le componenti di interesse necessarie per gli studi dei deadlock su `FJf`.

Giunti a questo punto è lecito chiedersi:

"Quindi cos'è `ABSFJf`?"

La risposta è semplice: `ABSFJf` è il nucleo sintattico di `ABS` che garantisce le funzionalità di `FJf` (Fig 2.1). Schematizzando i vantaggi dell'utilizzo di `ABSFJf` ricordiamo:

- Semantica operazionale equivalente ad `FJf`;

- Compilatore, Type-Checker ed Esecutore già presenti ("ereditati" da ABS);
- Permette di concentrarsi solo sull'implementazione delle componenti necessarie all'analisi dei deadlock.

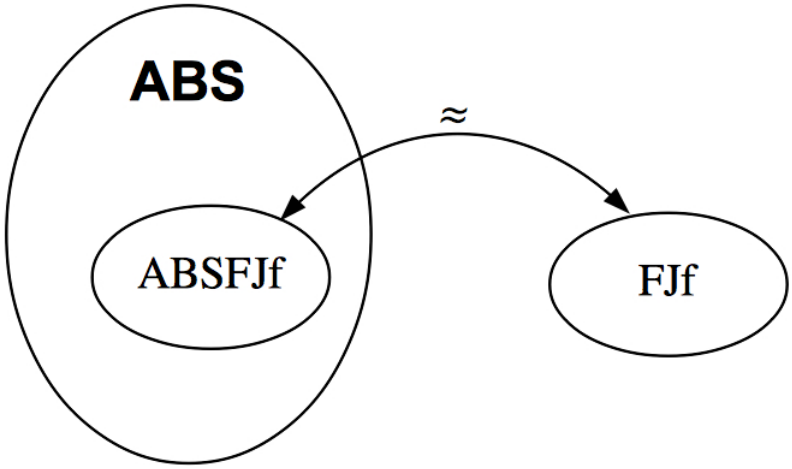

Figura 2.1: Snapshot delle relazioni tra ABS, FJf ed ABSFJf

Il simbolo ≈ in figura invita a precisare che, se da una parte abbiamo perfetta equivalenza tra le semantiche operazionali di `ABSFJf` ed `FJf`, dall'altra non abbiamo equivalenza sintattica (è possibile però definire una funzione che automatizzi il passaggio tra le due sintassi differenti).

2.2 ABSFJf in breve

In `ABSFJf` un programma è un insieme di definizioni di classe seguite da un **Main** da valutare. Ogni definizione di classe deve essere sempre preceduta dalla dichiarazione dell'interfaccia che implementa. La classe senza campi e metodi **Object** è sempre presente in ogni programma per rappresentare l'oggetto elementare più semplice.

```
interface Object {}
class Object extends Object {}

interface C {
     C m(){}
     C n1(C c){}
     C n2(C c){}   }

class C(Object f) extends C {
     C m() { return new cog C(f) ;}
     C n1(C c) { return (c!m()).await.get ;}
     C n2(C c) { return (c!m()).get ;}  }
```

Tabella 2.1: Semplice classe ABSFJf

Nella Tabella 2.1[1] abbiamo una semplice definizione di classe `C`, tale classe è definita fornendo un parametro (equivalente ad un campo di classe in Java) `f` e tre metodi `m`, `n1` ed `n2`. Quando viene invocato il metodo `m`, un nuovo oggetto della classe `C` viene creato (con il parametro `f` contenente lo stesso valore dell'oggetto chiamante) ed immediatamente restituito.

L'invocazione di metodi in `ABSFJf` è *asincrona*; per questo motivo utilizziamo il simbolo del punto esclamativo anziché l'usuale punto dell'invocazione di metodo (Vedere il corpo dei metodi `n1` ed `n2` nella Tabella 2.1).

Una caratteristica di `ABSFJf` è quella di permettere, all'interno della dichiarazione di un metodo, di decidere dove si vuole che il processore rilasci la risorsa a runtime, definendo quindi un punto di rilascio attraverso l'operazione di **await**. Facendo riferimento alla Tabella 2.1, l'invocazione

```
x!n1(x)
```

[1]La sintassi delle espressioni di ritorno nella Tabella 2.1 è quella di `FJf`, l'equivalente `ABSFJf` avrebbe avuto l'espressione spezzata in più statement, è stata fatta tale scelta di sintassi mista per aumentare la leggibilità

dove `x` è un oggetto della classe `C`, porta all'esecuzione del metodo `m` sull'oggetto `x`. Siccome il chiamante (mettodo `n1`) ed il metodo chiamato `m` condividono il medesimo oggetto `x`, il codice del metodo `m` non può essere valutato finché il chiamante `n1` non rilascia esplicitamente il controllo. È proprio l'operazione di `await` che permette di rilasciare il controllo in `(c!m().await.get)` mentre l'operazione di `get` permette di ricevere il risultato dell'invocazione una volta che questa è terminata.

Le operazioni di `await` e `get` consentono una flessibile procedura di sincronizzazione. Ovviamente al crescere della flessibilità spesso si ha una riduzione della sicurezza e questo contesto non è da meno. Ad esempio, se invece di invocare il metodo `n1` si invocasse il metodo `n2` con l'espressione `x!n2(x)` si avrebbe un blocco dell'esecuzione. Ciò accade perché il metodo `n2` non rilascia il controllo sull'oggetto `x` impedendo di proseguire con l'invocazione del metodo `m`.

Per catturare situazioni pericolose in cui non si ha una corretta sincronizzazione come quella vista sopra, in `ABSFJf` si utilizzano dei tipi comportamentali chiamati *contratti*. Per esempio, il contratto del metodo `m` della Tabella 2.1 è derivato usando la regola (che è una istanza di T-METHOD)

$$\frac{\Gamma + \texttt{this} : (\texttt{C}, a[\texttt{f} : X]) \vdash_a \texttt{new cog C(f)} \ : \ (\texttt{C}, b[\texttt{f} : X])\,,\ 0\{X = a[\texttt{f} : Y]\}}{\Gamma \vdash \ \texttt{C m}\ ()\{\texttt{return new cog C(f);}\} : \texttt{a[f : X]}()\{\texttt{0}\}\ \texttt{b[f : X]}\ \text{IN C}}$$

Il contratto $a[\texttt{f} : X]()\{0\}\ b[\texttt{f} : X]$ specifica che l'oggetto su cui viene invocato `m` è chiamato $a[\texttt{f} : X]$ dove a è il nome dell'oggetto mentre X è il valore del parametro `f`; l'oggetto restituito è $b[\texttt{f} : X]$ che ha un diverso nome di oggetto b ma il medesimo valore X come parametro. Il contratto inoltre specifica il comportamento del metodo che in questo caso è vuoto ($= 0$). Le premesse della regola vista presentano una valutazione del tipo $\Gamma \vdash_a \texttt{e} \ : \ (\texttt{T}, \mathbb{r})\,,\ \mathbb{c} \rhd \mathcal{U}$, dove Γ è l'ambiente, a è il nome dell'oggetto del metodo contenente l'espressione `e`, `e` è una espressione del linguaggio `ABSFJf` il cui tipo è `T`, $\mathbb{r}$ è il *record futuro* il cui significato verrà spiegato a breve, $\mathbb{c}$ è il contratto che racchiude le dipendenze tra oggetto chiamante ed oggetto chiamato mentre $\mathcal{U}$ è un insieme di vincoli (detto anche "Constraint Store").

Oltre ai *record futuri* (dobbiamo ancora spiegare cosa sono) in `ABSFJf` abbiamo anche i *tipi futuri*, in particolare, quando un metodo dichiara come tipo di ritorno il tipo `C`, la corrispondente invocazione del metodo restituisce un *valore futuro* di tipo `Fut(C)` perché il contesto dell'invocazione non può assumere la presenza immediata del valore di ritorno. L'espressione che consente di recuperare il valore di ritorno è l'espressione `get` che prende un'espressione di tipo `Fut(C)` e ne restituisce una di tipo `C`. L'operazione che

rilascia il processo, chiamata `await`, prende un'espressione di tipo `Fut(C)` e restituisce il medesimo tipo `Fut(C)`.

A questo punto risulta giusto chiedersi cosa sia $\mathbb{r}$, il *record futuro*; tale record raccoglie i nomi di oggetto per accedere ai valori futuri. Questi valori, risultato dell'invocazione di metodo, saranno disponibili a patto che venga acquisito il controllo anche dall'oggetto che invoca il metodo, a patto quindi che che il metodo invocato venga eseguito. Facciamo alcuni esempi per chiarire, l'espressione `new cog C(f)`, di tipo standard `C`, ha $b[\texttt{f} : X]$ come record futuro. L'espressione `c!m()` invece ha come tipo il tipo futuro `Fut(C)` e come record futuro $b \rightsquigarrow b'[\texttt{f} : Y]$. Ciò significa che, se risulta necessario ottenere il valore di `c!m()` (ovvero il record futuro $b'[\texttt{f} : X]$), come ad esempio all'interno del metodo `n1`, è necessario prima prendere il controllo dell'oggetto di nome b, altrimenti il corpo del metodo `m` non verrebbe eseguito. Raccogliendo i nomi di oggetto, i record futuri giocano un ruolo fondamentale nell'utilizzo e la creazione dei cosiddetti *vincoli*.

2.3 Sintassi

La sintassi di ABSFJf utilizza quattro insiemi disgiunti di nomi: *Nomi di classe* (o interfaccia), come A, B, C, $\cdots$, *nomi di parametro di classe* (equivalenti ai campi, per semplicità di seguito verranno sempre chiamati campi), come f, g, $\cdots$, *nomi di metodo*, come m, n, $\cdots$, e *variabili*, come x, y, $\cdots$. il nome speciale this si assume appartenere all'insieme delle variabili. La notazione $\bar{\mathtt{T}}$ è abbreviazione di $\mathtt{T}_1 ; \cdots ; \mathtt{T}_n$ e così similmente per gli altri nomi. Sequenze di coppie sono abbreviate come $\mathtt{T}_1\ \mathtt{f}_1 ; \cdots ; \mathtt{T}_n\ \mathtt{f}_n$ con $\bar{\mathtt{T}}\ \bar{\mathtt{f}}$. La concatenazione di sequenze è seguita da un punto e virgola; La sequenza vuota verrà scritta come $\bullet$ e verrà omessa quando sarà chiaro dal contesto la presenza di una sequenza vuota. La sintassi astratta di *dichiarazione di interfaccia* IL, *dichiarazione di classe* CL, *dichiarazione di metodo* $\mathtt{M}_s$, *implementazione di metodo* M, *espressioni* e e *tipi* T in ABSFJf è la seguente

$$
\begin{array}{lcl}
\mathtt{IL} & ::= & \mathtt{interface\ C}\ \{\overline{\mathtt{M_s}}\} \\
\mathtt{CL} & ::= & \mathtt{class\ C}(\bar{\mathtt{T}}\ \bar{\mathtt{f}})\ \mathtt{implements\ C}\ \{\bar{\mathtt{M}}\} \\
\mathtt{M_s} & ::= & \mathtt{T\ m}\ (\bar{\mathtt{T}}\ \bar{\mathtt{x}}) \\
\mathtt{M} & ::= & \mathtt{M_s}\{\ \mathtt{s};\ \mathtt{return\ e}\ ;\} \\
\mathtt{e} & ::= & \mathtt{v} \mid \mathtt{e}_e \\
\mathtt{v} & ::= & \mathtt{x} \mid \mathtt{this} \mid \mathtt{f} \\
\mathtt{e}_e & ::= & \mathtt{new\ cog\ C}(\bar{\mathtt{v}}) \mid \mathtt{v!m}(\bar{\mathtt{v}}) \mid \mathtt{v.get} \\
\mathtt{s} & ::= & \mathtt{C\ x} \mid \mathtt{x} := \mathtt{e} \mid \mathtt{await\ v?} \mid \mathtt{e} \mid \mathtt{s;s} \\
\mathtt{T} & ::= & \mathtt{C} \mid \mathtt{Fut(T)}
\end{array}
$$

Sequenze di dichiarazioni di campo $\bar{\mathtt{T}}\ \bar{\mathtt{f}}$, dichiarazioni di metodo $\bar{\mathtt{M}}$ e dichiarazione di parametri $\bar{\mathtt{T}}\ \bar{\mathtt{x}}$ non contengono ripetizioni di nomi per assunzione. Ulteriori assunzioni sono doverose per chiarire meglio la ricca sintassi di ABSFJf rispetto a quella di FJf:

- Le interfacce e le classi che le implementano devono condividere lo stesso nome, in questo modo l'oggetto creato con il costruttore di classe `new cog C()` avrà il tipo C dell'interfaccia che la classe implementa. Se ciò non fosse (come nel caso generico di ABS) l'oggetto creato avrebbe come tipo un insieme di interfacce.

- Una classe deve estendere una ed una sola interfaccia, vincolo che unito al precedente ci permette di pensare alla classe C come al tipo C, cosa che in generale non sarebbe vera;

- Nella dichiarazione di classe CL la voce $\mathtt{C}(\bar{\mathtt{T}}\ \bar{\mathtt{f}})$ corrisponde al costruttore di classe;

- I campi della classe $\bar{\texttt{f}}$ non vengono dichiarati all'interno del *body* di classe[2] ma passati come parametri attraverso il *costruttore*, possono poi essere utilizzati nel body della classe come normali variabili.

- Le espressioni `e` si dividono tra variabili ed *espressioni effettive* $\texttt{e}_e$, quest'ultime hanno la particolarità di creare un nuovo oggetto e restituire qualcosa, per cui non possono essere "accumulate" in un'unica espressione sintattica come accade in `FJf`.

- Gli *statement* `s`, tralasciando l'espressione `await` che ha una importanza rilevante, servono soltanto per scomporre espressioni complesse in più parti, dichiarando ed assegnando (una volta sola) variabili per memorizzare il risultato delle sottoespressioni, per cui la *dichiarazione* e *l'assegnazione* di variabili hanno solo una valenza "composizionale".

- Anche `return e` in `ABSFJf` è uno *statement*, quindi, all'interno del body di metodo della grammatica potremmo scrivere solo {`s`} e non {`s`; `return e`} con il vincolo, ovviamente, che nel body di metodo l'ultimo statement debba essere di tipo return.

- Ogni programma `ABSFJf` dichiara all'inizio un'interfaccia vuota dal nome `Object` ed una classe che la implementa di nome `Object` (in accordo con i primi due vincoli dell'elenco).

- Non essendoci ereditarietà in `ABS` non vi è neanche in `ABSFJf` e quindi neanche il sottotipaggio.

Un programma è una coppia $(\text{CT}, \texttt{s})$, dove la *class table* CT è una mappa finita che va dai nomi di classe alle dichiarazioni di classe `CL` mentre `s` è uno statement (il main). All'interno dei tipi, il tipo `Fut(T)` viene chiamato *futuro* del tipo `T`. Definiamo ora tre funzioni ausiliarie cosiddette di *lookup* che saranno utili in seguito, *fields*(`C`), *mtype*(`m`, `C`), ed *mbody*(`m`, `C`) definite nella Tabella 2.2. Scriviamo $\texttt{m} \in \texttt{C}$ quando *mtype*(`m`, `C`) è definita (`m` è un metodo implementato in `C`). Tutte le classi `C` definite in un programma `ABSFJf` appartengono al dominio della class table.

[2] I campi solitamente, come nei linguaggi tipo `Java` vengono dichiarati nel body della classe così come i metodi

Field lookup:

$$fields(\texttt{Object}) = \bullet \qquad \frac{\text{CT}(\texttt{C}) = \texttt{class C}(\bar{\texttt{T}}\ \bar{\texttt{f}})\ \texttt{implements C}\ \{\bar{\texttt{M}}\}}{fields(\texttt{C}) = \bar{\texttt{T}}\ \bar{\texttt{f}}}$$

Method type lookup:

$$\frac{\begin{array}{c}\text{CT}(\texttt{C}) = \texttt{class C implements C}\ \{\bar{\texttt{M}}\} \\ \texttt{T}'\ \texttt{m}\ (\bar{\texttt{T}}'\ \bar{\texttt{x}})\{\texttt{s return e;}\}\ \in \bar{\texttt{M}}\end{array}}{mtype(\texttt{m}, \texttt{C}) = \bar{\texttt{T}}' \to \texttt{T}'}$$

Method body lookup:

$$\frac{\begin{array}{c}\text{CT}(\texttt{C}) = \texttt{class C implements C}\ \{\bar{\texttt{M}}\} \\ \texttt{T}'\ \texttt{m}\ (\bar{\texttt{T}}'\ \bar{\texttt{x}})\{\texttt{s return e;}\}\ \in \bar{\texttt{M}}\end{array}}{mbody(\texttt{m}, \texttt{C}) = \bar{\texttt{x}}.\texttt{s}.\texttt{e}}$$

Tabella 2.2: Funzioni ausiliarie di lookup

2.4 Semantica

La semantica operazionale di `ABSFJf` utilizza due ulteriori insiemi di nomi, i *nomi di oggetto*, come a, b, $\cdots$ e *nomi di task*, come $\texttt{t}$, $\texttt{t}'$, $\cdots$. I nomi di oggetto sono "partizionati" in accordo alla loro classe di appartenenza. Si assume che ci sia un insieme infinito di nomi per ogni classe e la funzione $fresh(\texttt{C})$ restituisce un nuovo nome di oggetto per la classe `C`, viceversa, dato un nome di oggetto a la funzione $class(a)$ ne restituisce la classe di appartenenza.

I *valori* $\texttt{v}$, $\texttt{v}'$, $\cdots$, sono termini definiti dalla seguente grammatica:

$$\texttt{v} ::= \quad \texttt{t} \mid a[\bar{\texttt{f}} : \bar{\texttt{v}}]$$

Per esempio $a[\,]$ è il valore di una classe senza campi (proprio come `Object`). Valori come $a[\bar{\texttt{f}} : \bar{\texttt{v}}]$ sono *record completi*, dove a è il nome dell'oggetto mentre $\bar{\texttt{v}}$ sono i valori memorizzati nei campi (parametri) $\bar{\texttt{f}}$. La semantica operazionale utilizza i nomi di oggetto per implementare mutua esclusione tra task dello stesso oggetto. L'analisi dei deadlock si baserà sulla ricerca di circolarità tra nomi di questo tipo. Di seguito, con un abuso di notazione, i valori, così come le espressioni, saranno rappresentati da $\texttt{e}, \texttt{e}', \cdots$.

Definiamo gli *stati* S, S', $\cdots$, come insieme di *task* $\mathtt{t} :^{\ell}_{a} \mathtt{e}$, dove $\mathtt{t}$ è un nome di task, a è un nome di oggetto, ℓ può essere o $\top$ (se il task detiene il controllo su a) oppure $\bot$ (altrimenti), mentre $\mathtt{e}$ è una espressione.

La semantica operazionale di `ABSFJf` è la relazione di transizione $\xrightarrow{a}$ tra stati definiti in Tabella 2.3 dove vengono utilizzate le seguenti notazioni[3]:

- *Contesto di valutazione* E la cui sintassi è:

$$\begin{array}{rcl} \mathsf{E} & ::= & \mathsf{E}!\mathtt{m}(\bar{\mathtt{e}}) \mid \mathsf{E}.\mathtt{f} \mid a[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \mid \mathtt{new}\ \mathtt{C}(\bar{\mathtt{v}}, \mathsf{E}, \bar{\mathtt{e}}) \mid \mathsf{E}.\mathtt{get} \mid \ldots \\ & & \ldots \mid \mathsf{E}.\mathtt{await} \mid \mathsf{E}; \mathtt{e} \end{array}$$

- Il predicato $\mathit{unlocked}(\mathsf{S}, a)$ restituisce *true* se ogni $\mathtt{t} :^{\ell}_{a} \mathtt{e}$ in S è del tipo $\ell = \bot$;

- La funzione $\mathit{freshtask}(\ )$ restituisce sempre un nuovo nome di task;

- In $\mathtt{t} :^{\ell}_{a} \mathtt{e}$, l'apice ℓ viene omesso quando non rilevante.

Le regole che definiscono l'accesso ad un campo, la creazione di un oggetto e la sequenza, chiamate (Field), (New), (Seq), sono pressoché standard; Discutiamo quindi le altre. La regola (Invk) definisce l'invocazione di un metodo. In accordo a questa regola, la valutazione di $b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')$ produce una referenza futura $\mathtt{t}'$ al valore restituito da $\mathtt{m}$. Il task che risolve la chiamata al metodo viene creato e la valutazione del task chiamante può continuare. L'invocazione è asincrona; in ogni caso, la valutazione del metodo chiamato $\mathtt{m}$ non può cominciare fino a quando il valore di ℓ non diventa $\top$. La regola (Get) permette di recuperare il valore di un metodo chiamato. Le regole (AwaitT) e (AwaitF) modellano l'operazione di `await`: se il task $\mathtt{t}'$ è terminato, ed è quindi accoppiato ad un valore, allora `await` è non bloccante; altrimenti il controllo dell'oggetto viene rilasciato da $\mathtt{t}$. La regola (Release) modella la terminazione di un task, che corrisponde al salvare il valore di ritorno nello stato e rilasciare il controllo dell'oggetto. In accordo alle relazioni di transizione, un task $\mathtt{t} :^{\ell}_{a} \mathtt{e}$ procede $\ell = \top$, ad eccezione della regola (Lock). Questa regola permette ad un task che non ha ancora a disposizione un valore di acquisire il controllo. La regola va letta in accompagnamento alla regola (State) che rafforza la proprietà/condizione che al massimo un task per oggetto può detenere il controllo. Ciò significa che la regola (Lock) non può essere utilizzata se lo stato ha un lock $\mathtt{t}' :^{\top}_{a} \mathtt{e}'$.

[3]Sia la Tabella 2.3 sia le notazioni semantiche utilizzano una sintassi FJf-like per aumentarne la leggibilità, il lettore noti come risulti più semplice scrivere `e.await.get` anziché `x := e; await x?; x.get`

$$\text{(Field)}\quad \frac{\mathtt{f} : \mathtt{v} \in \bar{\mathtt{f}} : \bar{\mathtt{v}} \quad \mathtt{this} = b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]}{\mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{f}] \xrightarrow{a} \mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{v}]}$$

$$\text{(Invk)}\quad \frac{\mathit{mbody}(\mathtt{m}, \mathit{class}(b)) = \bar{\mathtt{x}}.\mathtt{e} \quad \mathtt{t}' = \mathit{freshtask}(\ )}{\mathtt{t} :^{\top}_{a} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]!\mathtt{m}(\bar{\mathtt{v}}')] \xrightarrow{a} \mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{t}'],\ \mathtt{t}' :^{\bot}_{b} \mathtt{e}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]/_{\mathtt{this}}][\bar{\mathtt{v}}'/_{\bar{\mathtt{x}}}]}$$

$$\text{(New)}\quad \frac{\mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}\ \bar{\mathtt{f}} \quad b = \mathit{fresh}(\mathtt{C})}{\mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{new\ cog\ C}(\bar{\mathtt{v}})] \xrightarrow{a} \mathtt{t} :^{\top}_{a} \mathsf{E}[b[\bar{\mathtt{f}} : \bar{\mathtt{v}}]]}$$

$$\text{(Seq)}\quad \mathtt{t} :^{\top}_{a} \mathtt{v}; \mathtt{e} \xrightarrow{a} \mathtt{t} :^{\top}_{a} \mathtt{e}$$

$$\text{(Get)}\quad \mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{t}'.\mathtt{get}],\ \mathtt{t}' :_{b} \mathtt{v} \xrightarrow{a} \mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{v}],\ \mathtt{t}' :_{b} \mathtt{v}$$

$$\text{(AwaitT)}\quad \mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{t}'.\mathtt{await}],\ \mathtt{t}' :_{b} \mathtt{v} \xrightarrow{a} \mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{t}'],\ \mathtt{t}' :_{b} \mathtt{v}$$

$$\text{(AwaitF)}\quad \frac{\mathtt{e} \neq \mathtt{v}}{\mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{t}'.\mathtt{await}],\ \mathtt{t}' :_{b} \mathtt{e} \xrightarrow{a} \mathtt{t} :^{\bot}_{a} \mathsf{E}[\mathtt{t}'.\mathtt{await}],\ \mathtt{t}' :_{b} \mathtt{e}}$$

$$\text{(Release)}\quad \mathtt{t} :^{\top}_{a} \mathtt{v} \xrightarrow{a} \mathtt{t} :^{\bot}_{a} \mathtt{v}$$

$$\text{(Lock)}\quad \frac{\mathtt{e} \neq \mathtt{v}}{\mathtt{t} :^{\bot}_{a} \mathtt{e} \xrightarrow{a} \mathtt{t} :^{\top}_{a} \mathtt{e}}$$

$$\text{(State)}\quad \frac{\mathsf{S} \xrightarrow{a} \mathsf{S}' \quad \mathit{unlocked}(\mathsf{S}'', a)}{\mathsf{S}, \mathsf{S}'' \xrightarrow{a} \mathsf{S}', \mathsf{S}''}$$

Tabella 2.3: Relazioni di transizione di ABSFJf.

2.5 Esempi

Vediamo alcuni esempi per vedere come funzionano le regole di riduzione della semantica operazionale, negli esempi utilizzeremo i metodi della Tabella 2.4 ottenuta a partire dalla Tabella 2.1.

```
interface Object {}
class Object extends Object {}

interface C {
     C m(){}
     C n1(C c){}
     C n2(C c){}
     C p(){}
     Fut(C) q(C b, C c){}   }

class C(Object f) extends C {
     C m() { return new cog C(f) ;}
     C n1(C c) { return (c!m()).await.get ;}
     C n2(C c) { return (c!m()).get ;}
     C p() { return ((new cog C(f))!p()).await.get ;}
     Fut(C) q(C b,C c) { return b!n2(c);c!n2(b) ;}  }
```

Tabella 2.4: Semplice classe ABSFJf

2.5.1 Esempio di corretta terminazione

Come primo esempio vediamo la valutazione dell'espressione `(new cog C(this))!n1(new C(this))`, dove la classe `C` è definita nella Tabella 2.4 (`this` nello stato iniziale è un valore della classe `Object`).

$\texttt{t} :^{\top}_{a}$ `(new cog C(`a`[ ]))!n1(new cog C(`a`[ ]))`

$\xrightarrow{a}$ $\texttt{t} :^{\top}_{a}$ b`[f:`a`[ ]]!n1(new cog C(`a`[ ]))` (1) (New)

$\xrightarrow{a}$ $\texttt{t} :^{\top}_{a}$ b`[f:`a`[ ]]!n1(`c`[f:`a`[ ]])` (2) (New)

$\xrightarrow{a}$ $\texttt{t} :^{\top}_{a}$ `t1`, $\texttt{t1} :^{\bot}_{b}$ c`[f:`a`[ ]]!m().await.get` (3) (Invk)

$\xrightarrow{b}$ $\texttt{t} :^{\top}_{a}$ `t1`, $\texttt{t1} :^{\top}_{b}$ `t2.await.get`, $\texttt{t2} :^{\bot}_{c}$ `new cog C(f)` (4) (Invk)

$\xrightarrow{c}$ $\texttt{t} :^{\top}_{a}$ `t1`, $\texttt{t1} :^{\top}_{b}$ `t2.await.get`, $\texttt{t2} :^{\top}_{c}$ `new cog C(f)` (5) (Lock)

$\xrightarrow{c}$ $\texttt{t} :^{\top}_{a}$ `t1`, $\texttt{t1} :^{\top}_{b}$ `t2.await.get`, $\texttt{t2} :^{\top}_{c}$ `new cog C(`a`[ ])` (6) (Field)

$\xrightarrow{c}$ $\texttt{t} :^{\top}_{a}$ `t1`, $\texttt{t1} :^{\top}_{b}$ `t2.await.get`, $\texttt{t2} :^{\top}_{c}$ d`[f:`a`[ ]]` (7) (New)

$\xrightarrow{b}$ $\texttt{t} :^{\top}_{a}$ `t1`, $\texttt{t1} :^{\top}_{b}$ `t2.get`, $\texttt{t2} :^{\top}_{c}$ d`[f:`a`[ ]]` (8) (AwaitT)

$\xrightarrow{b}$ $\texttt{t} :^{\top}_{a}$ `t1`, $\texttt{t1} :^{\top}_{b}$ d`[f:`a`[ ]]`, $\texttt{t2} :^{\top}_{c}$ d`[f:`a`[ ]]` (9) (Get)

Nello stato finale i task `t`, `t1` and `t2` termineranno uno dopo l'altro rilasciando il controllo dell'oggetto corrispondente. In questo esempio il pattern di chiamate asincrone delle operazioni di `await` e di `get` viene utiliz-

zato correttamente portando ad una corretta esecuzione dell'espressione da valutare.

2.5.2 Esempio di deadlock

Come secondo esempio vediamo la valutazione dell'espressione `x!n2(x)`, quindi invochiamo il metodo `n2` sull'oggetto `x` passando `x` come argomento. Supponiamo che `x` abbia record $b[\texttt{f} : a[\ ]]$, la valutazione di tale espressione diventa quindi:

$$
\begin{aligned}
&\texttt{t} :^{\top}_{a} b[\texttt{f} : a[\ ]]\texttt{!n2}(b[\texttt{f} : a[\ ]]) \\
&\quad \xrightarrow{a} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\bot}_{b} b[\texttt{f}:a[\ ]]\texttt{!m().get} && \text{(Invk)} \\
&\quad \xrightarrow{b} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{b} b[\texttt{f}:a[\ ]]\texttt{!m().get} && \text{(Lock)} \\
&\quad \xrightarrow{b} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{b} \texttt{t2.get}\ ,\ \texttt{t2} :^{\bot}_{b} \texttt{new cog C(f)} && \text{(Invk)}
\end{aligned}
$$

L'ultimo stato è un esempio di deadlock; `t2` non riceverà mai il controllo sull'oggetto b, che è posseduto da `t1`, il quale a sua volta è bloccato a causa dell'operazione di `get` e non rilascerà mail il lock.

2.5.3 Esempio di creazione infinita

Come terzo esempio vediamo la valutazione dell'espressione `(new cog C(this))!p()`:

$$
\begin{aligned}
&\texttt{t} :^{\top}_{a} b[\texttt{f} : a[\ ]]\texttt{!p}() \\
&\quad \xrightarrow{a} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\bot}_{b} ((\texttt{new cog C(f)})\texttt{!p}()).\texttt{await.get} && \text{(Invk)} \\
&\quad \xrightarrow{a} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\bot}_{b} ((\texttt{new cog C}(a[\ ]))\texttt{!p}()).\texttt{await.get} && \text{(Field)} \\
&\quad \xrightarrow{b} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{b} (c[\texttt{f} : a[\ ]]\texttt{!p}()).\texttt{await.get} && \text{(New)} \\
&\quad \xrightarrow{b} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{b} \texttt{t2.await.get},\ \texttt{t2} :^{\bot}_{c} ((\texttt{new cog C(f)})\texttt{!p}()).\texttt{await.get} && \text{(Invk)} \\
&\quad \xrightarrow{b} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{b} \texttt{t2.await.get},\ \texttt{t2} :^{\bot}_{c} ((\texttt{new cog C}(a[\ ]))\texttt{!p}()).\texttt{await.get} && \text{(Field)} \\
&\quad \xrightarrow{c} \texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{b} \texttt{t2.await.get},\ \texttt{t2} :^{\top}_{c} (d[\texttt{f} : a[\ ]]\texttt{!p}()).\texttt{await.get} && \text{(New)} \\
&\quad \vdots
\end{aligned}
$$

In questo caso non abbiamo uno stato finale, la riduzione prosegue all'infinito creando nuovi nomi di oggetto, eventualmente tutti i task del tipo $\texttt{t(n)} :^{\top}_{\texttt{n}} \texttt{t(n+1).await.get}$ possono rilasciare il controllo del lock con una (AwaitF) ma la computazione comunque prosegue senza sosta.

2.5.4 Esempio di indecisione da scheduler

Come quarto esempio vediamo un caso in cui i deadlock possono essere causati da una scelta dello scheduler rispetto ad un'altra. Si consideri lo stato
$\texttt{t} :^{\top}_{a}$ `(new cog C(new Object))!q(new cog C(new Object),` a`[f:`b`[ ]])`,
dove a è un oggetto della classe `C`. La sua valutazione è la seguente ($\xrightarrow{a}{}^{k}$ significa $\underbrace{\xrightarrow{a} \cdots \xrightarrow{a}}_{k \text{ volte}}$):

$\texttt{t} :^{\top}_{a}$ `(new cog C(new Object))!q(new cog C(new Object),` a`[f:`b`[ ]])`

$\xrightarrow{a}{}^{4}$ $\texttt{t} :^{\top}_{a} (a'[\texttt{f} : b'])\texttt{!q}(a''[\texttt{f} : b''], a[\texttt{f} : b[\ ]])$ (New)

$\xrightarrow{a}\xrightarrow{a'}$ $\texttt{t} :^{\top}_{a} \texttt{t1}, \texttt{t1} :^{\bot}_{a'} (a''[\texttt{f} : b''])\texttt{!n2}(a[\texttt{f} : b[\ ]]); (a[\texttt{f} : b[\ ]])\texttt{!n2}(a''[\texttt{f} : b''])$ (Invk)

$\xrightarrow{a}\xrightarrow{a'}$ $\texttt{t} :^{\top}_{a} \texttt{t1}, \texttt{t1} :^{\top}_{a'} (a''[\texttt{f} : b''])\texttt{!n2}(a[\texttt{f} : b[\ ]]); (a[\texttt{f} : b[\ ]])\texttt{!n2}(a''[\texttt{f} : b''])$ (Lock)

$\xrightarrow{a'}$ $\texttt{t} :^{\top}_{a} \texttt{t1}, \texttt{t1} :^{\top}_{a'} \texttt{t2}; (a[\texttt{f} : b[\ ]])\texttt{!n2}(a''[\texttt{f} : b'']), \texttt{t2} :^{\bot}_{a''} (a[\texttt{f} : b[\ ]])\texttt{!m().get}$ (Invk)

$\xrightarrow{a'}$ $\texttt{t} :^{\top}_{a} \texttt{t1}, \texttt{t1} :^{\top}_{a'} \texttt{t3}, \texttt{t2} :^{\bot}_{a''} (a[\texttt{f} : b[\ ]])\texttt{!m().get}, \texttt{t3} :^{\bot}_{a} (a[\texttt{f} : b[\ ]])\texttt{!n2}(a''[\texttt{f} : b''])$ (Seq)

$\xrightarrow{a'}$ $\texttt{t} :^{\top}_{a} \texttt{t1}, \texttt{t1} :^{\top}_{a'} \texttt{t3}, \texttt{t2} :^{\bot}_{a''} (a[\texttt{f} : b[\ ]])\texttt{!m().get}, \texttt{t3} :^{\bot}_{a} (a''[\texttt{f} : b''])\texttt{!m().get}$ (Invk)

$\xrightarrow{a}$ $\texttt{t} :^{\bot}_{a} \texttt{t1}, \texttt{t1} :^{\top}_{a'} \texttt{t3}, \texttt{t2} :^{\bot}_{a''} (a[\texttt{f} : b[\ ]])\texttt{!m().get}, \texttt{t3} :^{\bot}_{a} (a''[\texttt{f} : b''])\texttt{!m().get}$ (Rel)

$\xrightarrow{a''}{}^{2}$ $\texttt{t} :^{\bot}_{a} \texttt{t1}, \texttt{t1} :^{\top}_{a'} \texttt{t3}, \texttt{t2} :^{\top}_{a''} \texttt{t4.get}, \texttt{t3} :^{\bot}_{a} (a''[\texttt{f} : b''])\texttt{!m().get}, \texttt{t4} :^{\bot}_{a} \texttt{new cog C(f)}$ (Lock)+(Invk)

L'ultimo stato è quello critico: ci sono due task `t3` e `t4` che attendono di ottenere il controllo sull'oggetto a. A seconda della scelta dello scheduler tra `t3` e `t4` si ha rispettivamente deadlock o no.

2.5.5 Esempio di livelock

Come ultimo esempio vediamo un caso di livelock. Definiamo un nuovo metodo della classe `C` per mostrare l'esempio:

```
C h(C b) { return b!n1(this).get ;}
```

e consideriamo la seguente valutazione:

$\texttt{t} :^{\top}_{a}$ `(new cog C(new Object))!h(new cog C(new Object))`

$\xrightarrow{a}{}^{2} \xrightarrow{a}{}^{2}$ $\texttt{t} :^{\top}_{a} a'[\texttt{f} : b'[\,]]\texttt{!h}(a''[\texttt{f} : b''[\,]])$ (New)

$\xrightarrow{a}$ $\texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\bot}_{a'} a''[\texttt{f} : b''[\,]]\texttt{!n1}(a'[\texttt{f} : b'[\,]]).\texttt{get}$ (Invk)

$\xrightarrow{a'}{}^{2}$ $\texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{a'} \texttt{t2.get},\ \texttt{t2} :^{\bot}_{a''} a'[\texttt{f} : b'[\,]]\texttt{!m().await.get}$ (Invk)

$\xrightarrow{a''}{}^{2}$ $\texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{a'} \texttt{t2.get},\ \texttt{t2} :^{\top}_{a''} \texttt{t3.await.get},\ \texttt{t3} :^{\bot}_{a'} \texttt{new cog C}(b'[\,])$ (Lock)

$\xrightarrow{a''}$ $\texttt{t} :^{\top}_{a} \texttt{t1},\ \texttt{t1} :^{\top}_{a'} \texttt{t2.get},\ \texttt{t2} :^{\bot}_{a''} \texttt{t3.await.get},\ \texttt{t3} :^{\bot}_{a'} \texttt{new cog C}(b'[\,])$ (Await)

$\vdots$

Già dal penultimo stato `t1` è bloccato mentre `t2` rilascia e riprende continuamente il lock su a'' aspettando che `t3` termini. Purtroppo però `t3` non otterrà mai il controllo dell'oggetto a' (controllato da `t1`), quindi, non terminerà mai.

Capitolo 3

Inferenza dei contratti

3.1 I contratti

La tecnica di analisi che verrà presentata nel capitolo successivo utilizza una descrizione astratta dei metodi e del comportamento delle espressioni, chiamate rispettivamente *contratti di metodo* e *contratti*. La sintassi di questa descrizione utilizza un insieme infinito di *nomi di record* come X, Y, Z, $\cdots$. I *record futuri* come $\mathbb{r}$, $\mathbb{s}$, $\cdots$, ed i *contratti* come $\mathbb{c}$, $\mathbb{c}'$, $\cdots$ sono definiti dalla grammatica seguente:

$$\begin{array}{lcl} \mathbb{r} & ::= & X \mid a[\bar{\mathtt{f}} : \bar{\mathbb{r}}] \mid a \rightsquigarrow \mathbb{r} \\ \mathbb{c} & ::= & \mathtt{0} \mid \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}' \mid \mid \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}'\mathtt{.}(a, a')^{[\mathtt{a}]} \mid (a, a')^{[\mathtt{a}]} \mid \mathbb{c} \fatsemi \mathbb{c} \end{array}$$

Nella grammatica presentata l'apice in $(a, a')^{[\mathtt{a}]}$ significa che la coppia di oggetti può presentarsi sia nella forma (a, a') che nella forma $(a, a')^{\mathtt{a}}$, vedremo poi quale sia la differenza tra le due notazioni. Un nome di record X rappresenta una variabile che può essere istanziata per sostituzione. Il record futuro $a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$ definisce il nome di oggetto ed i record futuri all'interno dei campi. Il record futuro $a \rightsquigarrow \mathbb{r}$ specifica che, per poter accedere al record $\mathbb{r}$ è necessario acquisire il controllo dell'oggetto di nome a (e di rilasciare il controllo appena il metodo è stato valutato). I record futuri di questo tipo, $a \rightsquigarrow \mathbb{r}$, sono associati all'invocazione di metodo: il nome di oggetto a rappresenta l'oggetto del metodo invocato. Il nome a presente in $a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$ ed in $a \rightsquigarrow \mathbb{r}$ prende il nome di *radice del record futuro* e viene restituito dalla funzione parziale $root(\cdot)$.

Il contratto $\mathbb{c}$ raccoglie le invocazioni di metodo all'interno delle espressioni e le dipendenze tra nomi di oggetto. Un contratto può essere vuoto, denotato da $\mathtt{0}$, specificando che il comportamento del metodo è irrilevante ai

fini dell'analisi; oppure può essere $\texttt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \rightarrow \mathbb{r}'$, specificando che il metodo `m` della classe `C` verrà invocato su di un oggetto $\mathbb{r}$, con argomenti $\bar{\mathbb{r}}$, ed un oggetto $\mathbb{r}'$ verrà restituito; oppure possiamo trovare $\texttt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \rightarrow \mathbb{r}'\texttt{.}(a, a')$, indicando che l'esecuzione del metodo corrente richiede la terminazione del metodo `C.m` eseguito su a' per rilasciare l'oggetto di nome a; oppure $\texttt{C.m}\,\mathbb{r}(\bar{\mathbb{r}}) \rightarrow \mathbb{r}'\texttt{.}(a, a')^{\texttt{a}}$, indicando che l'esecuzione del metodo corrente richiede la terminazione del metodo `C.m` eseguito su a' per continuare (l'oggetto di nome a nel frattempo può essere rilasciato); oppure soltanto (a, a') (rispetto a $(a, a')^{\texttt{a}}$) quando la dipendenza è causata da una operazione di `get` (rispetto ad una `await`) su un campo o su un parametro del metodo (anche `this`), che ha contratto `0`, invece di essere applicata direttamente ad una invocazione di metodo. Le coppie (a, a') e $(a, a')^{\texttt{a}}$ sono decisamente importanti e prendono il nome di *dipendenze tra nomi di oggetto*, saranno le dipendenze su cui si basa il funzionamento del sistema di analisi. Infine, non meno importante, il contratto $\mathbb{c}\,\texttt{;}\,\mathbb{c}'$ definisce il comportamento astratto di una composizione sequenziale di espressioni.

Come esempi di contratti per chiarire le idee vediamo:

(a) $\texttt{C.m}\ a\texttt{[f:}b\texttt{[]]()}\rightarrow a''\texttt{[f:}b\texttt{[]]}\ \texttt{;}\ \texttt{C.m}\ a'\texttt{[f:}b'\texttt{[]]()}\rightarrow b''\texttt{[f:}b'\texttt{[]]}$

(b) $\texttt{C.m}\ a\texttt{[f:}b\texttt{[]]()}\rightarrow a''\texttt{[f:}b\texttt{[]].}(a''', a)\ \texttt{;}\ \texttt{C.m}\ a'\texttt{[f:}b'\texttt{[]]()}\rightarrow b''\texttt{[f:}b'\texttt{[]].}(a''', a')^{\texttt{a}}$

Il contratto (a) definisce una sequenza di due invocazioni del metodo `m` della Tabella 2.4; Il record futuro della prima chiamata è $a[\texttt{f} : b[\,]]$, mentre il record futuro della seconda chiamata è $a'[\texttt{f} : b'[\,]]$. Questi contratti non forzano alcun vincolo sui nomi di oggetti (non creano dipendenze significative) perché il valore delle invocazioni (il risultato) non viene richiesto dal contesto. Come vedremo dopo, una espressione `ABSFJf` con questo contratto è `x!m() ; y!m()`, con `x` e `y` variabili della classe `C`.

Il contratto (b) definisce due invocazioni del metodo `m` proprio come (a) ma, in aggiunta, chiede che il valore della prima invocazione venga acquisito mentre chiede che la seconda invocazione termini. Una espressione `ABSFJf` con questo contratto è `x!m().get ; y!m().await`, con `x` e `y` variabili della classe `C`.

Un record futuro è *lineare* se i nomi di oggetto ed i nomi di record occorrono linearmente. La funzione $names(\cdot)$ restituisce i nomi di oggetto e record. I *contratti di metodo*, chiamati $\mathbb{C}$, $\mathbb{C}'$, $\cdots$, sono termini della forma

$$\mathbb{r}(\bar{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{r}'$$

dove

1. I record futuri $\mathbb{r}$ ed $\bar{\mathbb{s}}$ sono lineari;

2. Nomi di record e di oggetto che occorrono in $\mathbb{r}$ ed in $\bar{\mathbb{s}}$ sono differenti a due a due (per ogni $\mathbb{s} \in \bar{\mathbb{s}}$, $names(\mathbb{r}) \cap names(\mathbb{s}) = \varnothing$ e per argomenti differenti $\mathbb{s}, \mathbb{s}' \in \bar{\mathbb{s}}$, $names(\mathbb{s}) \cap names(\mathbb{s}') = \varnothing$);

3. I nomi di record che occorrono in $\mathbb{c}$ o in $\mathbb{r}'$ sono un sottoinsieme di quelli in $names(\mathbb{r}) \cup names(\bar{\mathbb{s}})$.

È importante specificare come il punto 3. non si applica ai nomi di oggetto che occorrono in $\mathbb{c}$ o in $\mathbb{r}'$, perché possono non occorrere in $names(\mathbb{r}) \cup names(\bar{\mathbb{s}})$ (nuovi oggetti nel body di un metodo possono essere creati e quindi avere nomi nuovi, così detti *fresh*). Il sottotermine $\mathbb{r}(\bar{\mathbb{s}})$ di un contratto di metodo $\mathbb{r}(\bar{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{r}'$ viene chiamato *header*; $\mathbb{r}'$ è chiamato *record futuro di ritorno*. L'header ed il record futuro di ritorno, scritti $\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'$, prendono il nome di *interfaccia*. Si osservi che, in una interfaccia $\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'$, i record $\mathbb{r}$, $\bar{\mathbb{s}}$ e $\mathbb{r}'$ sono soggetti ai vincoli 1., 2. e 3. definiti sopra. Un contratto di metodo può quindi essere visto come una interfaccia di metodo unito al contratto dell'espressione al suo interno (il body). In $\mathbb{r}(\bar{\mathbb{s}})\ \{\mathbb{c}\}\ \mathbb{r}'$, i nomi (di oggetti e record) nell'header *sono legati* ai nomi (di oggetti e record) che compaiono in $\mathbb{c}$ ed in $\mathbb{r}'$. Ad esempio

- Il termine $a[\texttt{f} : b[\,]]()\ \{\texttt{0}\}\ a'[\texttt{f} : b[\,]]$ è il contratto di metodo di `C.m` nell a Tabella 2.4. Il nome b nell'header lega con l'occorrenza b nel record futuro di ritorno. Il nome a' nel record futuro di ritorno è *fresh*, quindi non è legato all'header (è un nome libero). Ciò significa che `m` restituisce un oggetto che ha creato durante l'esecuzione del metodo.

- Il termine $a[\texttt{f} : X](a'[\texttt{f} : b[\,]])\ \{\texttt{C.m}\ a'[\texttt{f} : b[\,]]() \to a''[\texttt{f} : b[\,]].(a, a')\}\ a''[\texttt{f} : b[\,]]$ é il contratto del metodo `C.n1` della Tabella 2.4. Alcune osservazioni sono: (i) il campo `f` dell'oggetto di `n` non viene mai letto o usato nel body (non si hanno accessi a questo campo); per tale ragione abbiamo un nome di record X invece di un record futuro; (ii) i nomi a, a' ed b nell'header del contratto di metodo legano con le occorrenze dei nomi di oggetto nel body e nel record futuro da restituire; (iii) il nome a'' nel record futuro di ritorno è *fresh*.

3.2 Sostituzione

Una sostituzione σ tra record futuri, è una funzione finita che mappa nomi di oggetto in nomi di oggetto e mappa nomi di record in record futuri. Ad esempio $[^{a'}/_{a}][^{b[\bar{\mathtt{f}} : \bar{\mathbb{r}}]}/_{X}]$ sta ad indicare che la sostituzione associa a ad a', ed X a $b[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$. L'applicazione di una sostituzione ad un determinato oggetto viene definita secondo le regole standard come segue:

$$\sigma(X) = \begin{cases} \mathbb{r} & \text{se } \sigma(X) = \mathbb{r} \\ X & \text{se } X \notin dom(\sigma) \end{cases}$$

$$\sigma(a) = \begin{cases} b & \text{se } \sigma(a) = b \\ a & \text{se } a \notin dom(\sigma) \end{cases}$$

$$\sigma(a[\bar{\mathtt{f}} : \bar{\mathbb{r}}]) = \sigma(a)[\bar{\mathtt{f}} : \sigma(\bar{\mathbb{r}})]$$

$$\sigma(a \rightsquigarrow \mathbb{r}) = \sigma(a) \rightsquigarrow \sigma(\mathbb{r})$$

Una sostituzione di record si estende ovviamente anche a sequenze di oggetti

$$\sigma(\mathbb{r}_1, \ldots, \mathbb{r}_n) = \sigma(\mathbb{r}_1), \ldots, \sigma(\mathbb{r}_n)$$

ed ai contratti. In aggiunta, quando $\mathbb{r}$ è lineare, definiamo $\mathbb{s}[^{\mathbb{r}'}/_{\mathbb{r}}]$ per induzione sulla struttura di $\mathbb{r}$:

$$\mathbb{s}[^{\mathbb{r}'}/_{\mathbb{r}}] = \begin{cases} \mathbb{s}[^{\mathbb{r}'}/_{X}] & \mathbb{r} = X \\ \mathbb{s}[^{b}/_{a}][^{\bar{\mathbb{r}}'}/_{\bar{\mathbb{r}}}] & \mathbb{r} = a[\bar{\mathtt{f}} : \bar{\mathbb{r}}] \text{ ed } \mathbb{r}' = b[\bar{\mathtt{f}} : \bar{\mathbb{r}}'] \\ \mathbb{s}[^{b}/_{a}][^{\mathbb{r}'''}/_{\mathbb{r}''}] & \mathbb{r} = a \rightsquigarrow \mathbb{r}'' \text{ ed } \mathbb{r}' = b \rightsquigarrow \mathbb{r}''' \end{cases}$$

Da notare come $\mathbb{s}[^{\mathbb{r}'}/_{\mathbb{r}}]$ sia una funzione parziale, quindi non sempre definita, in quanto richiede il *matching* tra le strutture di $\mathbb{r}$ e di $\mathbb{r}'$; ad esempio $\mathbb{s}[^{b[\,]}/_{a \rightsquigarrow X}]$ e $\mathbb{s}[^{a \rightsquigarrow X}/_{b[\,]}]$ non sono definite proprio per l'assenza di matching.

Anche la composizione di sostituzioni σ e σ', scritta $\sigma \circ \sigma'$ viene definita in maniera standard come $(\sigma \circ \sigma')(\mathbb{r}) = \sigma'(\sigma(\mathbb{r}))$.

Nel seguente sistema di tipi, utilizzeremo l'operatore $\between$ per la composizione di contratti definito come segue:

$$
\begin{array}{rcll}
\mathtt{0} \between (a,a')^{[\mathtt{a}]} & = & (a,a')^{[\mathtt{a}]} & (1)\\
\mathbb{c} \fatsemi \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}' \between (a,a')^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\mathtt{.}(a,a')^{[\mathtt{a}]}) & (2)\\
\mathbb{c} \fatsemi (a,a') \between (a,b)^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (a,a') \fatsemi (a,b)^{[\mathtt{a}]} & (3)\\
\mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\mathtt{.}(a,a')) \between (a,b)^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\mathtt{.}(a,a')) \fatsemi (a,b)^{[\mathtt{a}]} & (4)\\
\mathbb{c} \fatsemi (a,a')^{\mathtt{a}} \between (a,a')^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (a,a')^{\mathtt{a}} & (5)\\
\mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\mathtt{.}(a,a')^{\mathtt{a}}) \between (a,a')^{[\mathtt{a}]} & = & \mathbb{c} \fatsemi (\mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'\mathtt{.}(a,a')^{\mathtt{a}}) & (6)
\end{array}
$$

Lo scopo di $\between$ è quello, come abbiamo accennato, di gestire l'accumulazione di coppie della forma $(a,b)^{[\mathtt{a}]}$, in particolare, la regola (1) si applica quando una operazione di `get` o di `await` viene eseguita su un campo o un parametro del metodo, che ricordiamo avere contratto `0`. La regola (2) si applica quando una operazione di `get` o di `await` viene eseguita direttamente su di una invocazione di metodo. Le regole (3) e (4) si applicano quando le espressioni da tipare sono della forma `e.get.get` oppure `e.get.await`[1]. In questo caso la prima operazione di `get` richiede il valore di ritorno, che deve essere un dato di tipo futuro per la `get` successiva (così come per l'operazione di `await`). Quindi la nuova coppia di dipendenza di nomi (a,b) viene separata dalla precedente (a,a') perché non contribuisce a bloccare il risultato della prima `get`. La dipendenza (a,b) contribuisce ad una nuova catena di dipendenze. Ricordiamo come la prima componente delle due coppie sia la stessa (a) perché si riferisce all'oggetto corrente (che appunto è lo stesso). Le regole (5) e (6) si applicano quando le espressioni da tipare sono della forma `e.await.await` ed `e.await.get`; in questo caso la nuova coppia di dipendenza di nomi viene ignorata, infatti se la prima operazione di `await` ha successo, nessun lock può crearsi a seguito della seconda operazione. In aggiunta, se la prima operazione di `await` portasse ad un lock, allora la seconda operazione non verrebbe mai eseguita. Il comportamento della composizione quindi è dettato della prima `await`.

Siccome i contratti di metodo contengono legami tra nomi di record e di oggetti, identifichiamo i termini che sono equivalenti grazie ad un "renaming" (di nomi legati e liberi) con l'equivalenza $=^{\alpha}$. Per esempio:

$$a[\mathtt{f} : b[\,]]()\ \{\mathtt{0}\}\ a'[\mathtt{f} : b[\,]] =^{\alpha} b[\mathtt{f} : b'[\,]]()\ \{\mathtt{0}\}\ a[\mathtt{f} : b'[\,]]$$

ed

$$
\begin{array}{rl}
a[\mathtt{f} : X](a'[\mathtt{f} : b[\,]])\ \{\mathtt{E.m}\ a'[\mathtt{f} : b[\,]]() \to a''[\mathtt{f} : b[\,]]\mathtt{.}(a,a')\}\ a''[\mathtt{f} : b[\,]] & \\
\quad =^{\alpha} \quad b[\mathtt{f} : Z](c[\mathtt{f} : a[\,]])\ \{\mathtt{E.m}\ c[\mathtt{f} : a[\,]]() \to d[\mathtt{f} : a[\,]]\mathtt{.}(b,c)\}\ d[\mathtt{f} : a[\,]] &
\end{array}
$$

[1] Si noti come, anche in questo contesto, l'utilizzo della sintassi `FJf` sia dovuto solo alla leggibilità, un'espressione della forma `e.await.get` nella sintassi pura `ABSFJf` diverrebbe `x := e; await x?; x.get`

3.3 Il sistema di tipi dei contratti

La valutazione dei tipi fa affidamento all'*ambiente* Γ che lega variabili a coppie della forma $(\mathtt{T}, \mathbb{r})$ e lega metodi $\mathtt{C.m}$ ad interfacce $\mathbb{r}(\bar{\mathbb{s}}) \to \mathbb{r}'$. La valutazione ha la forma

$$\Gamma \vdash_a \mathtt{e} : (\mathtt{T}, \mathbb{r}) \,,\ \mathbb{c} \rhd \mathcal{U}$$

e si legge: l'espressione $\mathtt{e}$ che occorre in un metodo di un oggetto con root a ha il tipo $\mathtt{T}$, record futuro $\mathbb{r}$, e contratto $\mathbb{c}$ nell'ambiente Γ. $\mathcal{U}$ è l'insieme dei vincoli ed ha valenza solo nelle regole di inferenza dei contratti che vedremo nella sezione successiva. La valutazione delle espressioni sono definite nella Tabella 3.1. In questa tabella utilizziamo la stessa notazione $\bar{\mathbb{c}}$ per indicare una tupla di contratti $(\mathbb{c}_1, \cdots, \mathbb{c}_n)$, quando il simbolo compare nella valutazione di un insieme di espressioni $\bar{\mathtt{e}}$, ma anche per indicare la composizione sequenziale di contratti $\mathbb{c}_1 \fatsemi \cdots \fatsemi \mathbb{c}_n$, quando il simbolo compare nella valutazione di una singola espressione.

La regola (T-Field) definisce la valutazione dell'accesso a un campo, ciò può accadere solo all'interno di un body di metodo in cui troviamo la variabile `this`. La regola vincola il `this` ad avere un tipo di classe `C` (non un futuro) e di avere un record futuro come $a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$, da cui facilmente estrarre il record del parametro (campo) `f`.

La regola (T-Invk) definisce la valutazione di una chiamata di metodo $\mathtt{e!m}(\bar{\mathtt{e}})$. Supponiamo che $a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{r}$ sia l'interfaccia di $\mathtt{C.m}$ contenuta in Γ. I nomi di record e di oggetto nell'interfaccia devono essere rimpiazzati dai valori attuali, quindi, per tipare $\mathtt{e!m}(\bar{\mathtt{e}})$, deve esistere una sostituzione σ tale che $\Gamma \vdash_a \mathtt{e} : (\mathtt{C}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]))$, $\mathbb{c}$ e $\Gamma \vdash_a \bar{\mathtt{e}} : (\bar{\mathtt{T}}, \sigma(\bar{\mathbb{s}}))$, $\bar{\mathbb{c}}$. (è possibile utilizzare un'unica σ se i nomi in $a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]$ e $\bar{\mathbb{s}}$ sono disgiunti). Il tipo (standard) di $\mathtt{e!m}(\bar{\mathtt{e}})$ è un tipo futuro $\mathtt{Fut}(\mathtt{T}')$. Il record futuro di $\mathtt{e!m}(\bar{\mathtt{e}})$ è $\sigma(a') \rightsquigarrow \sigma(\mathbb{r})$ indicando che il valore possa essere recuperato non appena il controllo di $\sigma(a')$ venga acquisito. Il contratto di $\mathtt{e!m}(\bar{\mathtt{e}})$ è abbastanza scontato: compone in sequenza i contratti di $\mathtt{e}$, $\bar{\mathtt{e}}$ e della invocazione di metodo.

La regola (T-New) tipa la creazione di oggetto che, nel sistema di tipi, prevede l'utilizzo di un nuovo nome di oggetto fresh (chiamato a' nella regola) come root del record futuro. Le restanti parti della valutazione di tale regola sono standard.

Le regole (T-Get) e (T-Await) definiscono un tipo per le operazioni di `e.get` ed `await e?`. In questi casi il tipo di `e` deve essere `Fut(T)` e, parallelamente, il tipo di record futuro deve essere strutturato come $a' \rightsquigarrow \mathbb{s}$. Nel caso di (T-Get), il tipo di `e.get` viene ridotto a $(\mathtt{T}, \mathbb{s})$, mentre, in caso di `await e?`, il tipo non cambia. Per quanto riguarda i contratti, (T-Get) e

$$\text{(T-Var)}\quad \Gamma \vdash_a \mathtt{x} : \Gamma(\mathtt{x}),\ \mathtt{0} \qquad \text{(T-Field)}\quad \frac{\Gamma \vdash_a \mathtt{this} : (\mathtt{C}, a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]),\ \mathbb{c} \quad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}\,\bar{\mathtt{f}} \quad \mathtt{T}\,\mathtt{f} \in \bar{\mathtt{T}}\,\bar{\mathtt{f}} \quad \mathtt{f} : \mathbb{r} \in \bar{\mathtt{f}} : \bar{\mathbb{r}}}{\Gamma \vdash_a \mathtt{f} : (\mathtt{T}, \mathbb{r}),\ \mathbb{c}}$$

$$\text{(T-Invk)}\quad \frac{\begin{array}{c}\Gamma(\mathtt{C.m}) = a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{r} \\ \Gamma \vdash_a \mathtt{e} : (\mathtt{C}, \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])),\ \mathbb{c} \qquad \Gamma \vdash_a \bar{\mathtt{e}} : (\bar{\mathtt{T}}, \sigma(\bar{\mathbb{s}})),\ \bar{\mathbb{c}} \\ \mathit{mtype}(\mathtt{m}, \mathtt{C}) = \bar{\mathtt{T}} \to \mathtt{T}'\end{array}}{\Gamma \vdash_a \mathtt{e!m}(\bar{\mathtt{e}}) : (\mathtt{Fut}(\mathtt{T}'), \sigma(a') \rightsquigarrow \sigma(\mathbb{r})),\ \mathbb{c} \fatsemi \bar{\mathbb{c}} \fatsemi \mathtt{C.m}\ \sigma(a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}])(\sigma(\bar{\mathbb{s}})) \to \sigma(\mathbb{r})}$$

$$\text{(T-New)}\quad \frac{\Gamma \vdash_a \bar{\mathtt{e}} : (\bar{\mathtt{T}}, \bar{\mathbb{r}}),\ \bar{\mathbb{c}} \quad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{T}}\,\bar{\mathtt{f}} \quad a' \mathit{\ fresh}}{\Gamma \vdash_a \mathtt{new\ cog\ C}(\bar{\mathtt{e}}) : (\mathtt{C}, a'[\bar{\mathtt{f}} : \bar{\mathbb{r}}]),\ \bar{\mathbb{c}}}$$

$$\text{(T-Get)}\quad \frac{\Gamma \vdash_a \mathtt{e} : (\mathtt{Fut}(\mathtt{T}), a' \rightsquigarrow \mathbb{s}),\ \mathbb{c}}{\Gamma \vdash_a \mathtt{e.get} : (\mathtt{T}, \mathbb{s}),\ \mathbb{c} \between (a, a')} \qquad \text{(T-Await)}\quad \frac{\Gamma \vdash_a \mathtt{e} : (\mathtt{Fut}(\mathtt{T}), a' \rightsquigarrow \mathbb{s}),\ \mathbb{c}}{\Gamma \vdash_a \mathtt{await\ e?} : (\mathtt{Fut}(\mathtt{T}), a' \rightsquigarrow \mathbb{s}),\ \mathbb{c} \between (a, a')^{\mathtt{a}}}$$

$$\text{(T-Seq)}\quad \frac{\Gamma \vdash_a \mathtt{e} : (\mathtt{T}, \mathbb{r}),\ \mathbb{c} \qquad \Gamma \vdash_a \mathtt{e}' : (\mathtt{T}', \mathbb{r}'),\ \mathbb{c}'}{\Gamma \vdash_a \mathtt{e}\,;\,\mathtt{e}' : (\mathtt{T}', \mathbb{r}'),\ \mathbb{c} \fatsemi \mathbb{c}'}$$

Tabella 3.1: Regole di tipaggio delle principali espressioni ABSFJf

(T-Await) estendono il contratto di e con la coppia (a, a') o $(a, a')^{\mathtt{a}}$, rispettivamente, dove l'indice a della valutazione definisce il primo elemento della dipendenza tra nomi, tale nome corrisponde al nome del `this` all'interno del metodo di cui stiamo valutando l'espressione di `e.get` o `await e?`. L'elemento a' della coppia di dipendenza di nomi corrisponde al root del record futuro di e.

La Tabella 3.2 estende le regole della Tabella 3.1 per tipare i metodi e le classi. Sia $\mathit{mname}(\bar{\mathtt{M}})$ la sequenza di nomi di metodo in $\bar{\mathtt{M}}$. La regola (T-Method) definisce il tipo ed il contratto di un metodo. Risulta simile alla corrispondente regola in Featherweight Java.

(T-Method)

$$\frac{\Gamma(\texttt{C.m}) = a[\bar{\texttt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}}) \to \mathbb{r}' \qquad \Gamma + \bar{\texttt{x}} : (\bar{\texttt{T}}, \bar{\mathbb{s}}) + \texttt{this} : (\texttt{C}, a[\bar{\texttt{f}} : \bar{\mathbb{r}}]) \vdash_a \texttt{e} : (\texttt{T}, \mathbb{r}')\,,\ \mathbb{c} \quad \texttt{m} \in \texttt{C} \ \text{ imply } \ \mathit{mtype}(\texttt{m}, \texttt{C}) = \bar{\texttt{T}} \to \texttt{T}}{\Gamma \vdash\ \texttt{T m}\ (\bar{\texttt{T}}\ \bar{\texttt{x}})\{\texttt{return e ;}\} : a[\bar{\texttt{f}} : \bar{\mathbb{r}}](\bar{\mathbb{s}})\{\mathbb{c}\}\ \mathbb{r}' \text{ IN } \texttt{C}}$$

(T-Class)

$$\frac{\Gamma \vdash \bar{\texttt{M}} : \bar{\mathbb{C}} \text{ IN } \texttt{C}}{\Gamma \vdash \texttt{class C}(\bar{\texttt{C}}\ \bar{\texttt{f}})\ \texttt{implements C}\ \{\bar{\texttt{M}}\} : \{\mathit{mname}(\bar{\texttt{M}}) \mapsto \bar{\mathbb{C}}\}}$$

Tabella 3.2: Regole di tipaggio per la dichiarazione di metodo e la dichiarazione di classe

La *contract class table*, chiamata CCT, di un programma `ABSFJf` $(\text{CT}, \texttt{s})$ è una mappa $\texttt{C} \mapsto \{\bar{\texttt{m}} \mapsto \bar{\mathbb{C}}\}$, con $\mathit{dom}(\text{CCT}) = \mathit{dom}(\text{CT})$, che soddisfa i seguenti vincoli:

esiste un ambiente Γ tale che

(a) $\Gamma \vdash \text{CT}(\texttt{C}) : \text{CCT}(\texttt{C})$, per ogni `C`;

(b) $\Gamma(\texttt{C.m})$ è l'interfaccia di $\text{CCT}(\texttt{C})(\texttt{m})$, per ogni `C.m`;

3.4 Il sistema di inferenza

Così com'è, il sistema di tipi per i contratti presentato nelle Tabelle 3.1 e 3.2 non permette di inferire, quindi calcolare, la *contract class table* CCT. In particolare non è specificato come definire l'interfaccia $\Gamma(\texttt{C.m})$ che corrisponda al contratto di metodo in $\text{CCT}(\texttt{C})(\texttt{m})$. Attualmente, la soluzione a questo tipo di problema è piuttosto standard e si basa sul concetto di unificazione. In questa sezione discuteremo alcune regole utilizzate nell'algoritmo di inferenza.

L'algoritmo di inferenza dei contratti comincia con un ambiente che fa corrispondere a ogni metodo `C.m` una interfaccia $a_{\texttt{C.m}}[\texttt{f} : \overline{X_{\texttt{C.m}}}](\overline{Y_{\texttt{C.m}}}) \to Z_{\texttt{C.m}}$ tale che interfacce associate a coppie differenti `C.m` non hanno nomi (di record e di oggetto) in comune.

Le regole di inferenza dei contratti per le espressioni sono presentate nella Tabella 3.3, si utilizza una funzione *class*() che data una variabile restituisce la sua classe di appartenenza, si noti come in tutte le regole non vengono considerati i tipi di dato ma solo i record, questo accade per due motivi:

1. Compaiono già nel sistema di tipi delle Tabelle 3.1 e 3.2 e sarebbero

ridondanti;

2. Ai fini dell'implementazione dell'algoritmo di inferenza dei contratti possiamo disinteressarci del corretto tipaggio delle espressioni in quanto `ABS` (e di conseguenza `ABSFJf`) possiede già un type-checker.

In questo modo le regole sono meno "verbose" e più leggibili:

- La regola (T-VAR) risulta banale ed è un assioma, si recupera il record della variabile direttamente dall'ambiente mentre contratto e vincoli sono nulli;

- La regola (T-FIELD) viene eseguita all'interno di un metodo, quindi nell'ambiente abbiamo a disposizione la variabile `this`. Viene recuperata la struttura della classe del `this` per creare un record della stessa struttura con un nome fresh, popolato con campi fresh ed eguagliato al contenuto della variabile `this` nell'ambiente. In questo modo il record di `f` sarà il corrispettivo record del campo di nome `f` all'interno del `this`

- La regola (T-INVK) utilizza una strategia simile a quella della regola (T-FIELD) per ottenere un vincolo sul nome di oggetto invece che su un campo. Si crea infatti un nome a', si crea un record della stessa struttura di quello dell'espressione `e` (che sarà il `this` del metodo) e si aggiunge il vincolo $a'[\bar{\mathtt{f}} : \overline{Y}] = \mathbb{r}$ con $\mathbb{r}$ record di `e`. Si aggiunge inoltre il vincolo $\Gamma(\mathtt{C.m}) \preceq \mathbb{r}(\bar{\mathbb{s}}) \to Y$ che consente all'insieme dei vincoli di poter raffinare l'interfaccia del metodo (parleremo in seguito del vincolo di disuguaglianza). È grazie a questa procedura che al termine della computazione la CCT avrà al suo interno delle interfacce correttamente istanziate. Sempre in questa regola aggiungiamo ai contratti dell'oggetto chiamante e dei parametri il contratto ottenuto dall'invocazione di metodo.

- Nelle regole (T-GET) e (T-AWAIT) si utilizza l'operatore $\wr$ definito in precedenza per accumulare contratti. Tutte e due definiscono il medesimo vincolo $a' \rightsquigarrow X = \mathbb{s}$ dove $\mathbb{s}$ è il record dell'espressione principale eguagliato alla struttura $a' \rightsquigarrow X$; in tal modo si recupera la *root* di $\mathbb{s}$ utilizzando a' al suo posto per costruire i contratti.

- La regola (T-NEW) crea semplicemente un nuovo oggetto recuperando la corretta struttura della classe `C`.

(T-Var)
$$\Gamma \vdash_a \mathtt{x} : \Gamma(\mathtt{x}),\ \mathtt{0} \rhd \varnothing$$

(T-Field)
$$\frac{\Gamma + \mathtt{this} : \mathbb{r} \vdash_a \quad \mathtt{f} \in \bar{\mathtt{f}} \quad \bar{\mathtt{f}} = \mathit{fields}(\mathit{class}(\mathtt{this})) \quad \mathtt{f} : Y \in \bar{\mathtt{f}} : \bar{Y} \quad a', \bar{Y}\ \mathit{fresh}}{\Gamma \vdash_a \mathtt{f} : Y,\ \mathbb{c} \rhd \mathcal{U} \cup \{a'[\bar{\mathtt{f}} : \bar{Y}] = \Gamma(\mathtt{this})\}}$$

(T-Invk)
$$\frac{\Gamma \vdash_a \mathtt{e} : \mathbb{r},\ \mathbb{c} \rhd \mathcal{U} \qquad \Gamma \vdash_a \bar{\mathtt{e}} : \bar{\mathbb{s}},\ \bar{\mathbb{c}} \rhd \bar{\mathcal{U}} \quad \mathit{fields}(\mathit{class}(\mathtt{e})) = \bar{\mathtt{f}} \quad a', Y, \overline{Y}\ \mathit{fresh}}{\Gamma \vdash_a \mathtt{e!m}(\bar{\mathtt{e}}) : a' \rightsquigarrow Y,\ \mathbb{c} \fatsemi \bar{\mathbb{c}} \fatsemi \mathtt{C.m}\ \mathbb{r}(\bar{\mathbb{s}}) \rhd \mathcal{U} \cup \bar{\mathcal{U}} \cup \{a'[\bar{\mathtt{f}} : \overline{Y}] = \mathbb{r},\ \ \Gamma(\mathtt{C.m}) \preceq \mathbb{r}(\bar{\mathbb{s}}) \to Y\}}$$

(T-New)
$$\frac{\Gamma \vdash_a \bar{\mathtt{e}} : \bar{\mathbb{s}},\ \bar{\mathbb{c}} \rhd \bar{\mathcal{U}} \quad \mathit{fields}(\mathtt{C}) = \bar{\mathtt{f}} \quad a'\ \mathit{fresh}}{\Gamma \vdash_a \mathtt{new\ cog\ C}(\bar{\mathtt{e}}) : a'[\bar{\mathtt{f}} : \bar{\mathbb{s}}],\ \bar{\mathbb{c}} \rhd \bar{\mathcal{U}}}$$

(T-Get)
$$\frac{\Gamma \vdash_a \mathtt{e} : \mathbb{s},\ \mathbb{c} \rhd \mathcal{U} \quad a', X\ \mathit{fresh}}{\Gamma \vdash_a \mathtt{e.get} : X,\ \mathbb{c} \between (a, a') \rhd \mathcal{U} \cup \{a' \rightsquigarrow X = \mathbb{s}\}}$$

(T-Await)
$$\frac{\Gamma \vdash_a \mathtt{e} : \mathbb{s},\ \mathbb{c} \rhd \mathcal{U} \quad a', X\ \mathit{fresh}}{\Gamma \vdash_a \mathtt{await\ e?} : \mathbb{s},\ \mathbb{c} \between (a, a')^{\mathtt{a}} \rhd \mathcal{U} \cup \{a' \rightsquigarrow X = \mathbb{s}\}}$$

(T-Seq)
$$\frac{\Gamma \vdash_a \mathtt{e} : \mathbb{r},\ \mathbb{c} \rhd \mathcal{U} \qquad \Gamma \vdash_a \mathtt{e}' : \mathbb{r}',\ \mathbb{c}' \rhd \mathcal{U}'}{\Gamma \vdash_a \mathtt{e}\,;\,\mathtt{e}' : \mathbb{r}',\ \mathbb{c} \fatsemi \mathbb{c}' \rhd \mathcal{U} \cup \mathcal{U}'}$$

(T-Ret)
$$\frac{\Gamma \vdash_a \mathtt{e} : \mathbb{r},\ \mathbb{c} \rhd \mathcal{U}}{\Gamma \vdash_a \mathtt{return\ e} : \varnothing,\ \mathbb{c} \rhd \mathcal{U} \cup \{\Gamma(\mathtt{ret}) = \mathbb{r}\}}$$

(T-Method)
$$\frac{\Gamma + \mathtt{this} : a[\bar{f} : \bar{X}] + \bar{\mathtt{x}} : (\bar{\mathtt{T}}, \bar{Y}) + \mathtt{ret} : \mathtt{Y} \vdash_a \bar{\mathtt{s}} : \varnothing,\ \mathbb{c} \rhd \mathcal{U} \quad a, \bar{\mathtt{X}}, \bar{\mathtt{Y}}, \mathtt{Y}\ \mathit{fresh} \quad \mathtt{m} \in \mathit{class}(\mathtt{this})}{\Gamma \vdash \mathtt{m}\ (\bar{\mathtt{x}})\{\bar{\mathtt{s}}\ ;\} : ,\ a[\bar{f} : \bar{X}](\bar{Y})\{\mathbb{c}\}\ \mathbb{r} \rhd \mathcal{U} \cup \{a[\bar{f} : \bar{X}](\bar{Y}) \to Y = \Gamma(\mathtt{C.m})\} \quad \text{IN } \mathtt{C}}$$

(T-Class)
$$\frac{\Gamma \vdash \bar{\mathtt{M}} : ,\ \bar{\mathbb{C}} \rhd \bar{\mathcal{U}}\ \text{IN } \mathtt{C}}{\Gamma \vdash \mathtt{class\ C\ implements\ C}\ \{\bar{\mathtt{M}}\} : ,\ \{\mathit{mname}(\bar{\mathtt{M}}) \mapsto \bar{\mathbb{C}}\} \rhd \bar{\mathcal{U}}}$$

Tabella 3.3: Regole di inferenza dei contratti in ABSFJf

- La regola (T-Ret) anticipa alcuni problemi di implementazione. In `ABSFJf` all'interno di un body di metodo ci sono solo statement in cui le espressioni vengono utilizzate al loro interno. Gli statement sono costrutti che non hanno record, mentre le espressioni sì. Per poter quindi recuperare il record dell'espressione di ritorno partendo dallo statement `return` si è scelto di battezzare una variabile `ret` all'interno dell'ambiente e vincolarla al valore dell'espressione di ritorno. Si noti infatti come nella regola (T-Ret) tale variabile sia utilizzate nella costruzione dell'interfaccia (per ottenere appunto il valore di ritorno).

Prima di proseguire definiamo alcuni punti chiave dell'unificazione:
Una sostituzione σ *unifica*

- *una equazione tra record* $\mathbb{r} \preceq \mathbb{s}$ se $\sigma(\mathbb{r}) = \mathbb{s}$ sono identiche;
- *una equazione tra interfacce* $\mathbb{r}(\bar{\mathbb{r}}) \to \mathbb{r}' \preceq \mathbb{s}(\bar{\mathbb{s}}) \to \mathbb{s}'$ se σ unifica $\mathbb{r} = \mathbb{s}$ e $\bar{\mathbb{r}} = \bar{\mathbb{s}}$ e $\mathbb{r}' = \mathbb{s}'$ a meno di alpha-conversione (renaming);

Una sostituzione σ è un *more general unifier per* σ' se esiste σ'' tale che $\sigma' = \sigma \circ \sigma''$, si dice quindi che σ è "più generale" di σ'. Una sostituzione σ è una *most general unifier*, scritto mgu, se è più generale di ogni altra sostituzione.

3.5 Semiunificazione

3.5.1 Preliminari sulla semiunificazione

Un termine $\mathtt{M}$ sussume $\mathtt{N}$ ($\mathtt{N}$ è istanza di $\mathtt{M}$) se esiste una sostituzione σ tale che

$$\sigma(\mathtt{M}) = \mathtt{N}$$

in tal caso si scrive $\mathtt{M} \preceq \mathtt{N}$. Dato un insieme di disequazioni

$$\mathtt{M_1} \preceq \mathtt{N_1}; \cdots ; \mathtt{M_k} \preceq \mathtt{N_k}$$

con $k > 0$, una sostituzione σ si dice semiunificatore delle suddette disequazioni se sono soddisfatte le condizioni

$$\sigma(\mathtt{M_1}) \preceq \sigma(\mathtt{N_1}); \cdots ; \sigma(\mathtt{M_k}) \preceq \sigma(\mathtt{N_k})$$

cioè se esistono le sostituzioni $\rho_1, \ldots, \rho_k$ tali che

$$\rho_1(\sigma(\mathtt{M_1})) = \sigma(\mathtt{N_1}); \ldots ; \rho_\mathtt{k}(\sigma(\mathtt{M_k})) = \sigma(\mathtt{N_k})$$

Si possono anche scrivere sistemi di disequazioni che sono in tupla tra loro, ad esempio

$$(\mathtt{M_1} \preceq \mathtt{N_1}; \ldots ; \mathtt{M_k} \preceq \mathtt{N_k})$$

la differenza con il sistema non in tupla è che invece di avere $\mathtt{k}$ sostituzioni $\rho_1, \ldots, \rho_k$ possibilmente distinte, in questo caso vi è un' unica sostituzione ρ per tutte le disequazioni:

$$\rho(\sigma(\mathtt{M_1})) = \sigma(\mathtt{N_1}); \ldots ; \rho(\sigma(\mathtt{M_k})) = \sigma(\mathtt{N_k})$$

E' inoltre noto che ogni sistema di disequazioni, se risolvibile, ammette un unico semiunificatore più generale, ovvero esiste un'unica sostituzione σ che, oltre a essere essa stessa soluzione del sistema, gode della proprietà di essere la più generale (il così detto mgu). Il problema di trovare la sostituzione σ a partire dalle disequazioni proposte è detto problema della semiunificazione [12].

3.5.2 Semiunificazione in ABSFJf

Nella Tabella 3.3 sono state presentate le regole di inferenza delle espressioni in `ABSFJf`. Analizzando alcune regole si nota come non vengano usate sostituzioni ma vengano raccolti vincoli all'interno di $\mathcal{U}$. La maggior parte delle regole utilizza vincoli di uguaglianza per costringere nomi di oggetto o record futuri a recuperare una certa informazione (si vedano le regole rispettivamente di (T-INVK) e (T-FIELD)). Viene però utilizzato anche un vincolo di disuguaglianza, nella regola (T-INVK), infatti ciò che si vuole ottenere è un continuo raffinamento dell'interfaccia del metodo chiamato introducendo nuove informazioni nella corrispondente interfaccia all'interno dell'ambiente. È l'espressione con cui si invoca il metodo che influenza l'interfaccia nell'ambiente e non il contrario, in questo modo l'interfaccia del metodo si specifica quanto basta durante l'algoritmo di inferenza. Alcune parti dell'interfaccia (ad esempio dei parametri del metodo) potrebbero rimanere variabili e non essere istanziate da record futuri se queste non venissero utilizzate mai nel codice.

A causa di questa necessità di disuguaglianza ($\preceq$) tra termini, per arrivare ad un corretto calcolo della CCT, è stato necessario utilizzare un algoritmo di semiunificazione e non l'unificazione standard.

L'algoritmo è stato creato partendo dall'articolo [12]. Così come un normale algoritmo di unificazione gestisce la creazione di termini generici `t` definiti in modo standard come segue:

$$\mathtt{t} := \mathtt{x} \mid \mathtt{c}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$$

dove $\mathtt{t}, \mathtt{t}_1, \ldots, \mathtt{t}_n$ sono termini generici, `x` è una generica variabile (il dominio delle variabili $\mathtt{x}, \mathtt{y}, \ldots$ si assume essere illimitato) mentre `c` è il costruttore di termine, dichiara la sua struttura attribuendogli un nome. Anche in questo caso, come nella normale unificazione, è possibile creare termini di vario tipo; per facilitare l'inferenza sono state create classi per costruire velocemente termini strutturati come i contratti, le interfacce e i contratti di metodo. Lo stesso algoritmo di semiunificazione è stato esteso per gestire facilmente l'accumulo dei contratti (l'operatore $\between$ visto in precedenza) e la stampa di questi per facilitare la lettura dell'output dell'inferenza. Come abbiamo già più di una volta notato nelle regole della Tabella 3.3 abbiamo solo vincoli e non sostituzioni, con l'algoritmo di semiunificazione abbiamo a disposizione la classe `Constraint`, che ci consente di creare, aggiungere e risolvere vincoli incontrati durante l'esecuzione dell'algoritmo di inferenza (i vincoli sono chiaramente di due tipi, vincolo di uguaglianza $=$ e vincolo di semiuguaglianza $\preceq$). Al termine dell'inferenza si ha a disposizione un vincolo globale la cui risoluzione restituisce una sostituzione, il risultato della

semiunificazione. La sostituzione ottenuta applicata ai contratti di metodo porta ad un calcolo corretto della CCT, la struttura dati principale che verrà usata dall'algoritmo di analisi che presenteremo nel prossimo capitolo.

3.6 Dettagli implementativi

In questo capitolo vediamo alcuni dettagli sull'implementazione dell'algoritmo di inferenza. Il linguaggio `ABS` (di cui `ABSFJf` è un puro nucleo sintattico) possiede già diversi tool implementati che lo rendono un linguaggio completo, ciò che più interessa ai fini del nostro risultato sono il parser, e il type-checker:

- Il **Parser** analizza la struttura di un programma `ABSFJf` e ne produce il corrispondente albero sintatto astratto (`AST` fig. 3.1);

- Il **Type-checker** analizza l'`AST` prodotto dal parser per verificare il corretto tipaggio delle espressioni.

La presenza di questi ci permette di tralasciare i tipi classe nelle regole di inferenza e nella loro implementazione, e ci permette di non scrivere da zero un parser per il linguaggio `ABSFJf`, in quanto già presente e funzionante.

Ambedue i tool sono realizzati con `JastAdd` [11], un sistema open-source per generare compilatori ed altri tool per nuovi linguaggi di programmazione. `JustAdd` è orientato agli oggetti ed utilizza molti costrutti `Java`. Con questo tool è relativamente semplice, partendo da una grammatica, definire un albero sintattico, creare un parser e implementare diversi aspetti come type-checking o type-inference. `ABS` è infatti organizzato in un ben specifico albero sintattico, il parser popola i nodi di questo albero e il type-checker lo visita per verificare la proprietà di corretto uso dei tipi. L'algoritmo di inferenza creato non è da meno e si basa anch'esso sulla visita dell'`AST` di `ABSFJf`.

Nella Figura 3.1 vediamo un esempio semplificato di albero sintattico di `ABSFJf`; nella figura sono riportati, oltre ai nodi e le relazioni padre figlio in nero, anche archi relativi alle visite dell'albero da parte dell'algoritmo.

In colore verde abbiamo un tipo di visita bottom-up; questa corrisponde alla fase iniziale dell'algoritmo in cui viene calcolato l'ambiente partendo dalle foglie, tenendo traccia delle variabili e componendo via via l'ambiente di ogni metodo, classe ed infine l'intero programma. Oltre alle variabili (parametri usati nel body di metodo ad esempio) vengono chiaramente create ed inizializzate anche le interfacce di metodo. Tali interfacce in prima approssimazione vengono inizializzate con variabili istanziabili fresh, che al termine dell'algoritmo di inferenza saranno poi rimpiazzate da termini più specifici e correttamente calcolati.

In colore rosso, con ordine di visita top-down, troviamo i passi dell'algoritmo di inferenza vero e proprio che discuteremo a breve.

L'ambiente Γ gioca un ruolo fondamentale ed oltre ad esso, come fase preliminare, viene creata la funzione $class()$ usata nella Tabella 3.3; questa è necessaria perché in `ABSFJf` le classi non sono un tipo di dato ma le interfacce

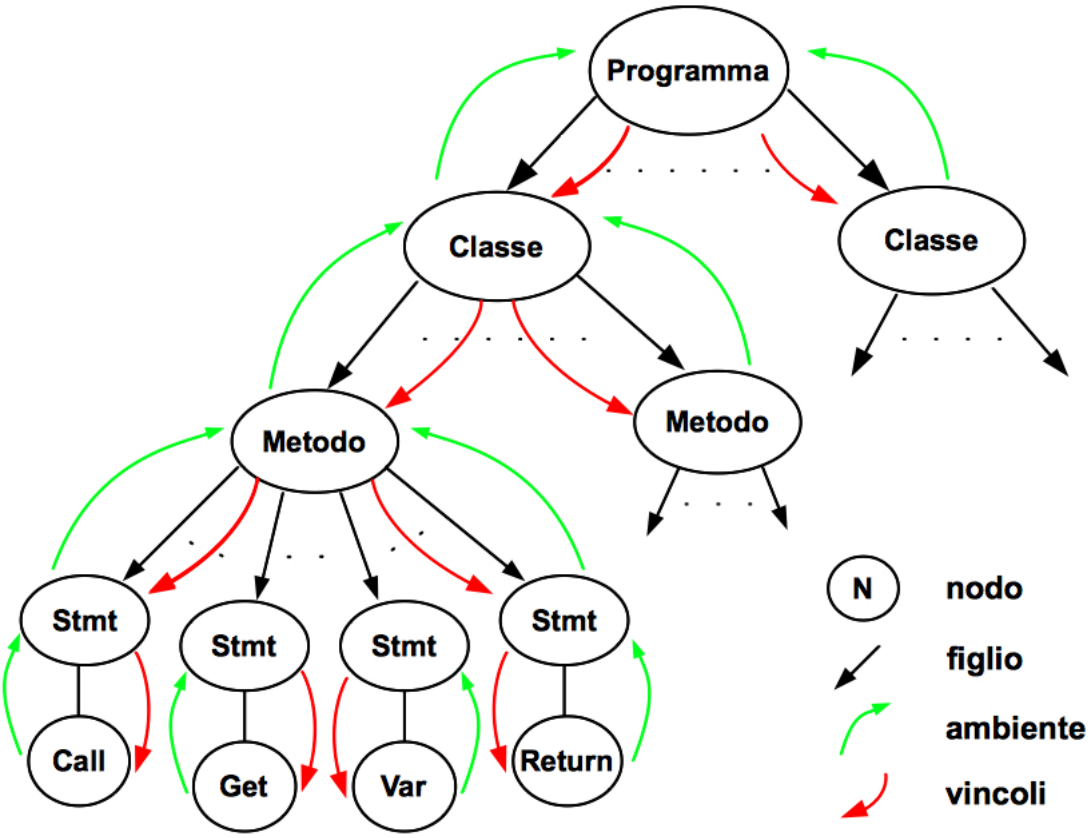

Figura 3.1: Modello semplificato di `AST` in ABSFJf

sì. L'inconveniente si risolve creando un mapping tra interfacce e classi, il ché è possibile grazie ai due vincoli:

1. Ogni classe implementa al più una interfaccia;

2. Tale interfaccia deve avere lo stesso nome della classe.

In tal modo quando si incontra una variabile di cui si vuole conoscere la classe (regole (T-Field), (T-Invk), (T-New)) basta invocare la funzione *class* sul tipo di dato (interfaccia) della variabile. La funzione che data una variabile restituisce il suo tipo è nativa del type-checker di `ABS`. Siccome esula dalla teoria ed ha una valenza puramente applicativa non compare nelle regole della Tabella 3.3 ma viene presentata solo ora per completezza. Inoltre gli unici tipi dato di `ABSFJf` sono le interfacce, quindi chiamiamo tale funzione $interface()$. Anche la funzione $fields()$ è già presente come funzione base dell'albero sintattico una volta noto il nodo della classe. La procedura che

data un'espressione `e` restituisce il tipo di interfaccia `I` e i campi $\bar{\texttt{f}}$ di classe `C` è la seguente:

$$interface(\texttt{e}) = \texttt{I}$$

$$class(\texttt{I}) = \texttt{C}$$

$$fields(\texttt{C}) = \bar{\texttt{f}}$$

Una volta calcolato l'ambiente Γ e la funzione *class*, le regole della tabella Tabella 3.3 si implementano quasi naturalmente visitando l'albero dalla radice alle foglie, seguendo appunto le frecce rosse della Figura 3.1, e risalendo, vengono collezionati i vincoli delle regole. L'implementazione è "quasi" naturale a causa degli statement di `ABSFJf`, necessari per dichiarare ed assegnare variabili fondamentali per scomporre le espressioni complesse. Se in `FJf` è possibile scrivere:

```
{return e.await.get;}
```

In `ABSFJf` ciò va scomposto in più statement in questo modo:

```
{ x := e;
  await x?;
  y = x.get;
  return y; }
```

Questo ha portato alla creazione di regole per gestire gli statement nuovi e completare l'inferenza, Tabella 3.4.

Le nuove regole hanno alcune caratteristiche differenti da quelle della Tabella 3.3. Ad esempio in queste è possibile modificare l'ambiente, infatti le variabili dichiarate e assegnate vanno a finire nell'ambiente Γ e con loro anche il record delle espressioni con cui sono inizializzate. Con questa strategia è possibile garantire che ogni statement non restituisca alcun record $\mathbb{r}$. Questo è il motivo per cui nella regola (T-Method) della Tabella 3.3 si usa la variabile `ret` e non il risultato di `return e`, proprio perché l'operazione di `return` essendo uno statement ha record nullo e viene quindi inserito il valore del record di `e` nella variabile speciale `ret` (si veda anche la regola (T-Ret)). Anche la regola (T-Await) va riscritta per correttezza, l'operazione di `await` è infatti uno statement (non una espressione come in `FJf`) e si applica ad una variabile che, già inizializzata, è contenuta nell'ambiente.

Definite anche queste regole per gestire gli statement l'algoritmo di inferenza dei contratti risulta realizzabile. Tale algoritmo, come abbiamo detto, accumula vincoli durante l'esecuzione ed alla fine si ha a disposizione un più o

$$\frac{\mathtt{Y}\ \textit{fresh}}{\Gamma + \mathtt{x} : \mathtt{Y} \vdash_a \mathtt{T}\ \mathtt{x}\ :\ \varnothing\,,\ \varnothing \rhd \varnothing}\ (\text{T-VarDecl}) \qquad \frac{\Gamma \vdash_a \mathtt{e}\ :\ \mathbb{r}\,,\ \mathbb{c} \rhd \mathcal{U}}{\Gamma \vdash_a \mathtt{x} := \mathtt{e}\ :\ \varnothing\,,\ \mathbb{c} \rhd \mathcal{U} \cup \{\Gamma(\mathtt{x}) = \mathbb{r}\}}\ (\text{T-VarAssign})$$

$$\frac{\Gamma \vdash_a \mathtt{e}\ :\ \mathbb{r}\,,\ \mathbb{c} \rhd \mathcal{U},\ \mathtt{Y}\ \textit{fresh}}{\Gamma + \mathtt{x} : \mathtt{Y} \vdash_a \mathtt{T}\ \mathtt{v} := \mathtt{e}\ :\ \varnothing\,,\ \mathbb{c} \rhd \mathcal{U} \cup \{\Gamma(\mathtt{x}) = \mathbb{r}\}}\ (\text{T-VarDeclWithAssign})$$

$$\frac{\Gamma \vdash_a \mathtt{x}\ :\ \Gamma(\mathtt{x})\,,\ \mathbb{c} \rhd \mathcal{U} \quad a', X\ \textit{fresh}}{\Gamma \vdash_a \mathtt{await}\ \mathtt{x}?\ :\ \varnothing\,,\ \mathbb{c} \between (a, a')^{\mathtt{a}} \rhd \mathcal{U} \cup \{a' \rightsquigarrow X = \Gamma\}}\ (\text{T-Await})$$

$$\frac{\Gamma_1 \vdash_a \mathtt{s}_1\ :\ \varnothing\,,\ \mathbb{c}_1 \rhd \mathcal{U}_1 \quad \Gamma_2 \vdash_a \mathtt{s}_2\ :\ \varnothing\,,\ \mathbb{c}_2 \rhd \mathcal{U}_2}{\Gamma_2 \vdash_a \mathtt{s}_1; \mathtt{s}_2\ :\ \varnothing\,,\ \mathbb{c}_1; \mathbb{c}_2 \rhd \mathcal{U}_1 \cup \mathcal{U}_2}\ (\text{T-StmtSeq})$$

Tabella 3.4: Regole degli statement per l'inferenza dei contratti in ABSFJf

meno cospicuo insieme di vincoli da soddisfare. La semiunificazione descritta in precedenza risolve esattamente questo problema producendo in output una sostituzione come soluzione. Applicando tale sostituzione alla CCT "grezza" costruita in precedenza tramite il calcolo dell'ambiente Γ si ottiene il vero output dell'algoritmo di inferenza: la *Contract Class Table* che associa ad ogni nome di metodo il rispettivo contratto di metodo.

3.7 Esempi

In questa sezione vediamo alcuni esempi di output prodotti dall'algoritmo di inferenza su alcuni semplici programmi `ABSFJf`.

3.7.1 Esempio di creazione di oggetto

In questo primo esempio vediamo un programma che implementa una semplice classe `C` con un parametro (campo) `f` al suo interno. La classe implementa un metodo `m` che restituisce un nuovo oggetto della stessa classe con il medesimo campo `f` al suo interno. Il codice di tale programma viene qui riportato:

```
interface Object {
}

class Object implements Object {
}

interface C {
    C m();
}

class C(Object f) implements C {
    C m() {
        C x = new cog C(f);
        return x;
    }
}

{

}
```

Si noti che, in questo primo esempio, il `main`, l'ultimo blocco tra parentesi graffe, è vuoto. La compilazione[2] di questo programma produce il seguente output, dapprima viene stampato l'ambiente iniziale, calcolato con la prima visita dell'`AST`:

[2]L'algoritmo di inferenza dei contratti, così come il type-checker, viene eseguito a tempo di compilazione

```
Method_C.m :== 'a
```

Il metodo `C.m` infatti è l'unico metodo dell'intero programma e quindi l'unico inizializzato nell'ambiente. Al termine della compilazione, una volta risolti i vincoli ed applicata la sostituzione, la *Contract Class Table* calcolata è:

```
############################################################
Method:  Main.main
    Contract:  0
Method:  C.m
    Contract:  'f[ Field_f('g) ]( ) -> 'h[ Field_f('g) ] {0}
############################################################
```

Come ci si aspetta il metodo `C.m` ha un contratto di metodo così composto:

- Il contratto del *body* di metodo è nullo, infatti la creazione e restituzione di un oggetto non produce dipendenze tra nomi di oggetto.
- L'interfaccia di metodo è strutturata in modo che il `this` ed il valore di ritorno abbiano nomi di oggetto diverso ma lo stesso record come campo.
- Non vengono passati parametri al metodo, l'interfaccia infatti non prende argomenti.

Siccome questo è il primo esempio di output di CCT è opportuno specificare che troveremo sempre un metodo di nome `Main.main` che presenta il contratto, come si può intuire, del `main` del programma. A differenza degli altri metodi che posseggono un contratto di metodo, il main possiede un normale contratto, infatti il `main` non è un metodo vero e proprio ma viene inserito per praticità all'interno della CCT.

3.7.2 Esempio di restituzione di un parametro

In questo esempio vediamo un programma simile al precedente per quanto riguarda la semplicità. Questa volta il metodo `m` restituisce il campo `f`, cambia quindi anche il tipo di ritorno del metodo che in questo caso è `Object`. Il codice relativo a questo programma è il seguente:

```
interface Object {
}

class Object implements Object {
}

interface C {
    Object m();
}

class C(Object f) implements C {
    Object m() {
        return f;
    }
}

{

}
```

Al termine della compilazione la CCT calcolata risulta:

```
############################################################
Method:  Main.main
    Contract:  0
Method:  C.m
    Contract:  'e[ Field_f('d) ]( ) -> 'd ] {0}
############################################################
```

Anche in questo caso la struttura del contratto di metodo di `C.m` risulta essere quella che ci si aspetta: il valore di ritorno dell'interfaccia è proprio il campo contenuto all'interno del `this`.

3.7.3 Esempio di dipendenza di tipo get

In questo esempio vediamo un primo caso di dipendenza tra nomi di oggetto, per semplificare le cose inizialmente utilizziamo l'operazione di `get` su un parametro e non sull'invocazione di metodo che vedremo nell'esempio successivo. Il metodo `C.m` prende in ingresso un parametro di tipo futuro e restituisce il risultato dell'esecuzione della `get` su tale parametro, il codice è il seguente:

```
interface Object {
}

class Object implements Object {
}

interface C {
    C m();
}

class C(Object f) implements C {
    C m(Fut<C> a) {
        return a.get;
    }
}

{

}
```

Al termine della compilazione la CCT calcolata risulta:

```
############################################################
Method:  Main.main
    Contract:  0
Method:  C.m
    Contract:  'c[ Field_f('b) ]( 'f ⇝ 'e ) -> 'e {('c,'f)}
############################################################
```

In questa *Contract Class Table* vediamo come abbiamo detto un primo esempio di dipendenza tra nomi di oggetto, del metodo di contratto di `C.m` vanno notate le seguenti caratteristiche:

- L'interfaccia di metodo in questo caso richiede un parametro in ingresso;
- Il record del parametro in ingresso è effettivamente un tipo di record futuro ‘`f` ⇝ ‘`e`;
- In accordo con il record del parametro, il record restituito dal metodo è ‘`e`, risultato dell'operazione di `get` su ‘`f` ⇝ ‘`e`;
- Il body di metodo ha un contratto non nullo, in particolare il contratto è la coppia (‘`c`, ‘`f`) infatti il rilascio dell'oggetto di nome ‘`c`, che è appunto

il *root*(`this`) del metodo, richiede il rilascio dell'oggetto di nome '`f` e quindi che l'operazione di `get` produca un risultato.

3.7.4 Esempio di chiamata di metodo

In questo esempio troviamo, oltre a molte cose viste negli esempi precedenti, anche la chiamata di metodo. Il metodo `C.m` come in precedenza rimane semplice e in questo programma restituisce banalmente il `this`. Il metodo `C.n` crea un nuovo oggetto della classe `C` con lo stesso campo `f` del `this`, su questo oggetto esegue la chiamata al metodo `C.m`, sulla chiamata effettua un'operazione di `await` e restituisce il risultato della chiamata con una `get`. Il codice che implementa la descrizione data è il seguente:

```
interface Object {
}

class Object implements Object {
}

interface C {
    C m();
    C n();
}

class C(Object f) implements C {
    C m() {
        return this;
    }
    C n() {
        C x = new cog C(f);
        Fut<C> y = x!m();
        await y?;
        return y.get;
    }
}

{

}
```

Al termine della compilazione la CCT calcolata risulta:

```
#################################################################
Method:  Main.main
    Contract:  0
Method:  C.m
    Contract:  'd[ Field_f('c) ]( ) -> 'd[ Field_f('c) ] {0}
Method:  C.n
    Contract:  'j[ Field_f('k) ]( ) -> 'l[ Field_f('k) ] ...
      ... {ClassC.m('l[ Field_f('k) ]( ) -> 'l[ Field_f('k) ]).('j,'l)@}
#################################################################
```

In questa *Contract Class Table* nel metodo `C.n` viene separata l'interfaccia dal contratto tramite punti "..." a causa delle dimensioni del contratto di metodo, mentre il metodo `C.m` è perfettamente comprensibile e restituisce lo stesso oggetto del `this`. Il contratto di metodo di `C.n` merita di essere analizzato:

- All'interno del contratto del body troviamo l'invocazione ad un metodo, si tratta appunto del metodo `C.m`;
- Il metodo `C.m` viene invocato su un nuovo oggetto, con nome nuovo, ma con lo stesso campo del parametro `this` del metodo chiamante `C.n`;
- Il record di ritorno della chiamata al metodo `C.m` è lo stesso del record di ritorno dell'interfaccia del metodo `C.n`;
- Il record di ritorno della chiamata al metodo `C.m` è giustamente lo stesso record su cui si effettua la chiamata, infatti, a livello strutturale, il metodo `C.m` restituisce l'oggetto su cui viene invocato;
- Nel contratto del body a seguito dell'invocazione abbiamo un contratto di tipo `await` (si distingue dal contratto di tipo `get` già visto in precedenza per la presenza della chiocciola @) che ricorda che per restituire il controllo all'oggetto di nome '`j`, il `this` di `C.n`, occorre prima che venga risolta la chiamata sull'oggetto di nome '`l`, il `this` del metodo `C.m` invocato. Siccome questa dipendenza è di tipo `await` il lock su '`j` nel frattempo può essere rilasciato.
- L'operazione di `get` che segue quella di `await` non aggiunge una dipendenza di tipo ('`j`, '`l`) al contratto come specificato nelle regole dell'operatore $\wr$, dimostrando che l'accumulazione dei contratti viene eseguita correttamente.

Capitolo 4

Deadlock analysis

In questo capitolo discutiamo il modello e il corrispondente algoritmo di analisi statica dei deadlock.

Chiamiamo O l'insieme dei nomi di oggetto ed $\mathsf{O}^{\mathtt{a}}$ l'insieme dei nomi di oggetto marcati con l'esponente $\mathtt{a}$; il primo insieme si riferisce quindi a dipendenze di tipo `get`, il secondo, a dipendenze di tipo `await`. Sia anche $\mathcal{O}^{[\mathtt{a}]} = (\mathsf{O} \times \mathsf{O}) \cup (\mathsf{O}^{\mathtt{a}} \times \mathsf{O}^{\mathtt{a}})$ l'insieme di tutte le coppie di dipendenza.

Definizione 4.1. Un *modello per analisi dei deadlock con coppie*, in breve *lamp*[1], è un insieme di *stati*, dove ogni stato è un sottoinsieme di $\mathcal{P}(\mathcal{O}^{[\mathtt{a}]})$.

I *lamp* vengono indicati con $\mathcal{W}, \mathcal{W}', \cdots$ sono illustrati come "ovali" rappresentanti stati in cui ogni stato contiene coppie di nomi di oggetto, coppie di dipendenze quindi. Gli insiemi di coppie vengono indicati con $W, W', \cdots$.

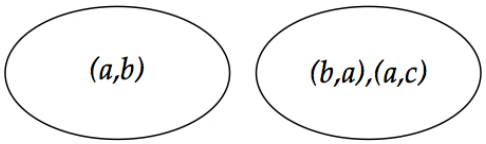

Figura 4.1: Un semplice *lamp* con due stati.

Per esempio, in Figura 4.1 viene illustrato un *lamp* con due stati. I nomi di oggetto delle coppie negli stati sono rispettivamente $\{(a, b)\}$ e $\{(b, a), (a, c)\}$

Diciamo $a \in \mathcal{W}$ se a oppure $a^{\mathtt{a}}$ è contenuto in $\mathcal{W}$. Chiamiamo $fv(\mathcal{W}) = \{a \mid a \in \mathcal{W}\}$ l'insieme delle variabili libere, quindi dei nomi di oggetto, all'interno del *lamp*. Sia inoltre $\mathcal{W}[^{b}/_{a}]$ il *lamp* ottenuto dalla sostituzione delle occorrenze di a e $a^{\mathtt{a}}$ con b e $b^{\mathtt{a}}$, rispettivamente.

I *lamp* hanno una relazione d'ordine $\preceq_{\widetilde{a}}$, con $\widetilde{a} \in \mathsf{O}$, definito come segue:

[1] *lamp* detiva dall'equivalente inglese di Lock Analysis Model with Pairs

$\mathcal{W} \preceq_{\widetilde{a}} \mathcal{W}'$ se esiste una funzione iniettiva f che va da $\widetilde{b} = fv(\mathcal{W}) \setminus \widetilde{a}$ a $fv(\mathcal{W}') \setminus \widetilde{a}$ tale che per ogni $W \in \mathcal{W}$, esista $W' \in \mathcal{W}'$ con $W[^{f(\widetilde{b})}/_{\widetilde{b}}] \subseteq W'$.

Dove $\widetilde{a}$ sono i parametri formali del *lamp* che verranno definiti a breve, mentre $\widetilde{b}$ sono quindi tutti i nomi liberi $fresh$ all'interno del *lamp*.

I *lamp* vengono assunti come insieme non vuoti, quindi, il più piccolo, in accordo alla definizione di $\preceq_{\widetilde{a}}$ è $\{\varnothing\}$, che viene chiamato `0`.

Definiamo alcune operazioni tra *lamp* che serviranno nell'analisi:

sequenza: $\curvearrowright$ definita come segue:

- $\mathtt{0} \curvearrowright \mathcal{W} \stackrel{def}{=} \mathcal{W}$;
- $\mathcal{W} \curvearrowright \mathtt{0} \stackrel{def}{=} \mathcal{W}$;
- (altrimenti) $\mathcal{W} \curvearrowright \mathcal{W}' \stackrel{def}{=} \mathcal{W} \cup \mathcal{W}'$;

parallelo: $\mathcal{W} \| \mathcal{W}' \stackrel{def}{=} \{W \cup W' \mid W \in \mathcal{W} \text{ and } W' \in \mathcal{W}'\}$.

Si noti come l'operatore `parallelo` sia a tutti gli effetti il prodotto cartesiano tra gli stati contenuti nei due *lamp*. Si noti anche come i *lamp* ottenuti da `0` applicando le regole sopra sono `0` oppure contengono alla peggio stati non vuoti. Tale proprietà segue dalla definizione di `sequenza` che non rimuove mai alcun elemento.

Una operazione `op` è monotona rispetto a $\preceq_{\widetilde{a}}$, se, ogni volta che $\mathcal{W}_1 \preceq_{\widetilde{a}} \mathcal{W}_2$ si ha $\mathtt{op}(\mathcal{W}_1) \preceq_{\widetilde{a}} \mathtt{op}(\mathcal{W}_2)$. Alcune proprietà dimostrabili, fondamentali ai fini dell'analisi dei deadlock, sono le seguenti:

- Le operazioni di `sequenza` e `parallelo` sono monotone rispetto a $\preceq_{\widetilde{a}}$.
- Sia in ipotesi $\mathcal{W} \preceq_{\widetilde{a}} \mathcal{W}'$ e sia f una funzione su $\widetilde{b} = fv(\mathcal{W}) \setminus \widetilde{a}$ tale che $f(\widetilde{b}) \cap \widetilde{a} = \varnothing$. Allora, $\mathcal{W}[^{f(\widetilde{b})}/_{\widetilde{b}}] \preceq_{\widetilde{a}} \mathcal{W}'$. [Quindi $\mathcal{W} \simeq_{\widetilde{a}} \mathcal{W}[^{f(\widetilde{b})}/_{\widetilde{b}}]$.]
- Sia in ipotesi $\mathcal{W} \preceq_{\widetilde{a}} \mathcal{W}'$ e sia f una funzione su $\widetilde{a}$ tale che $f(\widetilde{a}) \cap (fv(\mathcal{W}) \setminus \widetilde{a}) = \varnothing$ e $f(\widetilde{a}) \cap (fv(\mathcal{W}') \setminus \widetilde{a}) = \varnothing$. Allora $\mathcal{W}[^{f(\widetilde{a})}/_{\widetilde{a}}] \preceq_{\widetilde{a}} \mathcal{W}'[^{f(\widetilde{a})}/_{\widetilde{a}}]$.

Dato $\preceq_{\widetilde{a}}$ sui *lamp*, è facile definire un ordinamento parziale su un prodotto cartesiano di *lamp* imponendo un ordinamento definito in questo modo:

$$\left(\mathcal{W}_1, \cdots, \mathcal{W}_k\right) \preceq_{\widetilde{a_1}, \cdots, \widetilde{a_k}} \left(\mathcal{W}'_1, \cdots, \mathcal{W}'_k\right) \quad \stackrel{def}{=} \quad \text{per ogni } i : \mathcal{W}_i \preceq_{\widetilde{a_i}} \mathcal{W}'_i$$

La seguente semantica astratta associa a *coppie di lamp* $\langle \mathcal{W}, \mathcal{W}' \rangle$ ad espressioni e metodi. Per motivare la necessità di coppie si consideri il contratto $\mathbb{c} = \mathtt{C.m}\, b[\,]()\mathtt{.}(a, b)$. Tale contratto aggiunge una dipendenza tra nomi

(a, b) allo stato corrente. Se il metodo `m` della classe `C` segue una chiamata di metodo, sia questa `D.n` $b[\,](\,)$ (senza alcun tipo di `get` o `await`), allora l'invocazione `C.m` $b[\,](\,)$ non contribuisce allo stato corrente con nuove coppie. Però è possibile che `D.n` $b[\,](\,)$ introduca coppie di dipendenze che influenzano *stati futuri* che non hanno nulla a che vedere con (a, b). La stessa situazione si ha anche quando `D.n` è un automa: le coppie di dipendenze future sono aggiunte in accordo alle regole dell'automa. Quindi, per aumentare la precisione della semantica astratta (composizionale), teniamo separati gli insiemi di coppie sopra nella costruzione del modello utilizzando coppie di *lamp*. Una volta che la costruzione termina, le coppie$\langle \mathcal{W}, \mathcal{W}' \rangle$ restitute dall'algoritmo per programmi `ABSFJf`, vanno interpretate come il (singolo) *lamp* $\mathcal{W} \curvearrowright \mathcal{W}'$. Quindi, i futuri sono semplicemente gli stati dopo lo stato finale del primo *lamp* della coppia.

Vengono usate le seguenti operazioni su coppie di *lamp*:

Aggiunta di dipendenza: $\langle \mathcal{W}, \mathcal{W}' \rangle \oplus (a, b) \stackrel{def}{=} \langle \mathcal{W} \oplus (a, b), \mathcal{W}' \rangle$; (analogo per $\langle \mathcal{W}, \mathcal{W}' \rangle \oplus (a^{\mathtt{a}}, b^{\mathtt{a}})$);

Sequenza tra coppie di lamp:

$$\langle \mathcal{W}_1, \mathcal{W}_1' \rangle \fatsemi \langle \mathcal{W}_2, \mathcal{W}_2' \rangle \stackrel{def}{=} \begin{cases} \langle \mathcal{W}_1, \mathcal{W}_1' \| \mathcal{W}_2' \rangle & \text{if } \mathcal{W}_2 = \mathtt{0} \\ \langle \mathcal{W}_1 \curvearrowright (\mathcal{W}_2 \| \mathcal{W}_1') \ , \ \mathcal{W}_1' \| \mathcal{W}_2' \rangle & \text{altrimenti} \end{cases}$$

Si noti che, grazie alla monotonia di `sequenza` e `parallelo`, l'aggiunta di una dipendenza e la sequenza tra coppie di lamp sono a loro volta monotone (su coppie di *lamp*).

4.1 Punto fisso con saturazione

Il modello astratto di un programma `ABSFJf` è ottenuto tramite l'utilizzo di un algoritmo basato sulla tecnica standard del punto fisso. Una operazione basilare dell'algoritmo di analisi è il rimpiazzamento di nomi di oggetto all'interno degli stati di un *lamp* con altri nomi. In particolare, sia $\mathcal{W}[^b/_a]$ un *lamp* dove ogni occorrenza di a viene rimpiazzata con b. Analogamente a come è stato fatto con i contratti nel relativo capitolo, è possibile definire $\mathcal{W}[^{\mathbb{s}}/_{\mathbb{r}}]$. Sia inoltre $\langle \mathcal{W}, \mathcal{W}' \rangle[^{\mathbb{s}}/_{\mathbb{r}}] = \langle \mathcal{W}[^{\mathbb{s}}/_{\mathbb{r}}], \mathcal{W}'[^{\mathbb{s}}/_{\mathbb{r}}] \rangle$.

Definizione 4.2. Sia CCT la *Contract Class Table* di un programma `ABSFJf` e $(\cdots \langle \mathcal{W}_{\mathtt{C.m}}, \mathcal{W}'_{\mathtt{C.m}} \rangle, \cdots)$ sia una tupla di coppie di *lamp* tale che, per ogni metodo `C.m` nel programma ci sia una corrispettiva coppia $\langle \mathcal{W}_{\mathtt{C.m}}, \mathcal{W}'_{\mathtt{C.m}} \rangle$. Siano

$\widetilde{a}_{\texttt{C.m}}$ i nomi di oggetto della relazione d'ordine $\preceq_{\widetilde{a}_{\texttt{C.m}}}$ corrispondenti a `C.m` (tali nomi sono i parametri formali del metodo, ovvero il `this`, oggetto su cui viene invocato il metodo, e gli argomenti).

La *trasformazione in lamp* di CCT, denotata

$$\Big(\cdots, \mathbb{r}_{\texttt{C.m}}(\overline{\mathbb{s}_{\texttt{C.m}}}).\mathbb{c}_{\texttt{C.m}}, \cdots\Big)$$

tale che $\text{CCT}(\texttt{C})(\texttt{m}) = \mathbb{r}_{\texttt{C.m}}(\overline{\mathbb{s}_{\texttt{C.m}}})\ \{\mathbb{c}_{\texttt{C.m}}\}\ \mathbb{r}'_{\texttt{C.m}}$, è definita come segue

1. Sia $\widetilde{b} = \bigcup_{\texttt{C.m}}(\mathit{names}(\mathbb{c}_{\texttt{C.m}}, \mathbb{r}'_{\texttt{C.m}}) \setminus \mathit{names}(\mathbb{r}_{\texttt{C.m}}, \overline{\mathbb{s}_{\texttt{C.m}}})$ (stiamo assumendo che i nomi liberi di metodi differenti siano disgiunti). Questi sono i nomi di oggetto che vengono creati ad ogni *passo di trasformazione*, riferiti al singolo metodo, sono i nomi dei nuovi oggetti creati nel body. Siano $\widetilde{b'}$ nomi di oggetto *fresh* (i nomi che rimpiazzeranno $\widetilde{b}$);

2. Sia $\widetilde{a}'_{\texttt{D.n}} = (\mathit{fv}(\mathcal{W}_{\texttt{D.n}}) \cup \mathit{fv}(\mathcal{W}'_{\texttt{D.n}})) \setminus \widetilde{a}_{\texttt{D.n}}$ con $\widetilde{a}_{\texttt{D.n}} = \mathit{names}(\mathbb{r}_{\texttt{D.n}}, \overline{\mathbb{s}_{\texttt{D.n}}})$ ovvero i parametri formali del metodo (da rimpiazzare quando il metodo `D.n` viene invocato);

3. La trasformazione $\mathbb{r}_{\texttt{C.m}}(\overline{\mathbb{s}_{\texttt{C.m}}}).\mathbb{c}_{\texttt{C.m}}(\cdots\langle\mathcal{W}_{\texttt{C.m}}, \mathcal{W}'_{\texttt{C.m}}\rangle, \cdots)$ restituisce una coppia di *lamp* $\langle\mathcal{W}''_{\texttt{C.m}}, \mathcal{W}'''_{\texttt{C.m}}\rangle$ definita come segue, per induzione sul contratto di metodo di `C.m`:

   - $\Big(\langle \texttt{0}, \texttt{0}\rangle \oplus (a_1^{[\texttt{a}]}, a_2^{[\texttt{a}]})\Big)[\widetilde{b'}/\widetilde{b}]$ (W-GAZERO)
     se $\mathbb{c}_{\texttt{C.m}} = (a_1, a_2)^{[\texttt{a}]}$;

   - $\langle \texttt{0}, \mathcal{W}_{\texttt{D.n}} \curvearrowright \mathcal{W}'_{\texttt{D.n}}\rangle\Big)[\widetilde{a'}/\widetilde{a}'_{\texttt{D.n}}][\mathbb{r}'[\widetilde{b'}/\widetilde{b}]/_{\mathbb{r}_{\texttt{D.n}}}][\overline{\mathbb{s}'}[\widetilde{b'}/\widetilde{b}]/_{\overline{\mathbb{s}_{\texttt{D.n}}}}][\mathbb{r}''[\widetilde{b'}/\widetilde{b}]/_{\mathbb{r}'_{\texttt{D.n}}}]$
     dove $\widetilde{a'}$ sono i nomi di oggetto *fresh* (W-INVK)
     se $\mathbb{c}_{\texttt{C.m}} = \texttt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$ e $\text{CCT}(\texttt{D})(\texttt{n}) = \mathbb{r}_{\texttt{D.n}}(\overline{\mathbb{s}_{\texttt{D.n}}})\ \{\mathbb{c}_{\texttt{D.n}}\}\ \mathbb{r}'_{\texttt{D.n}}$

   - $\Big(\langle \mathcal{W}_{\texttt{D.n}}, \mathcal{W}'_{\texttt{D.n}}\rangle[\widetilde{a'}/\widetilde{a}'_{\texttt{D.n}}][\mathbb{r}'[\widetilde{b'}/\widetilde{b}]/_{\mathbb{r}_{\texttt{D.n}}}][\overline{\mathbb{s}'}[\widetilde{b'}/\widetilde{b}]/_{\overline{\mathbb{s}_{\texttt{D.n}}}}][\mathbb{r}''[\widetilde{b'}/\widetilde{b}]/_{\mathbb{r}'_{\texttt{D.n}}}]\Big) \oplus (a_1^{[\texttt{a}]}, a_2^{[\texttt{a}]})[\widetilde{b'}/\widetilde{b}]$
     dove $\widetilde{a'}$ sono i nomi di oggetto *fresh* (W-GAINVK)
     if $\mathbb{c}_{\texttt{C.m}} = \texttt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''.(a_1, a_2)^{[\texttt{a}]}$ and $\text{CCT}(\texttt{D})(\texttt{n}) = \mathbb{r}_{\texttt{D.n}}(\overline{\mathbb{s}_{\texttt{D.n}}})\ \{\mathbb{c}_{\texttt{D.n}}\}\ \mathbb{r}'_{\texttt{D.n}}$

   - $\mathbb{r}_{\texttt{C.m}}(\overline{\mathbb{s}_{\texttt{C.m}}}).\mathbb{c}'_{\texttt{C.m}}(\cdots\langle\mathcal{W}_{\texttt{C.m}}, \mathcal{W}'_{\texttt{C.m}}\rangle, \cdots) \fatsemi \mathbb{r}_{\texttt{C.m}}(\overline{\mathbb{s}_{\texttt{C.m}}}).\mathbb{c}''_{\texttt{C.m}}(\cdots\langle\mathcal{W}_{\texttt{C.m}}, \mathcal{W}'_{\texttt{C.m}}\rangle, \cdots)$ (W-SEQ)
     se $\mathbb{c}_{\texttt{C.m}} = \mathbb{c}'_{\texttt{C.m}} \fatsemi \mathbb{c}''_{\texttt{C.m}}$.

Si noti che, al punto 3., non assumiamo che i nomi liberi in `C.m` e quelli in `D.n` della CCT non si sovrappongano. Comunque, una CCT con insiemi disgiunti di nomi liberi per ogni contratto di metodo raggiunge un risultato

maggiormente preciso con la tecnica descritta sopra. Nel punto 1. viene introdotto $\widetilde{b}$, un insieme di nomi fresh di oggetto che rimpiazzeranno i nomi liberi nel contratto di metodo. Questo insieme è creato una volta, all'inizio di ogni passo di trasformazione. Ci sono anche altri nomi fresh che devono essere creati ad ogni step di trasformazione, tutti i nomi che non sono i cosiddetti parametri formali, quindi i nomi che rientrano in $\widetilde{a}$ per ogni metodo.

Discutiamo la regola (W-Invk), le altre sono simili a questa. In questo caso, il contratto $\mathbb{c}_{\mathtt{C.m}}$ è $\mathtt{D.n}\ \mathbb{r}'(\overline{\mathbb{s}'}) \to \mathbb{r}''$. Siccome questa è una invocazione asincrona, non ha effetti sullo stato corrente, bensì li ha su quelli futuri. Quindi il modello di questo contratto $\mathbb{c}_{\mathtt{C.m}}$ è $\langle \mathtt{0}, \mathcal{W} \rangle$, per qualche $\mathcal{W}$.

Si può inoltre notare che la trasformazione di *lamp* della Definizione 4.2 è definita dalla composizione di operazioni monotone. Quindi, partendo da uno stato iniziale con la tupla $\left(\cdots \langle {\mathcal{W}_{\mathtt{C,m}}}^0, {\mathcal{W}'_{\mathtt{C,m}}}^0 \rangle, \cdots\right) = \left(\cdots \langle \mathtt{0}, \mathtt{0} \rangle, \cdots\right)$, otteniamo una sequenza non decrescente (rispetto a $\preceq$)

$$\left(\cdots \langle {\mathcal{W}_{\mathtt{C,m}}}^0, {\mathcal{W}'_{\mathtt{C,m}}}^0 \rangle, \cdots\right), \left(\cdots \langle {\mathcal{W}_{\mathtt{C,m}}}^1, {\mathcal{W}'_{\mathtt{C,m}}}^1 \rangle, \cdots\right), \left(\cdots \langle {\mathcal{W}_{\mathtt{C,m}}}^2, {\mathcal{W}'_{\mathtt{C,m}}}^2 \rangle, \cdots\right), \cdots$$

seguendo proprio la tecnica standard di Knaster-Tarski. La coppia $\langle \mathcal{W}^i_{\mathtt{C,m}}, {\mathcal{W}'_{\mathtt{C,m}}}^i \rangle$ è *l'i-esimo approssimante finito* del modello di `C.m`. Nel dominio dei *lamp*, a causa delle creazioni di nomi di oggetto nuovi (fresh), il punto fisso della sequenza sopra potrebbe non esistere. Per esempio, il contratto $\mathtt{C.m}\ b[\,](\,) \to b[\,].(a, b)$, dove $\textsc{cct}(\mathtt{C})(\mathtt{m}) = a[\,](\,)\ \{\{\mathtt{C.m}\ b[\,](\,) \to b[\,].(a, b)\}\}\ b[\,]$ produce una sequenza infinita

$$\Big(\langle \mathtt{0}, \mathtt{0} \rangle\Big), \Big(\langle \{(b_0, b_1)\}, \mathtt{0} \rangle\Big), \Big(\langle \{(b_0, b_1), (b_1, b_2)\}, \mathtt{0} \rangle\Big), \Big(\langle \{(b_0, b_1), (b_1, b_2), (b_2, b_3)\}, \mathtt{0} \rangle\Big), \cdots$$

dove l'insieme di coppie W rappresenta un *lamp a singolo stato senza transizioni* $(\{\mathtt{w}\}, \varnothing, \mathtt{w}, \mathtt{w})$. La sequenza sopra non ha un upper bound nel dominio dei *lamp* (che sono tutti finiti). Quindi, per raggiungere un risultato, si lancia la tecnica di Knaster-Tarski su un *insieme finito di nomi di oggetto*. Se l'n-esimo approssimante finito non ha raggiunto un punto fisso ed ha consumato i nomi di oggetto possibili, allora l'$(n+1)$-esimo approssimante riutilizzerà gli stessi nomi di oggetto utilizzati all'iterazione precedente, e via via anche l'$(n+2)$-esimo approssimante farà lo stesso fino a ché un punto fisso non viene raggiunto. Questa procedura prende il nome di *tecnica di saturazione ad n*. Per esempio, nel caso della sequenza sopra, se l'insieme dei nomi di oggetto utilizzabili è $\{b_0, b_1\}$ allora la tecnica di saturazione termina in due passi raggiungendo la coppia di *lamp* $\langle \{(b_0, b_1), (b_1, b_1)\}, \mathtt{0} \rangle$.

Definizione 4.3. Sia $(\textsc{ct}, \mathtt{s}, \textsc{cct})$ un programma `ABSFJf` e sia $\left(\cdots \langle {\mathcal{W}_{\mathtt{C.m}}}^{n+h}, {\mathcal{W}'_{\mathtt{C.m}}}^{n+h} \rangle, \cdots\right)$ il punto fisso ottenuto tramite la tecnica di saturazione ad n. La *Tabella delle*

Classi Astratta ad n, scritta $\text{ACT}_{[n]}$, è una funzione che prende il metodo `C.m` e restituisce $\langle {\mathcal{W}_{\texttt{C,m}}}^{n+h}, {\mathcal{W}'_{\texttt{C,m}}}^{n+h} \rangle$.

Sia $(\text{CT}, \texttt{s}, \text{CCT})$ un programma `ABSFJf`, $\Bbbc$ è il contratto tale che $\Gamma + \texttt{this} : (\texttt{Object}, a[\,]) \vdash_a \texttt{s} : (\texttt{T}, \mathbb{r})$, $\Bbbc$ dove Γ prende `C.m` e restituisce l'interfaccia di $\text{CCT}(\texttt{C})(\texttt{m})$, e sia $\text{ACT}_{[n]}$ la corrispondente tabella di classe astratta ad n. La *semantica astratta saturata ad* n di `s` (il `main`) è

$$a[\,]()^{\Bbbc}\Big(\cdots, \text{ACT}_{[n]}(\texttt{C.m}), \cdots\Big)$$

che, da definizione, è una coppia di *lamp*. Quindi, una volta calcolata l'$\text{ACT}_{[n]}$ per tutti i metodi del programma, la si utilizza per calcolare la coppia di *lamp* associata al main `s` del programma `ABSFJf`.

4.2 Analisi dei Deadlock e Livelock

In questo capitolo viene data una definizione formale dei concetti di deadlock e livelock, genericamente chiamati *locks*.

Definizione 4.4. Sia S uno stato contenete i task $\mathtt{t}_i :^{\ell_i}_{a_i} \mathtt{e}_i$, con $1 \leq i \leq n$ ($n \geq 1$). Allora S è *in lock* (e.g., bloccato in deadlock o livelock) se, dato $i + 1 = 1$ quando $i = n$:

1. $a_i = a_{i+1}$ implica che $\mathtt{e}_i$ non è un valore, $\ell_{i+1} = \top$ e $\mathtt{e}_{i+1} = \mathsf{E}_{i+1}[\mathtt{t}_{i+2}.\mathtt{get}]$, per qualche E_{i+1}($\mathtt{get}$ su se stessi bloccante);

2. $a_i \neq a_{i+1}$ implica che (i) $\mathtt{e}_i = \mathsf{E}_i[\mathtt{t}_{i+1}.\mathtt{get}]$ e $\ell_i = \top$ oppure (ii) $\mathtt{e}_i = \mathsf{E}_i[\mathtt{t}_{i+1}.\mathtt{await}]$, per qualche E_i (cicli o di $\mathtt{get}$ o $\mathtt{await}$ tra n oggetti);

3. esiste i tale che $\mathtt{e}_i = \mathsf{E}_i[\mathtt{t}_{i+1}.\mathtt{get}]$, per qualche E_i.

Uno stato S è *lock-free* se non è bloccato in un lock e, per ogni $\mathsf{S} \xrightarrow{a} \mathsf{S}'$, S' risulta *lock-free*. Un programma $(\mathrm{CT}, \mathtt{s}, \mathrm{CCT})$ è *lock-free* se la sua configurazione iniziale è *lock-free*.

Queste definizioni identificano stati bloccati cercando catene circolari di dipendenze tra task che non possono procedere. Per esempio

- Lo stato $\mathtt{t}_1 :^{\bot}_{a_1} \mathtt{e}_1, \mathtt{t}_2 :^{\top}_{a_1} \mathtt{t}_2.\mathtt{get}$ è in lock, in particolare in deadlock, a causa del punto 1;

- Lo stato $\mathtt{t}_1 :^{\top}_{a_1} \mathtt{t}_2.\mathtt{get}, \mathtt{t}_2 :^{\top}_{a_2} \mathtt{t}_3.\mathtt{await}$ è in lock, in particolare in livelock, a causa del punto 2;

- Lo stato $\mathtt{t}_1 :^{\top}_{a_1} \mathtt{t}_2.\mathtt{get}, \mathtt{t}_2 :^{\bot}_{a_2} \mathtt{e}_2, \mathtt{t}_3 :^{\top}_{a_2} \mathtt{t}_4.\mathtt{get}, \mathtt{t}_4 :^{\bot}_{a_4} \mathtt{e}_4, \mathtt{t}_5 :^{\top}_{a_4} \mathtt{t}_1.\mathtt{get}$ è in lock a causa dei punti 1 e 2;

(tutti e tre gli esempi soddisfano il punto 3).

Altri semplici esempi di deadlock visibili in `ABSFJf` sono $\mathtt{t} :^{\top}_{a} \mathtt{t}.\mathtt{get}$ e $\mathtt{t}_1 :^{\top}_{a_1} \mathtt{t}_2.\mathtt{get}, \mathtt{t}_2 :^{\top}_{a_2} \mathtt{t}_1.\mathtt{get}$.

In `ABSFJf` è possibile però raggiungere stati che, pur essendo dei lock, non vengono identificati dalla Definizione 4.4. Per esempio uno stato di questo tipo, $\mathtt{t}_1 :^{\bot}_{a_1} \mathtt{t}_2.\mathtt{await}, \mathtt{t}_2 :^{\top}_{a_2} \mathtt{t}_1.\mathtt{await}$, che presenta un livelock non risponde alla definizione a causa del punto 3. Si necessita quindi di una definizione più debole rimuovendo appunto la voce 3. dalla Definizione 4.4. Questo rilassamento delle ipotesi però introduce un problema; se da una parte permette di individuare correttamente come stati pericolosi (in lock)

quelli appena descritti, dall'altra segnala come pericolosi anche stati definiti "salvi"[2] come il seguente:

$$\mathtt{t}_1 :^{\bot}_{a_1} \mathtt{t}_2.\mathtt{await}, \mathtt{t}_2 :^{\top}_{a_1} \mathtt{e}$$

dove $\mathtt{e}$ non contiene $\mathtt{t}_1$. Quindi la nostra analisi potrà produrre falsi positivi (può segnalare stati in lock quando non lo sono come nell'esempio, oltre che in casi di saturazione). Come "correttezza" intendiamo che i programmi *non lock-free* vengono sempre identificati correttamente (non vi sono falsi negativi). Introduciamo così la nozione di *circolarità tra oggetti.*

Definizione 4.5. Uno stato S ha

(i) Una *dipendenza tra nomi di oggetto* (a, b) se contiene il task $\mathtt{t} :^{\top}_{a} \mathsf{E}[\mathtt{t}'.\mathtt{get}], \mathtt{t}' :_b \mathtt{e}$ ed $\mathtt{e}$ non è un valore;

(ii) Una *dipendenza tra nomi di oggetto* $(a^{\mathtt{a}}, b^{\mathtt{a}})$ se contiene il tasak $\mathtt{t} :_a \mathsf{E}[\mathtt{t}'.\mathtt{await}], \mathtt{t}' :_b \mathtt{e}$ ed $\mathtt{e}$ non è un valore.

Dato un insieme A di dipendenze di nomi di oggetto, sia $\mathtt{get}$-*closure* di A (la chiusura dell'insieme A risp. a dipendenze di tipo $\mathtt{get}$), chiamata $A^{\mathtt{get}}$, il più piccolo insieme tale che:

1. $A \subseteq A^{\mathtt{get}}$;
2. Se $(a, b) \in A^{\mathtt{get}}$ e $(b^{[\mathtt{a}]}, c^{[\mathtt{a}]}) \in A^{\mathtt{get}}$ allora $(a, c) \in A^{\mathtt{get}}$, dove $(b^{[\mathtt{a}]}, c^{[\mathtt{a}]})$ denota o la coppia (b, c) oppure la coppia $(b^{\mathtt{a}}, c^{\mathtt{a}})$.

Uno stato contiene una *circolarità di oggetti* se la $\mathtt{get}$-closure dell'insieme di dipendenze di nomi contiene la coppia (a, a).

Da questa definizione vengono considerate come circolarità pericolose quindi tutte quelle che contengono almeno una dipendenza di tipo $\mathtt{get}$, in altre parole vengono escluse solo quelle formate da sole operazioni di $\mathtt{await}$, che verrano comunque segnalate dall'algoritmo come vedremo.

Si noti che, mentre nel caso (i) della definizione di dipendenza tra nomi di oggetto, il lock di $\mathtt{t}$ è $\top$ a causa della corrispondente espressione $\mathtt{get}$, nel punto (ii) il lock di $\mathtt{t}$ può essere sia $\top$ che $\bot$ proprio per la corrispondente espressione $\mathtt{await}$ (dalla semantica di $\mathtt{ABSFJf}$). È importante oltretutto notare che la nozione di $\mathtt{get}$-closure e di circolarità di nomi andrà applicata poi ad ogni stato di ogni *lamp.*

[2]Si definiscono "salvi" gli stati non pericolosi che non contengono *lock*

4.3 Analizzatore e strutture dati

Nei capitoli precedenti è stato presentato un modello che utilizza la tecnica di Knaster-Tarski con saturazione per raggiungere un punto fisso, quindi, per produrre le dipendenze tra nomi di oggetto partendo dalla CCT di programmi `ABSFJf`. È stato inoltre mostrato come utilizzare le dipendenze calcolate per analizzare la presenza di Deadlock o Livelock e quindi la proprietà di *lock-free* di un programma `ABSFJf`. L'insieme di queste nozioni è stata implementata all'interno del compilatore di `ABS` ed il tool prende il nome di *Analizzatore Statico dei Deadlok*. All'interno dell'*analizzatore* sono quindi state inserite le regole mostrate nella Definizione 4.2 per produrre la soluzione astratta della tabella delle classi $\text{ACT}_{[n]}$ saturata ad n, oltre che un meccanismo per ricercare le circolarità presentate nella Definizione 4.5 per ogni stato di ogni *lamp*.

4.3.1 Stato e ciclo

Per iniziare una descrizione dell'algoritmo realizzato è opportuno presentare la struttura dati principale, lo stato, che raccoglie le informazioni sui possibili lock. Come abbiamo visto, all'interno di uno stato di un *lamp* abbiamo le informazioni di dipendenza tra nomi di oggetto sotto forma di coppie quali $(a_1^{[\mathtt{a}]}, a_2^{[\mathtt{a}]})$. Vediamo quindi uno stato di esempio, come $\{(a,b),(b,c),(b,d),(c^{\mathtt{a}},a^{\mathtt{a}})\}$ in Figura 4.2.

La struttura dati è così organizzata: ogni lista ha una *testa* (casella bianca a sinistra della freccia) che corrisponde ad un elemento che si trova ad occupare la prima posizione in almeno una coppia di dipendenze, mentre ogni elemento chiamato *tripla* (con tre caselle di diverso colore) è un nome che si trova nella posizione di destra in una coppia di dipendenze che ha come nome di sinistra proprio la *testa*. La *testa* conserva solo il nome, che possiamo considerare come stringa (anche se in realtà è un oggetto più complicato, una variabile, per facilitare l'operazione di sostituzione), ed è indifferente se tale nome si trova in una dipendenza di tipo `get` o `await`. Una *terna* invece possiede, come suggerisce il nome, tre informazioni:

1. La prima, riportata in bianco, è il nome dell'oggetto (ad es. b), che accoppiato con la testa (ad es. a) della lista a cui appartiene la *terna* riproduce la dipendenza $(a^{[\mathtt{a}]}, b^{[\mathtt{a}]})$ che si vuole memorizzare;

2. La seconda, in azzurro, è un banale flag che specifica se la coppia è di tipo `get` (a,b) o di tipo `await` $(a^{\mathtt{a}}, b^{\mathtt{a}})$;

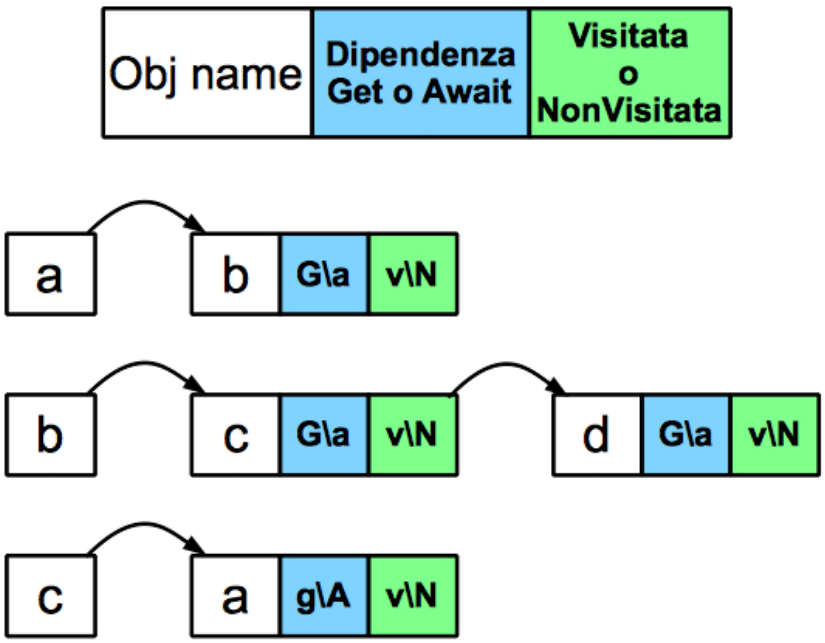

Figura 4.2: Struttura dati dello stato, elemento base dei *lamp*

3. La terza, in verde, anch'essa un flag, racchiude un dato temporaneo utilizzato per la ricerca di circolarità di nomi.

Per cercare una generica circolarità all'interno di uno stato viene eseguito l'algoritmo generico che esegue i seguenti passi:

1. Per ogni oggetto di testa a;
2. Seleziona la lista con testa a e cerca se a è presente nelle terne della stessa lista, se sì, termina segnalando il lock;
3. Per ogni oggetto b nelle terne della lista marca il flag di visita come $Visitato$ e visita la lista con testa b;
4. Se la lista con testa b possiede a tra le terne termina segnalando il lock, altrimenti:
   (a) Se le terne sono esaurite passa al punto 2. con una nuova testa ed azzerando tutti i flag di visita, se le teste sono terminate ferma senza lock;
   (b) Se le terne sono tutte visitate passa al punto 2. come sopra;

(c) Se ci sono ancora terne da visitare prende il nuovo b, marca $Visitato$ e passa al punto 3.

Un'operazione frequente all'interno degli stati è l'aggiunta di una singola dipendenza $(a^{[\mathtt{a}]}, b^{[\mathtt{a}]})$, in questo caso si può ottimizzare la ricerca di un ciclo utilizzando l'algoritmo precedente saltando il punto 1. e cercando direttamente così un ciclo di tipo (a,a), l'unico che potrebbe crearsi aggiungendo $(a^{[\mathtt{a}]}, b^{[\mathtt{a}]})$ ad uno stato che prima di cicli non ne aveva.

Oltre all'algoritmo presentato, con la sua versione ottimizzata, si utilizzano nell'*analizzatore* anche algoritmi simili ausiliari, da eseguire solo alla fine, che sfruttano anche il flag di `get`/`await` per determinare se un ciclo è di pure `get`, di pure `await` oppure misto.

La presenza di un ciclo è fortemente monitorata all'interno della coppia di *lamp* del `main`. L'intero algoritmo di analisi si interrompe nel caso di scoperta di un ciclo nel main, anche se non è ancora stato trovato un punto fisso, infatti, per la monotonia delle operazioni utilizzate, un ciclo una volta trovato non può più scomparire e quindi non ha senso proseguire l'analisi[3].

Tutte le strutture dati come `stati`, `lamp`, e `coppie di lamp` offrono la possibilità di eseguire sostituzioni (cambio di nomi). Tale operazione di sostituzione è fortemente utilizzata nelle regole della Definizione 4.2 per trasferire le informazioni delle chiamate di metodo e garantire ai nomi liberi di rimanerlo sempre (rendendoli fresh ad ogni iterazione).

4.3.2 Saturazione

Il problema della saturazione è stato affrontato tramite una scelta progettuale, infatti non viene contato il numero dei nomi fresh totali generati bensì il numero di iterazioni dell'algoritmo, più semplice da gestire. Ad ogni iterazione (non saturata) viene memorizzata l'ultima sostituzione $[\widetilde{b'}/\widetilde{b}]$ di nomi fresh usata, quando si raggiunge l'ultima iterazione viene segnalato il raggiungimento della *saturazione*. Da questo momento in poi la sostituzione di nomi fresh è sempre la stessa $[\widetilde{b'}/\widetilde{b}]$ e non cambierà più. Sopraggiunta la *saturazione* le operazioni da svolgere sono fortemente ridotte: le regole della Definizione 4.2 vengono private di tutte le sostituzioni, compresa quella sugli oggetti di tipo $\widetilde{a'}$ e tutte quelle applicate ai parametri formali dei metodi invocati. L'unica sostituzione ancora attiva è quella $[\widetilde{b'}/\widetilde{b}]$ dei nomi fresh salvata nell'ultima iterazione non saturata che, come ricordiamo, non cambierà più. Il raggiungimento della saturazione, quindi il riutilizzo di nomi non più

[3] un po' come accade quando un programma viene compilato, se c'è un errore sintattico nel codice si ferma l'analisi e si segnala la cosa all'utente

fresh, può portare all'introduzione di cicli che altrimenti non verrebbero mai inseriti. Vedremo a riguardo nella Sezione 4.4 dell'analizzatore casi in cui questo problema si presenta.

4.3.3 Output

L'analizzatore ha lo scopo di segnalare la presenza dei deadlock all'interno di un programma `ABSFJf`, in particolare del `main`. Le informazioni basilari che vengono presentate all'utente una volta terminata la computazione sono:

- Se è stata raggiunta la saturazione, in modo che l'utente sappia se il raggiungimento del punto fisso è avvenuto con o senza saturazione dei nomi di oggetto;
- Se è stato individuato un ciclo di sole `get` nel `main`;
- Se è stato individuato un ciclo di sole `await` nel `main`;
- Se è stato individuato un ciclo di tipo *misto* all'interno dei `main`, con sia dipendenze di tipo `get`, sia dipendenze di tipo `await`.

Da queste quattro informazioni basilari possiamo individuare tre tipologie di situazioni dalla più alla meno pericolosa:

1. Situazione pericolosa, possibilità di falsi positivi bassa:
   - Assenza di saturazione e ciclo di `get`.
   - Assenza di saturazione e ciclo *misto*.
2. Situazione apparentemente pericolosa ma con buona probabilità di falsi positivi:
   - Presenza di saturazione e ciclo di `get`.
   - Presenza di saturazione e ciclo *misto*.
3. Situazione non pericolosa, assenza di deadlock:
   - Assenza di saturazione e nessun ciclo nel `main`.

A queste informazioni possono essere chiaramente aggiunte stampe di *debug* che arricchiscono l'output come:

– Numero dell'iterazione corrente;

- La CCT con i corrispettivi *lamp* calcolati, sia iterazione per iterazione, sia solo a punto fisso raggiunto;
- I termini coinvolti nel passo dell'iterazione e le sostituzioni da effettuare.

4.4 Esempi

In questa sezione valutiamo esempi *completi* che partono dal codice di un programma `ABSFJf`, passano attraverso la generazione dei contratti e terminano con l'analisi effettuata dall'ultimo algoritmo presentato, l'*analizzatore*. Vengono rivisitati in particolare gli esempi utilizzati nel capitolo 3, al termine della descrizione della semantica di `ABSFJf`. Questa volta vedremo, per ogni programma, anche il codice `ABSFJf` dei metodi della tabella Tabella 2.4[4] e del relativo `main` del programma. Tutti gli esempi utilizzano informazioni grafiche il cui significato è mostrato in Figura 4.3.

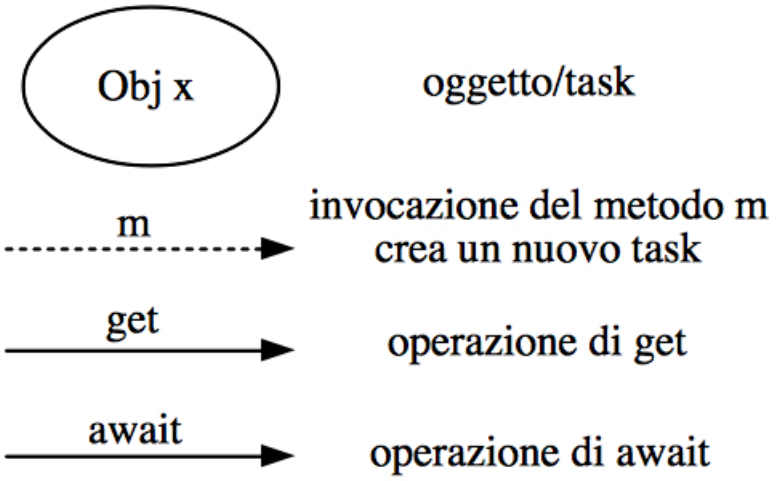

Figura 4.3: Legenda degli esempi

4.4.1 Esempio di corretta terminazione

Il primo esempio è un esempio di corretta terminazione, viene invocato il metodo `n1` nel `main` che semplicemente chiama `m` (il metodo che restituisce banalmente un nuovo oggetto), performa una `await` e successivamente una `get`. Viene riportato uno scenario grafico e di seguito il codice `ABSFJf`:

[4]per contenere la stampa dei contratti già molto lunghi sono state usate le stesse Classi della tabella ma senza campi

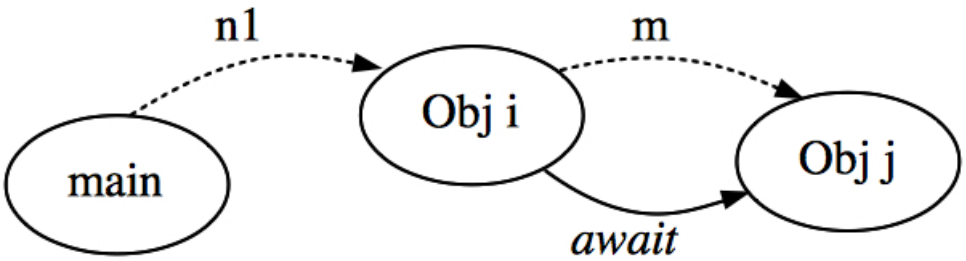

Figura 4.4: Grafo di un programma che termina

```
interface Object {
}
class Object implements Object {
}
interface C {
    C m();
    C n1(C c);
}
class C implements C {
    C m() {
        C x = new cog C();
        return x;
    }
    C n1(C c) {
        Fut<C> fut = c!m();
        await fut?;
        return fut.get;
    }
}

{
C i = new cog C();
C j = new cog C();
Fut<C> fut = i!n1(j);
}
```

L'output che si ottiene compilando tale codice con la nuova versione del compilatore `ABS` è il seguente (vengono riportate solo le parti più significative): l'inferenza dei contratti produce questa **Contract Class Table**:

```
############################################################
C.m:
  'ai[ ]( ) -> 'i[ ] {0}
C.n1:
  'am[ ]( 'ar[ ] ) -> 'ad[ ] ...
     ...  {C.m('ar[ ]( ) -> 'ad[ ]).('am,'ar)@}
Main:
  C.n1('ay[ ]( 'ba[ ] ) -> 'bc)
############################################################
```

l'analisi invece produce queste valutazioni finali del codice:

```
Saturation?                          false
Deadlock in Main?                    false
Await cycle in Main?                 false
Cycle mix of Get/Await in Main?      false
```

Il codice riportato, dopo un'analisi di appena due iterazioni, raggiunge un punto fisso che non presenta alcun ciclo all'interno dei *lamp* ricavati. Quindi il risultato dell'analisi del codice rispecchia la semantica dello stesso esempio nella Sezione 2.5.1.

4.4.2 Esempio di deadlock

Il secondo esempio è un esempio di deadlock, viene invocato il metodo `n2` nel `main` passando sia come `this` che come argomento lo stesso oggetto. Viene riportato uno scenario grafico e di seguito il codice `ABSFJf` :

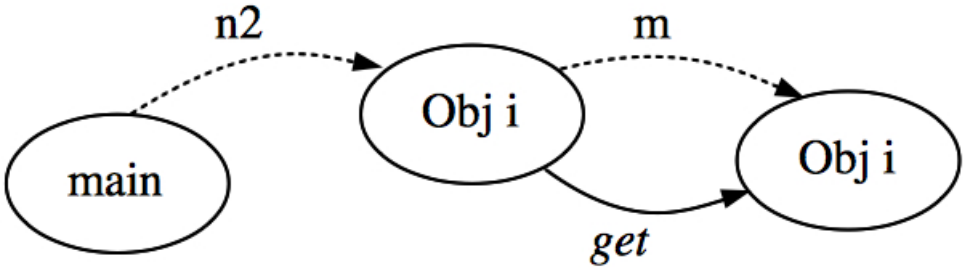

Figura 4.5: Grafo responsabile di deadlock

```
interface Object {
}
class Object implements Object {
}
interface C {
    C m();
    C n2(C c);
}
class C implements C {
    C m() {
        C x = new cog C();
        return x;
    }
    C n2(C c) {
        Fut<C> fut = c!m();
        return fut.get;
    }
}

{
C i = new cog C();
Fut<C> fut = i!n2(i);
}
```

L'output che si ottiene compilando tale codice con la nuova versione del compilatore ABS è il seguente (vengono riportate solo le parti più significative): l'inferenza dei contratti produce questa **Contract Class Table**:

```
############################################################
C.m:
  'ae[ ]( ) -> 'i[ ] {0}
C.n2:
  'ai[ ]( 'an[ ] ) -> 'ab[ ] ...
     ...  {C.m('an[ ]( ) -> 'ab[ ]).('ai,'an)}
Main:
  C.n2('as[ ]( 'as[ ] ) -> 'au)
############################################################
```

l'analisi invece produce queste valutazioni finali del codice:

```
Saturation?                          false
Deadlock in Main?                    true
Await cycle in Main?                 false
Cycle mix of Get/Await in Main?      false
```

Il codice riportato, dopo un'analisi di appena due iterazioni, raggiunge un punto fisso che presenta un ciclo all'interno dei *lamp* del `main`, in particolare si tratta di un ciclo di pure `get`. Quindi il risultato dell'analisi del codice rispecchia la semantica dello stesso esempio nella Sezione 2.5.2.

4.4.3 Esempio di creazione infinita

Il terzo esempio è un esempio di creazione infinita di nomi, l'equivalente di un programma che non termina mai, viene invocato il metodo `p` nel `main` che crea nuovi oggetti e reinvoca `p` su questi. Viene riportato uno scenario grafico e di seguito il codice `ABSFJf`:

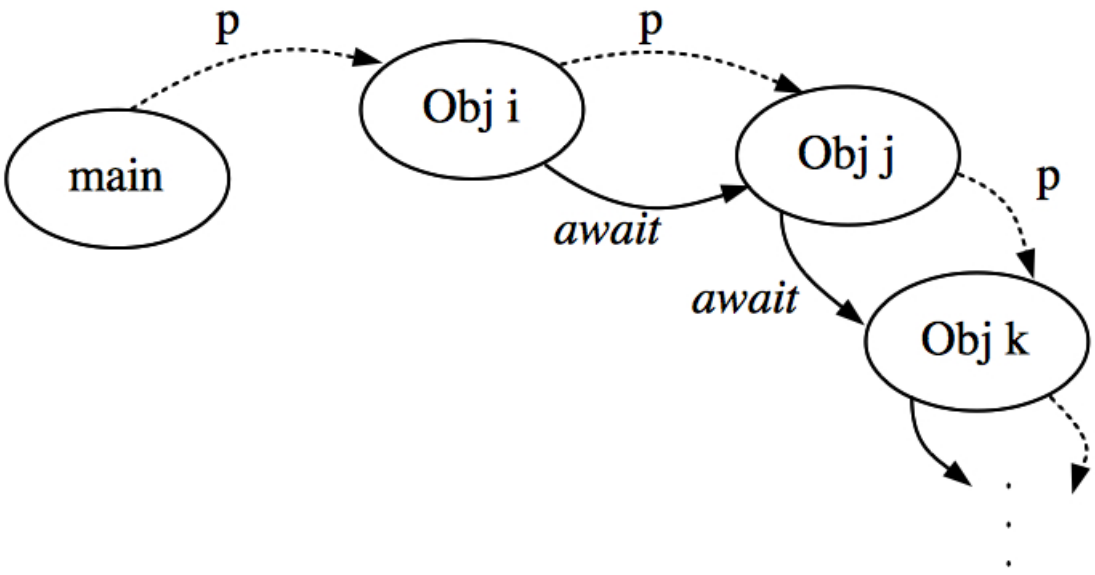

Figura 4.6: Grafo di non terminazione

```
interface Object {
}
class Object implements Object {
}
interface C {
    C m();
    C p();
}
class C implements C {
    C m() {
        C x = new cog C();
        return x;
    }
    C p() {
        C a = new cog C();
        Fut<C> fut = a!p();
        await fut?;
        return fut.get;
    }
}

{
C i = new cog C();
Fut<C> fut = i!p();
}
```

L'output che si ottiene compilando tale codice con la nuova versione del compilatore *ABS* è il seguente (vengono riportate solo le parti più significative): l'inferenza dei contratti produce questa **Contract Class Table**:

```
############################################################
C.m:
  'ai[ ]( ) -> 'i[ ] {0}
C.p:
  'am[ ]( ) -> 'an ...
     ...  {C.p('ap[ ]( ) -> 'an).('am,'ap)@}
Main:
  C.p('az[ ]( ) -> 'bb)
############################################################
```

l'analisi invece produce queste valutazioni finali del codice:

```
Saturation?                          true
Deadlock in Main?                    false
Await cycle in Main?                 true
Cycle mix of Get/Await in Main?      false
```

Il codice riportato, in questo caso, non raggiunge un punto fisso in poche iterazioni, bensì lo raggiunge a seguito di una saturazione dei nomi fresh. Il flag di saturazione infatti risulta attivo al termine della computazione e oltre ad esso anche il flag che segnala un ciclo di `await` nel main. A seguito della saturazione, infatti, all'interno del main non viene più inserita una coppia di nomi fresh, bensì la coppia identica (a, a), con il nome a che dipende chiaramente dal numero di variabili fresh generate. Quindi il risultato dell'analisi del codice rispecchia la semantica dello stesso esempio nella Sezione 2.5.3, a meno di saturazione che costringe l'analizzatore a trovare uno "stato finale".

4.4.4 Esempio di indecisione da scheduler

Il quarto esempio è un esempio in cui la scelta dello scheduler è cruciale, infatti lo stesso esempio può portare ad una corretta terminazione oppure a un deadlock a seconda dell'ordine di valutazione dei task. Viene invocato il metodo `q` nel `main`. Il metodo `q` riceve due oggetti in ingresso, a e b, ed invoca due volte il metodo `n2`, usando prima a come `this` e b come argomento, poi i nomi scambiati. Viene riportato uno scenario grafico e di seguito il codice `ABSFJf`:

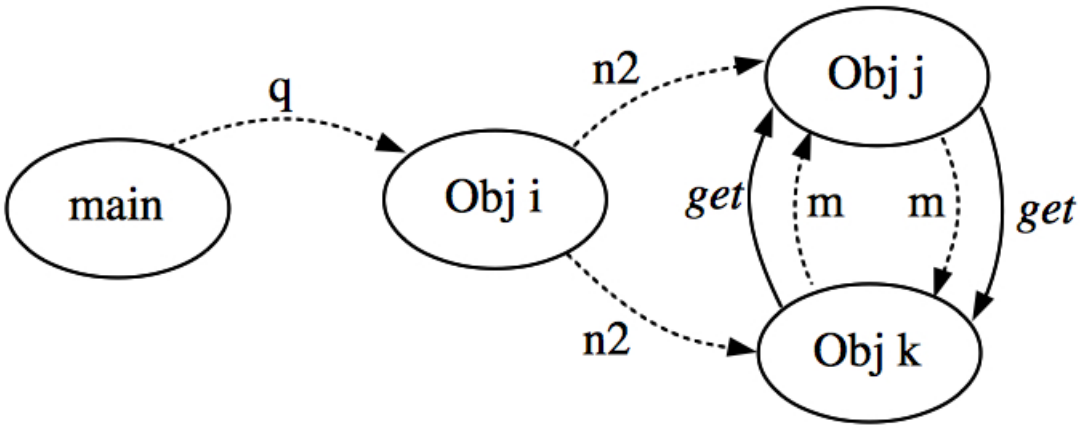

Figura 4.7: Grafo di possibile lock da scheduler

```
interface Object {
}
class Object implements Object {
}
interface C {
    C m();
    C n2(C c);
    Fut<C> q(C a, C b);
}
class C implements C {
    C m() {
        C x = new cog C();
        return x;
    }
    C n2(C c) {
        Fut<C> fut = c!m();
        return fut.get;
    }
    Fut<C> q(C a, C b) {
        Fut<C> fut1 = a!n2(b);
        Fut<C> fut2 = b!n2(a);
        return fut2;
    }
}

{
C i = new cog C();
C j = new cog C();
C k = new cog C();
Fut<C> fut = i!q(j,k);
}
```

L'output che si ottiene compilando tale codice con la nuova versione del compilatore ABS è il seguente (vengono riportate solo le parti più significative): l'inferenza dei contratti produce questa **Contract Class Table**:

```
##########################################################
C.m:
  'bc[ ]( ) -> 'm[ ] {0}
C.n2:
  'bg[ ]( 'bl[ ] ) -> 'ap[ ] ...
     ...  {C.m('bl[ ]( ) -> 'ap[ ]).('bg,'bl)}
C.q:
  'bo[ ]( 'af[ ] 'ae[ ] ) -> 'ae   > 'ba ...
     ...  {C.n2('af[ ]( 'ae[ ] ) -> 'bt); ...
     ...  C.n2('ae[ ]( 'af[ ] ) -> 'ba)}
Main:
  C.q('ca[ ]( 'cc[ ] 'ce[ ] ) -> 'cg)
##########################################################
```

l'analisi invece produce queste valutazioni finali del codice:

```
Saturation?                          false
Deadlock in Main?                    true
Await cycle in Main?                 false
Cycle mix of Get/Await in Main?      false
```

Il codice riportato, dopo un'analisi di appena tre iterazioni, raggiunge un punto fisso che presenta un ciclo all'interno dei *lamp* del `main`, in particolare si tratta di un ciclo di pure `get`. Quindi il risultato dell'analisi del codice rispecchia la semantica dello stesso esempio nella Sezione 2.5.4 interpretando una situazione ambigua che può portare ad un deadlock come situazione pericolosa segnalando la presenza di deadlock (che potrebbe non avvenire, ma staticamente è giusto segnalarlo).

4.4.5 Esempio di livelock

Il quinto esempio è un esempio in cui viene prodotto un livelock, ovvero, mentre alcuni processi sono bloccati, almeno uno prende e rilascia costantemente il lock tenendo attivo il processore. Viene invocato il metodo `h` nel `main`. Il metodo `h` riceve un oggetto in ingresso ed invoca il metodo `n1` su tale oggetto passando il `this` come argomento. Viene riportato uno scenario grafico e di seguito il codice `ABSFJf`:

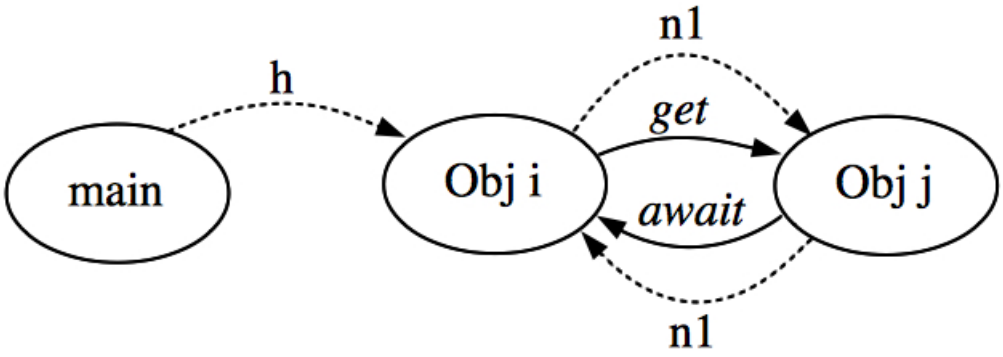

Figura 4.8: Grafo di un livelock

```
interface Object {
}
class Object implements Object {
}
interface C {
    C m();
    C n1(C c);
    C h(C b);
}
class C implements C {
    C m() {
        C x = new cog C();
        return x;
    }
    C n1(C c) {
        Fut<C> fut = c!m();
        await fut?;
        return fut.get;
    }
    C h(C b) {
        Fut<C> fut = b!n1(this);
        return fut.get;
    }
}

{
C i = new cog C();
C j = new cog C();
Fut<C> fut =i!h(j);
}
```

L'output che si ottiene compilando tale codice con la nuova versione del compilatore ABS è il seguente (vengono riportate solo le parti più significative): l'inferenza dei contratti produce questa **Contract Class Table**:

```
#########################################################
C.m:
  'bb[ ]( ) -> 'l[ ] {0}
C.n1:
  'bf[ ]( 'bk[ ] ) -> 'ao[ ] ...
     ...  {C.m('bk[ ]( ) -> 'ao[ ]).('bf,'bk)@}
C.h:
  'bp[ ]( 'bu[ ] ) -> 'bw ...
     ...  {C.n1('bu[ ]( 'bp[ ] ) -> 'bw).('bp,'bu)
Main:
  C.h('bz[ ]( 'cb[ ] ) -> 'cd)
#########################################################
```

l'analisi invece produce queste valutazioni finali del codice:

```
Saturation?                          false
Deadlock in Main?                    false
Await cycle in Main?                 false
Cycle mix of Get/Await in Main?      true
```

Il codice riportato, dopo un'analisi di appena tre iterazioni, raggiunge un punto fisso che presenta un ciclo all'interno dei *lamp* del `main`, in particolare si tratta di un ciclo *misto* di `get` ed `await`. Quindi il risultato dell'analisi del codice rispecchia la semantica dello stesso esempio nella Sezione 2.5.5 in cui notiamo che anche i livelock vengono catturati dall'analisi come cicli di dipendenza misti.

Capitolo 5

Conclusioni

Abbiamo presentato, sviluppato ed implementato una tecnica per l'analisi statica dei deadlock in un linguaggio concorrente orientato agli oggetti che è basata sulla descrizione astratta del comportamento dei metodi. Tali descrizioni astratte vengono poi analizzate costruendo un modello a stati finiti e controllando dipendenze tra nomi di oggetto.

Lo studio può essere esteso in diverse direzione. Una direzione è quella di studiare una tecnica per aumentare la precisione e l'accuratezza dell'algoritmo di punto fisso sui *lamp*, che, al momento, è impreciso a causa della saturazione, quando si raggiunge il limite nella creazione di nuovi nomi. Una possibilità è quella di usare automi a stati finiti con creazione di nomi, come quelli in [26], nei passi di modellazione del capitolo 5 e studiare i deadlock su questi automi. Alternativamente è possibile cercare di individuare pattern ricorrenti di creazione di oggetti e di dipendenza tra oggetti. Dopodiché questi pattern potrebbero essere modellati in qualche maniera (finita) per verificare la loro proprietà di lock-safe o meno.

Un'altra direzione riguarda l'estensione del linguaggio `ABSFJf`. Una estensione è quella di (re)introdurre invocazioni sincrone dei metodi, ad esempio per chiamate ricorsive (tail-recursion ad esempio). Tale estensione è gravosa in quanto richiede la modifica delle regole semantiche, delle regole dei contratti e della tecnica di modellazione, ma non è difficile dal punto di vista teorico. Un'estensione più complessa richiede l'introduzione dell'aggiornamento dei campi e delle variabili. In un caso simile ci sarebbero importanti cambiamenti nella semantica del linguaggio, come l'introduzione di heap e di correzioni alla Definizione 4.4 di stato di lock, rimuovendo la terza voce, e alla Definizione 4.5 di chiusura di `get`. Un'altra possibile estensione riguarda invece l'introduzione dei costrutti stile if-then-else e dei dati primitivi già presenti in `ABS` per estendere l'analisi implementata ad algoritmi più completi, ciò comporterebbe lo studio e l'implementazione delle regole di inferenza per

estrarre i contratti da queste estensioni, mentre l'*analizzatore* non subirebbe alcuna modifica. Infatti, se da una parte l'algoritmo di inferenza dei contratti è intimamente collegato allo specifico linguaggio di programmazione (opera sull'albero sintattico prodotto dal compilatore), d'altra parte, l'*analizzatore* opera solo sulla CCT e quindi sui contratti, non curandosi della provenienza di questi che potrebbero arrivare dall'inferenza su programmi dei più disparati linguaggi di programmazione.

Bibliografia

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28, 2006.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] P. America, J. de Bakker, J. N. Kok, and J. J. M. M. Rutten. Operational semantics of a parallel object-oriented language. In *Proceedings of the 13th Principles of programming languages (POPL '86)*, pages 194–208, New York, NY, USA, 1986. ACM.

[4] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe program.: preventing data races and deadlocks. In *Proc. OOPSLA '02*, pages 211–230. ACM, 2002.

[5] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31:560–599, 1984.

[6] R. Carlsson and H. Millroth. On cyclic process dependencies and the verification of absence of deadlocks in reactive systems, 1997.

[7] S. Chaki, S. K. Rajamani, and J. Rehof. Types as models: model checking message-passing programs. *SIGPLAN Not.*, 37(1):45–57, 2002.

[8] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *In PLDI 03: Programming Language Design and Implementation*, pages 338–349. ACM, 2003.

[9] E. Giachino, C. Laneve, and T. A. Lascu. Deadlock and livelock analysis in concurrent objects with futures. `www.cs.unibo.it/~laneve/publications.html`, 2011. Submitted.

[10] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the*

3rd International Workshop on Computer Aided Verification, CAV '91, pages 332–342, London, UK, UK, 1992. Springer-Verlag.

[11] G. Hedin and E. Magnusson. Jastadd: an aspect-oriented compiler construction system. *Sci. Comput. Program.*, 47(1):37–58, Apr. 2003.

[12] F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, Apr. 1993.

[13] A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004.

[14] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23:396–450, 2001.

[15] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, volume 6957 of *LNCS*, pages 142–164. Springer-Verlag, 2011.

[16] E. B. Johnsen and O. Owe. An asynchronous communication model for distributed concurrent objects. *Software and System Modeling*, 6(1):39–58, 2007.

[17] V. Khomenko and M. Koutny. Lp deadlock checking using partial order dependencies. In *Proceedings of the 11th International Conference on Concurrency Theory*, CONCUR '00, pages 410–425, London, UK, UK, 2000. Springer-Verlag.

[18] N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Trans. Program. Lang. Syst.*, 20(2):436–482, 1998.

[19] N. Kobayashi. A new type system for deadlock-free processes. In *Proc. CONCUR 2006*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[20] C. Laneve and L. Padovani. The *must* preorder revisited. In *Proc. CONCUR 2007*, volume 4703 of *LNCS*, pages 212–225. Springer, 2007.

[21] G. R. Lavender and D. C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. *Proc.Pattern Languages of Programs,*, 1995.

[22] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of Programming Language design and Implementation (PLDI '88)*, pages 260–267, New York, NY, USA, 1988. ACM.

[23] S. P. Masticola. *Static Detection Of Deadlocks In Polynomial Time.* PhD thesis, 1993.

[24] R. Milner. *A Calculus of Communicating Systems.* Springer, 1982.

[25] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, ii. *Inf. and Comput.*, 100:41–77, 1992.

[26] U. Montanari and M. Pistore. History-dependent automata: An introduction. In *Formal Methods for the Design of Computer, Communication, and Software Systems*, volume 3465 of *LNCS*, pages 1–28. Springer, 2005.

[27] J. Niehren, J. Schwinghammer, and G. Smolka. A concurrent lambda calculus with futures. *Theor. Comput. Sci.*, 364, 2006.

[28] H. R. Nielson and F. Nielson. Higher-order concurrent programs with finite communication topology. In *Proc. POPL '94*, pages 84–97. ACM, 1994.

[29] S. Parastatidis and J. Webber. *MEP SSDL Protocol Framework*, Apr. 2005. `http://ssdl.org`.

[30] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Programming Languages and Systems*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.

[31] K. Suenaga and N. Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In *Programming Languages and Systems*, volume 4421 of *LNCS*, pages 490–504. Springer, 2007.

[32] M. Torgersen. Asyncrony in .NET. October 2010.

[33] V. T. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Proc. PLACES'09*, volume 17 of *EPTCS*, pages 95–109, 2009.

[34] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *Proceedings of the 20th Object-oriented programming, systems, languages, and applications (OOPSLA '05)*, pages 439–453, New York, NY, USA, 2005. ACM.

[35] A. Yonezawa, editor. *ABCL: an object-oriented concurrent system.* MIT Press, Cambridge, MA, USA, 1990.