Alma Mater Studiorum · Università di Bologna

facoltà di scienze matematiche, fisiche e naturali
Corso di Laurea Magistrale in Informatica

Un prototipo per l'analisi di circolarità

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Presentata da:
Natale Patriciello

Correlatori:
Dott.ssa Elena Giachino

Sessione III
Anno Accademico 2011/2012

Abstract

Ogni programma concorrente può manifestare deadlock a causa di errori compiuti dal programmatore. In questa tesi si è studiato e implementato una tecnica di rilevazione statica dei deadlock, basata sulla teoria delle mutazioni (una generalizzazione della teoria delle permutazioni), su modelli che definiscono insiemi di dipendenze, i cosiddetti *LAM*. La tecnica si rivela precisa nel caso dei cosiddetti *LAM lineari*, mentre è imprecisa ma comunque *corretta* nel caso generale. La tecnica è stata implementata in linguaggio Java, ed è possibile costruire un analizzatore statico di deadlock per un qualsiasi linguaggio di programmazione previa definizione di una associazione tra programmi e LAM.

Indice

Capitolo 1

Introduzione

Il lavoro di tesi presenta una tecnica statica, i cui dettagli si possono trovare in [1], e la sua implementazione per scoprire deadlock in programmi scritti tramite un linguaggio con risorse, identificate da *nomi*, come i linguaggi ad oggetti di comune utilizzo. I punti salienti della tecnica stessa sono:

1. essa non usa nessun ordine parziale pre-definito

2. gestisce agevolmente la creazione dinamica di nomi.

Per scovare staticamente i deadlock si introduce un modello, chiamato *LAM* (un acronimo per *deadLock Analysis Model*), nel quale gli stati mantengono informazioni sulle dipendenze tra nomi, scritte d'ora in poi come coppie (x, y), il cui significato è letteralmente "il rilascio della risorsa x dipende dal rilascio della risorsa y". In particolare, ogni LAM è un insieme di relazioni sui nomi e un potenziale deadlock è segnalato dalla presenza di una *dipendenza circolare* (chiamata anche *circolarità*) in qualche sua relazione: se dovessero essere presenti, ad esempio, le dipendenze (x, y) e (y, x) si potrebbe affermare che esiste una circolarità. I modelli LAM sono intesi come modelli astratti di programmi reali: *gli stati di un LAM definiscono le dipendenze negli stati del programma reale corrispondente*. Essi sono dunque composti da definizioni di funzione e di un termine $Main$ da valutare. Un LAM potrebbe essere

$$\Big(\mathtt{f}(x, y, z) = (x, y) \| \mathtt{f}(y, z, x),\ \mathtt{f}(u, v, w)\Big),$$

in cui la funzione `f` produce una dipendenza (x, y) e una invocazione ricorsiva di `f` (dove $\|$ è interpretato come l'unione tra (x, y) e le dipendenze prodotte da $\mathtt{f}(y, z, x)$); il termine $\mathtt{f}(u, v, w)$ è il termine $Main$. Le espansioni delle invocazioni di `f` danno la sequenza di stati

$$\begin{array}{lcl} \mathtt{f}(u, v, w) & \longrightarrow & (u, v) \| \mathtt{f}(v, w, u) \\ & \longrightarrow & (u, v) \| (v, w) \| \mathtt{f}(w, u, v) \\ & \longrightarrow & (u, v) \| (v, w) \| (w, u) \| \mathtt{f}(u, v, w)\ , \end{array}$$

corrispondenti alle relazioni `0` (stato iniziale), (u, v) (secondo stato), $(u, v) \| (v, w)$ (terzo stato), e $(u, v) \| (v, w) \| (w, u)$.

Dato che i LAM gestiscono la creazione dinamica di nomi, il modello sottostante *non può* essere a stati finiti. Se si avesse una funzione $\mathtt{f}'(x) = (x, y) \| \mathtt{f}'(y)$, dato che y è libera, si avrebbe un nuovo nome ad ogni espansione (il che porta ad avere stati infiniti, uno per ognuna di esse). Il principale contributo della tecnica che verrà descritta è la risposta al

> PROBLEMA: *è decidibile scoprire se la computazione di un programma LAM produce una circolarità?*

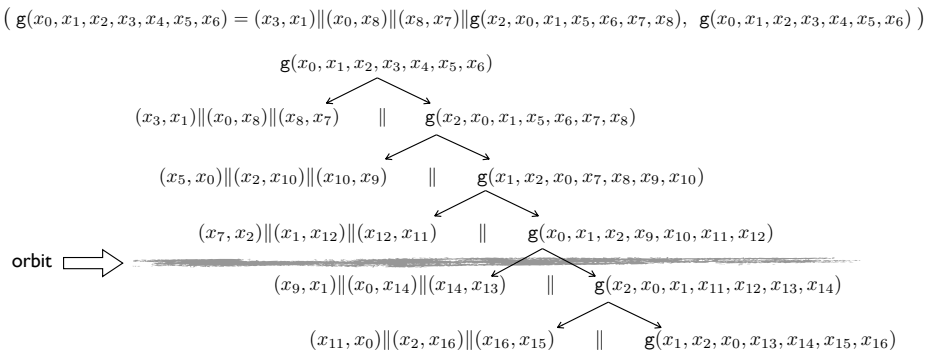


Figura 1.1: Un programma lam e le sue espansioni

La risposta è immediata quando le funzioni non sono (mutuamente) ricorsive: è sufficiente *espandere le invocazioni nel termine da valutare*. La risposta è semplice anche nel caso in cui

(i) le funzioni sono *lineari*, cioè in cui la (mutua) ricorsione è del tipo della funzione fattoriale, come ad esempio la funzione `f` descritta in precedenza;

(ii) le invocazioni di funzioni non hanno argomenti duplicati e le loro definizioni non creano nuovi nomi.

Quando valgono (i) e (ii), le funzioni ricorsive possono essere considerate come *permutazioni di nomi* – tecnicamente si andrà a definire in seguito una nozione di *(per)mutazione associata* – e la teoria corrispondente [2] garantisce che, applicando ripetutamente la stessa permutazione ad una tupla di nomi, ad un certo punto si otterrà la permutazione iniziale. Questo punto, conosciuto come *orbita* della permutazione, permette di definire il seguente algoritmo per il PROBLEMA: si calcoli l'orbita della permutazione associata alla funzione e si espanda la funzione stessa nel termine da valutare per un numero di volte pari all'orbita stessa. Per esempio, la permutazione della funzione `f` ha orbita 3. Dunque, è possibile fermare la valutazione di `f` dopo la terza espansione (nello stato $(u, v)\|(v, w)\|(w, u)\|\mathtt{f}(u, v, w)$) perché ogni coppia di dipendenza prodotta in seguito apparterrà sempre alla relazione $(u, v)\|(v, w)\|(w, u)$.

Quando si invalida la (ii), la risposta al PROBLEMA non è più così semplice. Si consideri la funzione $\mathtt{h}(x, y, z) = (y, z)\|\mathtt{h}(x, x, w)$ e si valuti $\mathtt{h}(u, v, w)$. Si ottiene

$$\begin{aligned} \mathtt{h}(u, v, w) &\longrightarrow (v, w)\|\mathtt{h}(u, u, w') \\ &\longrightarrow (v, w)\|(u, w')\|\mathtt{h}(u, u, w'') \\ &\longrightarrow (v, w)\|(u, w')\|(u, w'')\|\mathtt{h}(u, u, w''') \ . \end{aligned}$$

In altre parole, la valutazione non ritornerà mai $\mathtt{h}(u, v, w)$, così come qualsiasi altra invocazione negli stati successivi, perché la definizione di `h` contiene una invocazione ricorsiva dove il primo argomento è duplicato, il secondo e il terzo sono cancellati e nomi nuovi vengono creati al loro posto. Si noti come dal terzo stato in poi, le invocazioni di `h` non sono identiche, ma lo possono diventare attraverso *un' associazione* la quale

– associa nomi creati nell'ultimo passagio a nomi passati,

– corrisponde all'identità su tutti gli altri nomi.

La definizione dell'associazione, chiamata *flashback*, richiede che la trasformazione dei nomi della funzione in un LAM, la quale verrà chiamata *mutazione*, registri anche la creazione di nomi. In questo caso, la teoria delle mutazioni permette di mappare $\mathtt{h}(u, u, w'')$ in $\mathtt{h}(u, u, w')$ tramite il fatto che w'' è stato creato dopo w', *cioè* $w' < w''$. Analogamente, si avrà $w'' < w'''$, e così via.

Si generalizzerà quindi il risultato sulle orbite delle permutazioni:

> *applicando ripetutamente la stessa mutazione ad una tupla di nomi, ad un certo punto si otterrà una tupla che è identica, al più per un flashback, ad una tupla già incontrata in passato.*

Il punto è chiamato, come nelle permutazioni, *orbita* della mutazione, che si proverà essere calcolabile.

Purtroppo, espandere una funzione un numero di volte pari all'orbita della mutazione associata spesso non è sufficiente per mostrare circolarità. Il fatto è che le mutazioni e i flashback si concentrano sulle invocazioni di funzione, e non tengono in considerazione le dipendenze: nel caso delle permutazioni, ciò bastava perché esse riproducono *le stesse* dipendenze delle passate invocazioni, mentre con le mutazioni non è necessariamente così. Per esempio, in Figura 1.1 si vede un programma dove la funzione `g` ha orbita 3. Le prime tre espansioni di $\mathtt{g}(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$ sono quelle al di sopra della linea orizzontale – in effetti, esiste un flashback da $\mathtt{g}(x_0, x_1, x_2, x_9, x_{10}, x_{11}, x_{12})$ a $\mathtt{g}(x_0, x_1, x_2, x_3, x_4, x_5, x_6)$. Si osservi come le coppie prodotte fino alla terza espansione

$$\begin{array}{ll} & (x_3, x_1) \| (x_0, x_8) \| (x_8, x_7) \\ \| & (x_5, x_0) \| (x_2, x_{10}) \| (x_{10}, x_9) \\ \| & (x_7, x_2) \| (x_1, x_{12}) \| (x_{12}, x_{11}) \ , \end{array}$$

non manifestino alcuna circolarità. Eppure, espandendo la funzione per altre due volte (mostrate al di sotto della linea orizzontale della Figura 1.1), si ottiene la circolarità

$$\begin{array}{c} (x_0, x_8) \| (x_8, x_7) \| (x_7, x_2) \| (x_2, x_{10}) \| (x_{10}, x_9) \\ \| (x_9, x_1) \| (x_1, x_{12}) \| (x_{12}, x_{11}) \| (x_{11}, x_0) \ . \end{array}$$

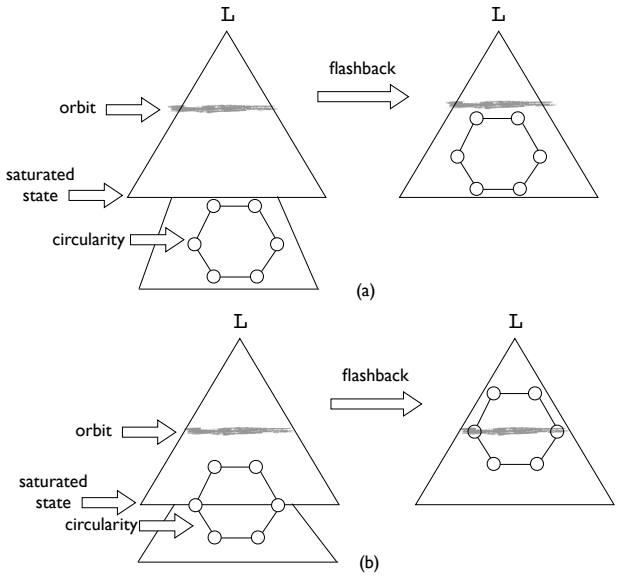


Figura 1.2: Flashback di circolarità

Tornando al PROBLEMA, si vedrà come una condizione sufficiente per decidere se un LAM produrrà mai una circolarità è espandere la funzione *due volte* l'orbita della mutazione associata, chiamando lo stato risultante come stato *saturato*. Se non saranno presenti circolarità nello stato saturato, il LAM sarà dunque libero da circolarità. La ragione per questa doppia valutazione è il fatto che le circolarità possono non essere *create completamente* dalle dipendenze introdotte nell'ultima riduzione (come in Figura 1.2.a), ma possono essere delle composizioni tra nuove dipendenze e quelle create nel passato, come in Figura 1.1. In questi casi, è possibile provare che le circolarità create dopo lo stato saturato possono essere mappate da flashback a circolarità già presenti nello stato saturato, come si può vedere in Figura 1.2.b.

La linearità dei programmi LAM è comunque alla base nell'algoritmo appena descritto, seppur informalmente, per determinare la saturazione di uno stato. Nei LAM non lineari, cioè quando non vale nemmeno la (i), le funzioni hanno *diverse* mutazioni associate. Un esempio di funzione non lineare è quando la ricorsione è del tipo della funzione di Fibonacci:

$$\Big(\texttt{fib}(x) = (x, y) \| (x, z) \| \texttt{fib}(y) \| \texttt{fib}(z),\ \texttt{fib}(u)\Big).$$

Per trattare questo tipo di programmi si è scelto di trasformarli in programmi lineari *introducendo dipendenze*, cioè introducendo falsi positivi in termini di circolarità, invece che analizzarli direttamente da non lineari. Il programma `fib` è trasformato nel seguente programma lineare, qui riportato

in una sua versione semplificata:

$$\left(\begin{array}{l} \texttt{fib}^{aux}(x, x') = (x, y) \| (x, z) \| (x', y) \| (x', z) \| \texttt{fib}^{aux}(y, z) \ , \\ \texttt{fib}^{aux}(u, u) \end{array} \right) .$$

Per evidenziare le dipendenze "fasulle" aggiunte da $\texttt{fib}^{aux}$, si noti come, dopo due espasioni, $\texttt{fib}^{aux}(u, u)$ dia

$$(u, v) \| (u, w) \| (v, v') \| (v, w') \| (w, v') \| (w, w') \| \texttt{fib}^{aux}(v', w') \ .$$

Ma al contrario `fib` dopo quattro passaggi ha uno stato corrispondente a

$$\begin{array}{c} (u, v) \| (u, w) \| (v, v') \| (v, v'') \| (w, w') \| (w, w'') \\ \| \texttt{fib}(v') \| \texttt{fib}(v'') \| \texttt{fib}(w') \| \texttt{fib}(w'') \ , \end{array}$$

che non ha dipendenze tra nomi creati dalle differenti invocazioni. È imporante osservare come le dipendenze aggiunte non possono essere eliminate completamente, per una questione di cardinalità che diverrà più chiara nel Capitolo 4.

In ogni caso, la tecnica di traformazione è *corretta*: se il programma trasformato in lineare non ha circolarità, allora anche l'originale (non lineare) non ha circolarità. In particolare, dato che l'analisi permette di determinare che lo stato saturato di $\texttt{fib}^{aux}$ è libero da circolarità, si può concludere che anche `fib` non ha circolarità.

È possibile applicare la tecnica anche a programmi reali, scritti in linguaggi ad oggetti, previa definizione di una associazione tra programmi stessi e LAM. Nell' articolo [3], vengono illustrati alcuni esempi di programmi JAVA che possono manifestare deadlock. La tesi è incentrata sul lavoro di implementazione della stessa, nell'ottica di sostituire la tecnica di ricerca deadlock attualmente implementata e descritta nel documento [4].

1.1 Stato dell'arte

Analisi statica di deadlock. L'analisi dei deaclock basata sui tipi è stata fortemente studiata. Alcune proposte riguardano calcoli di processi [5, 6, 7, 8], ma alcuni contributi riguardano anche i deadlock nei programmi orientati agli oggetti [9, 10, 11].

Kobayashi, in lavori sul pi-calcolo [12, 13, 5], 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 [6, 7] applica la tecnica di Kobayashi ai linguaggi concorrenti con risorse. Risulta facile estrarre il modello LAM da un programma in un linguaggio con interrupt di [6] e, pertanto, essere in grado di verificare la presenza di deadlock altrettanto facilmente. In ogni caso, un preciso confronto con [6] non è stato fatto. Il linguaggio di [7] utilizza referenze mutabili, le quali sono un noto aspetto che, insieme alla concorrenza, porta ad un comportamento non deterministico. L'applicazione della tecnica illustrata in questa tesi ad un linguaggio con assegnamenti sarà materia di ricerca futura e le referenze sopra riportate saranno sicuramente un punto di partenza molto importante.

In altri contributi, 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 tecnica che verrà illustrata 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.

Una ulteriore differenza con i lavori sopra citati è l'utilizzo dei contratti. I contratti sono termini in un processo algebrico [14]. L'utilizzo di un semplice processo algebrico (quindi a stati finiti) per descrivere protocolli (di comunicazione o sincronizzazione) non è nuovo. Questo è il caso di soluzioni di scambio in SSDL [15], che si basano su CSP [16] e il pi- calcolo [17], o sui tipi comportamentali in [18] ed in [19], che usano CCS [20].

Altri approcci statici, che non si basano sui tipi, sono [21, 22] dove le dipendenze circolari all'interno dei processi sono rilevate come configurazioni erronee, ma la creazione dinamica di nomi non è trattata. Il modello usato in [23] è basato su una tecnica standard di raggiungimento del punto fisso, gestendo infinite creazioni di nomi attraverso approssimanti finiti. Nel presente lavoro si supera questo limite attraverso il riconoscimento di un comportamento ricorsivo, in modo da ridurre l'analisi ad una porzione finita della computazione, stando però sicuri che ciò che potrà seguire sarà della stessa forma, o meglio, mostrerà gli stessi deadlock.

Languaggi e Contratti. Termini simili ai LAM sono stati studiati in [23] per un linguaggio della famiglia Creol [24] con l'obiettivo di controllare la presenza di deadlock. [In particolare, il linguaggio di [23] è simile a `Java` con tipi di dato futuri e l'operazione `get` descritta in [25].] La tecnica di derivazione dei LAM da programmi reali, non presente in questa tesi ma descritta in [3], è stata già prototipata in [4]. Il sistema di inferenza, quando

applicato a programmi reali necessita di alcune annotazioni addizionali, per poter superare le difficoltà legate all'utilizzo di tipi di dato strutturati e le iterazioni.

1.2 Organizzazione della tesi

La tesi è organizzata come segue. Nel Capitolo 2 verrà illustrata la teoria delle mutazioni e dei flashback, insieme alla definizione dei modelli e del linguaggio LAM. Il Capitolo 3 svilupperà l'analisi statica per la ricerca di deadlock per programmi LAM lineari, e ne illustrerà la corrispondente implementazione, mentre nel Capitolo 4 verranno trattati i programmi non lineari, in particolare il caso pseudolineare e le trasformazioni da programma pseudo-lineare a lineare. Le conclusioni, insieme ad indicazioni per ulteriori sviluppi del lavoro di tesi verranno presentate nel Capitolo 5.

Capitolo 2

Mutazioni e LAM

2.1 Mutazioni

Per definire le mutazioni si useranno numeri naturali, rappresentati con a, b, i, j, m, n, ..., con eventuali indici.

Sia V un insieme infinito di nomi, che spaziano su $x, y, z, \cdots$. Verranno usate relazioni d'ordine parziali su insiemi di nomi, le quali saranno riflessive, antisimmetriche e transitive, rappresentate attraverso $\mathbb{V}, \mathbb{V}', \mathbb{I}, \cdots$. Sia $x \in \mathbb{V}$ se, per qualche z, vale $(x, z) \in \mathbb{V}$ oppure $(z, x) \in \mathbb{V}$. Sia anche $var(\mathbb{V}) = \{x \mid x \in \mathbb{V}\}$. Per una convenienza di notazione, si scriverà $\widetilde{x}$ per indicare liste di nomi come $x_1, \ldots, x_n$.

Sia $\mathbb{V} \oplus \widetilde{x}{<}\widetilde{z}$, con $\widetilde{x} \in \mathbb{V}$ e $\widetilde{z} \notin \mathbb{V}$, il più piccolo ordine parziale $\mathbb{V}'$ che soddisfi la regola:

$$\mathbb{V} \subseteq \mathbb{V}' \qquad \frac{x \in \widetilde{x} \quad (x, y) \in \mathbb{V}' \quad z \in \widetilde{z}}{(y, z) \in \mathbb{V}'}$$

Si dice che $\widetilde{z}$ sono i *nomi massimali* di $\mathbb{V} \oplus \widetilde{x}{<}\widetilde{z}$. In altre parole, $\mathbb{V} \oplus x{<}z$ estende l' ordine parziale $\mathbb{V}$ con un nome z che é un *estremo superiore* di ogni catena che parte da nomi uguali o più piccoli di x.

Per esempio,

- $\{(x, x)\} \oplus x{<}z = \{(x, x), (x, z), (z, z)\}$;
- se $\mathbb{V} = \{(x, y), (x', y')\}$ (mancano le coppie riflessive) allora $\mathbb{V} \oplus y{<}z$ é la chiusura riflessiva e transitiva di $\{(x, y), (x', y'), (y, z)\}$;
- se $\mathbb{V} = \{(x, y), (x, y')\}$ (mancano le coppie riflessive) allora $\mathbb{V} \oplus x{<}z$ é la chiusura transitiva e riflessiva di $\{(x, y), (x, y'), (y, z), (y', z)\}$.

Sia $x \leq y \in \mathbb{V}$ scritto come $(x, y) \in \mathbb{V}$.

Definizione 2.1. Una *mutazione* di tuple di nomi, scritta come $[\![a_1, \cdots, a_n]\!]$ dove $1 \leq a_1, \cdots, a_n \leq 2 \times n$, trasforma una coppia $\left\langle \mathbb{V}, (x_1, \cdots, x_n) \right\rangle$ in una coppia $\left\langle \mathbb{V}', (x'_1, \cdots, x'_n) \right\rangle$ come segue. Sia $\{b_1, \cdots, b_k\} = \{a_1, \cdots, a_n\} \setminus \{1, 2, \cdots, n\}$ e $z_{b_1}, \cdots, z_{b_k}$ siano k differenti nomi nuovi. [Cioé nomi che non occorrono né in $x_1, \cdots, x_n$ né in $\mathbb{V}$.] Allora

- se $1 \leq a_i \leq n$ allora $x'_i = x_{a_i}$;
- se $a_i > n$ allora $x'_i = z_{a_i}$;
- $\mathbb{V}' = \mathbb{V} \oplus x_1, \cdots, x_n{<}z_{i_1}, \cdots, z_{i_k}$.

La mutazione $\llbracket a_1, \cdots, a_n \rrbracket$ di $\left\langle \mathbb{V}, (x_1, \cdots, x_n) \right\rangle$ verrà scritta come

$$\left\langle \mathbb{V}, (x_1, \cdots, x_n) \right\rangle \overset{\llbracket a_1, \cdots, a_n \rrbracket}{\longrightarrow} \left\langle \mathbb{V}', (x'_1, \cdots, x'_n) \right\rangle$$

e l' etichetta $\llbracket a_1, \cdots, a_n \rrbracket$ verrà omessa quando la mutazione sarà chiara dal contesto.

Data la mutazione $\mu = \llbracket a_1, \cdots, a_n \rrbracket$, si definisce l' applicazione di μ ad un indice i, $1 \leq i \leq n$, come $\mu(i) = a_i$.

Le permutazioni sono mutazioni $\llbracket a_1, \cdots, a_n \rrbracket$ in cui gli elementi, a due a due differenti, appartengono all'insieme $\{1, 2, \cdots, n\}$ (ad esempio $\llbracket 2, 3, 5, 4, 1 \rrbracket$). Nelle permutazioni l' ordine parziale $\mathbb{V}$ non cambia mai, e dunque non serve. La terminologia usata di seguito e nell'articolo [1] é ispirata dalla terminologia corrispondente a quella delle permutazioni. Una mutazione differisce da una permutazione perché può contenere elementi ripetuti, o addirittura nuovi elementi (identificati da $n + 1 \leq a_i \leq 2 \times n$, per qualche a_i). Per esempio, applicando in successione la mutazione $\llbracket 2, 3, 6, 1, 1 \rrbracket$ a $\left\langle \mathbb{V}, (x_1, x_2, x_3, x_4, x_5) \right\rangle$, con $\mathbb{V} = \{(x_1, x_1), \cdots, (x_5, x_5)\}$ e $\widetilde{x} = x_1, x_2, x_3, x_4, x_5$, si ottiene

$$\begin{aligned}
\left\langle \mathbb{V}, (x_1, x_2, x_3, x_4, x_5) \right\rangle &\longrightarrow \left\langle \mathbb{V}_1, (x_2, x_3, y_1, x_1, x_1) \right\rangle \\
&\longrightarrow \left\langle \mathbb{V}_2, (x_3, y_1, y_2, x_2, x_2) \right\rangle \\
&\longrightarrow \left\langle \mathbb{V}_3, (y_1, y_2, y_3, x_3, x_3) \right\rangle \\
&\longrightarrow \left\langle \mathbb{V}_4, (y_2, y_3, y_4, y_1, y_1) \right\rangle \\
&\longrightarrow \cdots
\end{aligned}$$

dove $\mathbb{V}_1 = \mathbb{V} \oplus \widetilde{x} < y_1$ e, per $i \geq 1$, $\mathbb{V}_{i+1} = \mathbb{V}_i \oplus y_{i-1} < y_i$. Nell' esempio precedente, 6 identifica un nuovo nome che deve essere aggiunto ad ogni applicazione della mutazione. Ogni nome creato é un nome massimale per l'ordine parziale.

Si osservi come, da definizione, $\llbracket 2, 3, 6, 1, 1 \rrbracket$ e $\llbracket 2, 3, 7, 1, 1 \rrbracket$ comportano la stessa trasformazione sui nomi. In altre parole, la scelta di numeri naturali tra 6 e 10 é irrilevante nella definizione di mutazione. La medesima osservazione si può fare per le mutazioni $\llbracket 2, 3, 6, 1, 6 \rrbracket$ e $\llbracket 2, 3, 7, 1, 7 \rrbracket$.

Definizione 2.2. Sia $\llbracket a_1, \cdots, a_n \rrbracket \approx \llbracket a'_1, \cdots, a'_n \rrbracket$ se esiste una funzione biiettiva f da $[n + 1..2 \times n]$ a $[n + 1..2 \times n]$ tale che:

1. $1 \leq a_i \leq n$ implica $a'_i = a_i$;
2. $n + 1 \leq a_i \leq 2 \times n$ implica $a'_i = f(a_i)$.

Si noti come $\llbracket 2, 3, 6, 1, 1 \rrbracket \approx \llbracket 2, 3, 7, 1, 1 \rrbracket$ e $\llbracket 2, 3, 6, 1, 6 \rrbracket \approx \llbracket 2, 3, 7, 1, 7 \rrbracket$. Dàltra parte $\llbracket 2, 3, 6, 1, 6 \rrbracket \not\approx \llbracket 2, 3, 6, 1, 7 \rrbracket$; in pratica queste due mutazioni definiscono due differenti trasformazioni sui nomi.

Definizione 2.3. Dato un ordine parziale $\mathbb{V}$, un $\mathbb{V}$-*flashback* é una ridenominazione iniettiva ρ sui nomi tali che $\rho(x) \leq x \in \mathbb{V}$.

Nella sequenza precedente di mutazioni delle variabili $(x_1, x_2, x_3, x_4, x_5)$ esiste un $\mathbb{V}_4$-flashback da $(y_2, y_3, y_4, y_1, y_1)$ a $(x_2, x_3, y_1, x_1, x_1)$.

Nel caso di mutazioni che sono in realtà permutazioni, un flashback é semplicemente la ridenominazione identità.

Data una mutazione μ, si scriverà μ^m per l' applicazione di μ per m volte, scritto come $\left\langle \mathbb{V}, (x_1, \cdots, x_n) \right\rangle \xrightarrow{\mu^m} \left\langle \mathbb{V}', (y_1, \cdots, y_n) \right\rangle$ come abbreviazione di $\underbrace{\left\langle \mathbb{V}, (x_1, \cdots, x_n) \right\rangle \xrightarrow{\mu} \cdots \xrightarrow{\mu} \left\langle \mathbb{V}', (y_1, \cdots, y_n) \right\rangle}_{m \text{ volte}}$.

I flashback possono essere applicati anche alle tuple: $\rho(x_1, \cdots, x_n) \stackrel{def}{=} (\rho(x_1), \cdots, \rho(x_n))$.

Il Teorema 2.1.1 dimostrato in [1] generalizza la proprietà che ogni permutazione ha un *ordine*, che é il numero di applicazioni che ritorna la tupla iniziale. Nella teoria delle permutazioni, l'ordine é il minimo comune multiplo (abbreviato in *mcm* o *lcm* usando terminologia inglese) della lunghezza dei cicli della permutazione. Questo risultato é chiaramente falso per quanto riguarda le mutazioni, a causa della presenza di duplicazioni e dei nomi nuovi. La generalizzazione che vale nel contesto delle mutazioni fa uso di flashback al posto delle identità: si inizierà estendendo la nozione di ciclo.

Definizione 2.4 (Cicli, sink e orbite)**.** Sia $\mu = \langle\!\langle\, a_1, \cdots, a_n \,\rangle\!\rangle$ una mutazione e $1 \leq a_{i_1}, \cdots, a_{i_\ell} \leq n$ siano naturali a due a due differenti. Allora:

i. il termine $(a_{i_1} \cdots a_{i_\ell})$ é un *ciclo* di μ se $\mu(a_{i_j}) = a_{i_{j+1}}$, con $1 \leq j \leq \ell - 1$, e $\mu(a_{i_\ell}) = a_{i_1}$ (in altre parole, $(a_{i_1} \cdots a_{i_\ell})$ é lo stesso tipo di ciclo che si trova nelle permutazioni);

ii. il termine $[a_{i_1} \cdots a_{i_{\ell-1}}]_{a_{i_\ell}}$ é un *bound sink* di μ se $a_{i_1} \notin \{a_1, \cdots, a_n\}$, $\mu(a_{i_j}) = a_{i_{j+1}}$, con $1 \leq j \leq \ell - 1$, e a_{i_ℓ} appartiene ad un ciclo;

iii. il termine $[a_{i_1} \cdots a_{i_\ell}]_a$, con $n < a \leq 2 \times n$, é un *free sink* di μ se $a_{i_1} \notin \{a_1, \cdots, a_n\}$ e $\mu(a_{i_j}) = a_{i_{j+1}}$, con $1 \leq j \leq \ell - 1$ e $\mu(a_{i_\ell}) = a$.

La *lunghezza di un ciclo* é il numero di elementi nel ciclo stesso; invece la *lunghezza di un sink* é il numero di elementi all'interno delle parentesi quadre.

Per esempio la mutazione $\langle\!\langle\, 5, 4, 8, 8, 3, 5, 8, 3, 3 \,\rangle\!\rangle$ ha come cicli $(3, 8)$ e ha i bound sink $[1, 5]_3$, $[6, 5]_3$, $[9]_3$, $[2, 4]_8$ e $[7]_8$. La mutazione $\langle\!\langle\, 6, 3, 1, 8, 7, 1, 8 \,\rangle\!\rangle$ ha un unico ciclo, $(1, 6)$, un unico bound sink $[2, 3]_1$ e due free sinks $[4]_8$ e $[5, 7]_8$.

I cicli e i sink offrono una descrizione alternativa di una mutazione. Ad esempio, $(3, 8)$ significa che la mutazione muove l'elemento in posizione 8 all'elemento in posizione 3, e l'elemento in posizione 3 alla posizione 8; Il free sink $[5, 7]_8$ indica che l'elemento in posizione 7 va in posizione 5, mentre si mette un nome nuovo in posizione 7.

Teorema 2.1.1. *Sia μ una mutazione, ℓ sia il mcm della lunghezza dei suoi cicli, ℓ' sia il più lungo bound sink in μ e ℓ'' sia il più lungo free sink in μ. Sia anche $k \stackrel{def}{=} \max\{\ell + \ell',\ \ell''\}$. Allora esiste $0 \leq h < k$ tale che $\left\langle \mathbb{V}, (x_1, \cdots, x_n) \right\rangle \xrightarrow{\mu^h} \left\langle \mathbb{V}', (y_1, \cdots, y_n) \right\rangle \xrightarrow{\mu^{k-h}} \left\langle \mathbb{V}'', (z_1, \cdots, z_n) \right\rangle$ e $\rho(z_1, \cdots, z_n) = (y_1, \cdots, y_n)$, per qualche $\mathbb{V}''$-flashback ρ. Il valore k é chiamato* ordine di μ *e identificato con $\mathfrak{o}_\mu$.*

Si prenda ad esempio $\mu = \langle\!\langle\, 6, 3, 1, 8, 7, 1, 8 \,\rangle\!\rangle$, la quale ha un ciclo $(1, 6)$, un bound sink $[2, 3]_1$ e free sink $[4]_8$ e $[5, 7]_8$. Dunque $\ell = 2$, $\ell' = 2$ e $\ell'' = 2$. In questo caso, i valori k e h del Teorema 2.1.1 sono 4 e 2, rispettivamente. In effetti, applicando la mutazione μ quattro volte alla coppia $\left\langle \mathbb{V}, (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \right\rangle$, dove $\mathbb{V} = \{(x_i, x_i) \mid 1 \leq i \leq 7\}$ si ottiene:

$$\begin{aligned}
&\left\langle \mathbb{V}, (x_1, x_2, x_3, x_4, x_5, x_6, x_7) \right\rangle \\
&\qquad \xrightarrow{\mu} \left\langle \mathbb{V}_1, (x_6, x_3, x_1, y_1, x_7, x_1, y_1) \right\rangle \\
&\qquad \xrightarrow{\mu} \left\langle \mathbb{V}_2, (x_1, x_1, x_6, y_2, y_1, x_6, y_2) \right\rangle \\
&\qquad \xrightarrow{\mu} \left\langle \mathbb{V}_3, (x_6, x_6, x_1, y_3, y_2, x_1, y_3) \right\rangle \\
&\qquad \xrightarrow{\mu} \left\langle \mathbb{V}_4, (x_1, x_1, x_6, y_4, y_3, x_6, y_4) \right\rangle.
\end{aligned}$$

dove $\mathbb{V}_1 = \mathbb{V} \oplus x_1, x_2, x_3, x_4, x_5, x_6, x_7 < y_1$ e, per $i \geq 1$, $\mathbb{V}_{i+1} = \mathbb{V}_i \oplus y_{i-1} < y_i$. Si noti come esista un $\mathbb{V}_4$-flashback ρ da $(x_1, x_1, x_6, y_4, y_3, x_6, y_4)$ (prodotto da μ^4) a $(x_1, x_1, x_6, y_2, y_1,\ x_6, y_2)$ (prodotto da μ^2).

Per meglio comprendere gli esempi seguenti, oltre alla rappresentazione testuale di una mutazione si indicherà anche la relativa rappresentazione grafica, costituita da un grafo in cui i nodi corrispondono alle posizioni all'interno della mutazione stessa, mentre l'arco che arriva al nodo i parte dal nodo referenziato all'interno del nodo stesso. Per come sono costruite le mutazioni, ogni nodo avrÃă al più un solo arco entrante, ma potrÃă avere diversi archi uscenti.

Esempio 2.1.2. *Si consideri la mutazione $\mu_1 = \langle\!\langle\, 6, 3, 5, 4, 4, 7, 1 \,\rangle\!\rangle$. Essa ha cicli in $(1, 6, 7)$ e (4), bound sink $[2, 3, 5]_4$. Si possono ricavare dunque i seguenti valori $\ell = 3$, $\ell' = max\{m_2\} = 3$, e quindi si ottiene l'ordine della mutazione, che é $\mathfrak{o}_{\mu_1} = \ell + \ell' = 6$. Ciò implica che dopo tre applicazioni della*

mutazione, gli elementi 2, 3, *e* 5 *scompaiono e* $\mu_1^3(2) = \mu_1^3(3) = \mu_1^3(5) = 4$. *Dopo altre* $mcm(3,1) = 3$ *applicazioni si ritornerà nello stato cosiddetto* stabile. *Da qui si vede chiaramente come, l'ordine di* μ_1 *é dato dai tre passi compiuti per raggiungere lo stato* stabile *più il mcm della lunghezza dei cicli:* $3 + 3 = 6$.

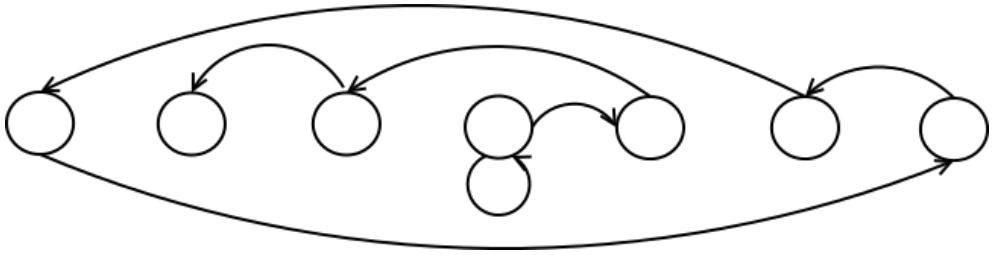


Figura 2.1: Grafico della mutazione dell'esempio 2.1.2

Esempio 2.1.3. *La mutazione* $\mu_2 = (6,3,4,4,7,1,8)$ *ha invece cicli in* $(1,6)$ *e* (4), *ed ha un bound sink in* $[2,3]_4$, *e free sink* $[5,7]_8$. *Si hanno perciò* $\ell = 2$, $\ell' = 2$, $\ell'' = 2$, *e* $\mathfrak{o}_\mu = max\{\ell + \ell', \ell''\} = 4$. *Dopo due applicazioni della mutazione le componenti* 2 *e* 3 *scompaiono, e* $\mu_2(2) = \mu_2(3) = 4$. *Applicando ulteriormente la mutazione per due volte,* $\mu_2(5) = 8$ *e* $\mu_2(7) = 9$. *Da qui, ad ogni passo é introdotta una nuova variabile ma* 2, 3, 5, *e* 7 *non compariranno più. La mutazione ha quindi raggiunto lo stato* stabile s. *Dopo* $mcm(2,1) = 2$ *ulteriori applicazioni verrà raggiunto lo stato* stabile s' *tale che esista un flashback* ρ *il quale consente che valga* $\rho(s') = s$. *Ricapitolando, l'ordine di* μ_2 *é dato dai due passi necessari per raggiungere lo stato* stabile s, *più gli ulteriori due passi per raggiungere* s': $2 + 2 = 4$.

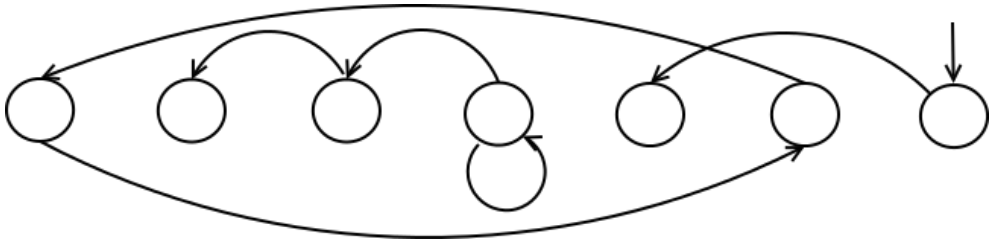


Figura 2.2: Grafico della mutazione dell' Esempio 2.1.3

Esempio 2.1.4. *Si consideri per un ulteriore esempio la mutazione* $\mu_2 = (2,1,4,5,6,7,8)$. *Essa presenta un ciclo* $(1,2)$ *e un free sink* $[3,4,5,6,7]_8$. *Si può quindi asserire che* $\ell = 2$, $\ell'' = max\{m_2\} = 5$, *e* $\mathfrak{o}_{\mu_2} = max\{\ell, \ell''\} = 5$. *Ciò implica che dopo cinque applicazioni della mutazione, le componenti* 3,

4, 5, 6, *e* 7 *scompariranno e* $\mu_3(3) = 8$, $\mu_3(4) = 9$, $\mu_3(5) = 10$, $\mu_3(6) = 11$, $\mu_3(7) = 12$. *Da questo stato una nuova variabile verrà introdotta ad ogni passo, e allo stesso tempo* 3, 4, 5, 6, *e* 7 *scompariranno definitivamente. La mutazione raggiungerà dunque uno stato* stabile s. *Curiosamente, essa produrrà uno stato* stabile s' *ad ogni passo, tale che esista un flashback* ρ *tale che* $\rho(s') = s$.

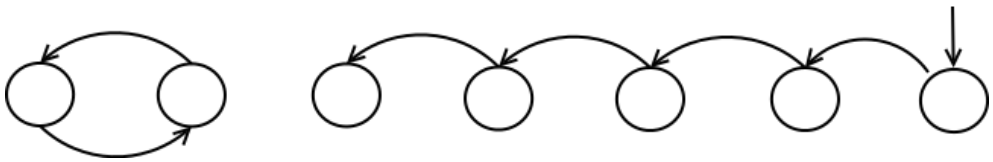


Figura 2.3: Grafico della mutazione dell' Esempio 2.1.4

Esempio 2.1.5. *Come ultimo esempio si veda* $\mu_3 = (2, 8, 4, 3, 9, 7, 2)$. *Ha un ciclo* $(3, 4)$ *e diversi free sinks,* $[1, 2]_8$, $[5]_9$, $[6, 7, 2]_8$. *Risulta quindi* $\ell = 2$, $\ell'' = max\{m_1, m_5, m_6\} = max\{2, 1, 3\} = 3$, *e* $\mathfrak{o}_{\mu_3} = max\{\ell, \ell''\} = 3$.

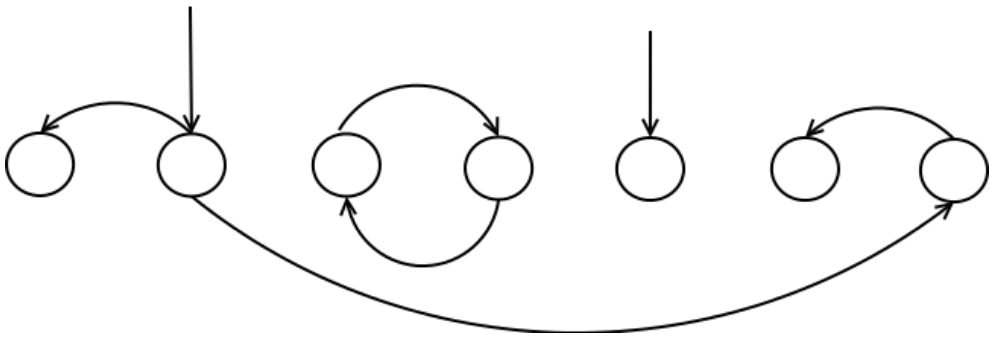


Figura 2.4: Grafico della mutazione dell' Esempio 2.1.5

2.2 Modelli LAM

Nella sezione appena iniziata si analizzaranno i modelli su cui si basa la tecnica per l'analisi delle circolarità. Essi sono chiamati *LAM*, un acronimo per *deadLock Analysis Model*, e sono termini che rappresentano relazioni sui nomi, le quali definiscono l'insieme di dipendenze tra risorse.

I modelli LAM sono definiti da programmi che usano, oltre all'insieme di *nomi* V della sezione sulle mutazioni, un insieme infinito di *nomi di funzione*, che spaziano su $\mathtt{f}$, $\mathtt{f}'$, $\mathtt{g}$, $\mathtt{g}'$, $\cdots$. Un *programma LAM* é una tupla $\left(\mathtt{f}_1(\widetilde{x_1}) =\right.$

$\mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\Big)$ dove $\mathtt{f}_i(\widetilde{x_i}) = \mathtt{L}_i$ sono *definizioni di funzione* e $\mathtt{L}$ é il *LAM principale*. La sintassi di $\mathtt{L}_i$ e $\mathtt{L}$ é la seguente:

$$\mathtt{L} \quad ::= \quad \mathtt{0} \quad | \quad (x, y) \quad | \quad \mathtt{f}(\widetilde{x}) \quad | \quad \mathtt{L} \| \mathtt{L} \quad | \quad \mathtt{L}\,;\,\mathtt{L}$$

Quando le parentesi sono omesse, l'operazione "$\|$" ha precedenza su "$\,;\,$". Si abbrevierà la scrittura $\mathtt{L}_1 \| \cdots \| \mathtt{L}_n$ in $\prod_{i \in 1..n} \mathtt{L}_i$ e si useranno le operazioni sugli insiemi "$\in$" e "$\setminus$" su $\mathtt{L}$. Nello specifico, si scriverà $\mathtt{f}(\widetilde{u}) \in \mathtt{L}$ quando é presente una occorrenza di $\mathtt{f}(\widetilde{u})$ in $\mathtt{L}$, e $\mathtt{L} \setminus \{\mathtt{f}(\widetilde{u})\}$ con l'ovvio significato di eliminare $\mathtt{f}(\widetilde{u})$ da $\mathtt{L}$. Allo stesso modo si scriverà $\mathtt{f}(\widetilde{u}) \in \mathtt{L}$ quando esiste $\mathtt{T} \in \mathtt{L}$ tale che $\mathtt{f}(\widetilde{u}) \in \mathtt{T}$, e anche $(x, y) \in \mathtt{L}$. Si userà $\mathtt{T}$ per spaziare su LAM che non contengono invocazioni di funzioni.

Sia $var(\mathtt{L})$ l'insieme dei nomi in $\mathtt{L}$. In una definizione di funzione $\mathtt{f}(\widetilde{x}) = \mathtt{L}$, $\widetilde{x}$ sono i *parametri formali* e le occorrenze dei nomi $x \in \widetilde{x}$ in $\mathtt{L}$ sono *legate*. Tutti gli altri nomi in $\mathtt{L}$, e specificatamente $var(\mathtt{L}) \setminus \widetilde{x}$, sono *liberi*.

Nella sintassi di $\mathtt{L}$, le operazioni "$\|$" e "$\,;\,$" sono associative, commutative con $\mathtt{0}$ identità. Oltre a ciò valgono i seguenti assiomi (con $\mathtt{T}$ che non contiene invocazioni di funzione):

$$\mathtt{T} \| \mathtt{T} = \mathtt{T} \qquad \mathtt{T}\,;\,\mathtt{T} = \mathtt{T} \qquad \mathtt{T} \| (\mathtt{L}'\,;\,\mathtt{L}'') = \mathtt{T} \| \mathtt{L}'\,;\,\mathtt{T} \| \mathtt{L}''$$

Questi assiomi permettono di riscrivere un LAM senza invocazioni di funzione come una sequenza di paralleli, in cui ognuno di essi rappresenta un *insieme di relazioni*.

Proposizione 2.2.1. *Per ogni* $\mathtt{T}$*, si ha* $\mathtt{T} = \mathtt{T}_1\,;\,\cdots\,;\,\mathtt{T}_n$*, dove* $\mathtt{T}_i$ *sono paralleli tra dipendenze.*

Osservazione 2.1. *Nei modelli LAM, l'operazione "*$\,;\,$*" é commutativa e può sembrare controintuitivo se lo si paragona alla composizione sequenziale dei linguaggi di programmazione, che di solito é indicata allo stesso modo. La commutatività di "*$\,;\,$*" é dovuta al fatto che i LAM sono* modelli astratti *dei linguaggi di programmazione che specificano le dipendenze tra risorse negli* stati a tempo di esecuzione *dei programmi. Se un programma é una composizione sequenziale, ad esempio* $\mathtt{Stm}_1\,;\,\mathtt{Stm}_2$*, e* $\mathtt{L}_1$ *and* $\mathtt{L}_2$ *definiscono rispettivamente le dipendenze di* $\mathtt{Stm}_1$ *e* $\mathtt{Stm}_2$*, allora le dipendenze di* $\mathtt{Stm}_1\,;\,\mathtt{Stm}_2$ *sono quelle di* $\mathtt{L}_1$ e *quelle di* $\mathtt{L}_2$*. In altre parole, usando la notazione sopra riportata, sono* $\mathtt{L}_1\,;\,\mathtt{L}_2$ *oppure* $\mathtt{L}_2\,;\,\mathtt{L}_1$*, che sono dunque uguali.*

D'ora in avanti, si assumerà che i programmi LAM $\Big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\Big)$ siano sempre *ben definiti*, e cioé *(1)* tutti i nomi di funzione che occorrono in $\mathtt{L}_i$ e $\mathtt{L}$ sono definiti; *(2)* l'arietà delle invocazioni corrispe a quella delle definizioni delle funzioni stesse.

Semantica operazionale. Sia un *contesto LAM*, scritto come $\mathfrak{L}[\ ]$, un termine derivato dalla seguente sintassi:

$$\mathfrak{L}[\ ] \quad ::= \quad [\ ] \qquad | \qquad \mathtt{L} \| \mathfrak{L}[\ ] \qquad | \qquad \mathtt{L}\,;\, \mathfrak{L}[\ ]$$

Come solito, $\mathfrak{L}[\mathtt{L}]$ é il LAM dove i posti vuoti di $\mathfrak{L}[\ ]$ sono rimpiazzati con $\mathtt{L}$. La semantica operazionale di un programma $\Big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}_{\ell+1}\Big)$ é un sistema di transizioni dove gli *stati* sono coppie $\Big\langle \mathbb{V},\ \mathtt{L} \Big\rangle$ e la *relazione di transizione* é la più piccola che soddisfa la regola:

$$\frac{\begin{array}{c}\text{(RED)}\\ \mathtt{f}(\widetilde{x}) = \mathtt{L} \qquad var(\mathtt{L}) \setminus \widetilde{x} = \widetilde{z} \qquad \widetilde{w} \text{ sono fresh} \\ \mathtt{L}[\widetilde{w}/\widetilde{z}][\widetilde{u}/\widetilde{x}] = \mathtt{L}'\end{array}}{\Big\langle \mathbb{V},\ \mathfrak{L}[\mathtt{f}(\widetilde{u})] \Big\rangle \longrightarrow \Big\langle \mathbb{V} \oplus \widetilde{u}_{<\widetilde{w}},\ \mathfrak{L}[\mathtt{L}'] \Big\rangle}$$

Per la regola (RED), un LAM $\mathtt{L}$ é valutato rimpiazzando in successione le invocazioni di funzione con le corrispondenti istanze LAM. Le creazioni dei nomi sono gestite tramite un meccanismo simile a quello delle mutazioni. Per esempio, se $\mathtt{f}(x) = (x, y) \| \mathtt{f}(y)$ e $\mathtt{f}(u)$ occorrono nel LAM principale, allora $\mathtt{f}(u)$ é rimpiazzato da $(u, v) \| \mathtt{f}(v)$, dove v é un *nome fresh massimale* in qualche ordine parziale.

Lo stato iniziale di un programma con LAM principale $\mathtt{L}$ é $\Big\langle \mathbb{I}_{\mathtt{L}},\ \mathtt{L} \Big\rangle$, dove $\mathbb{I}_{\mathtt{L}} \stackrel{def}{=} \{(x, x) \mid x \in var(\mathtt{L})\}$.

Esempio 2.2.2. *Si consideri il programma:*

$$\Bigg(\begin{array}{l} \mathtt{l(x, y, z, w, u)} = \mathtt{f(y, z, x, w, u, v, t)}, \\ \mathtt{f(x', y', z', w', u', v', t')} = \mathtt{g(w', x')} \| \mathtt{l(y', z', x', v', t')} \| \mathtt{(z', t')} \| \mathtt{(t', v')}, \\ \mathtt{g(x'', y'')} = \mathtt{(x'', y'')} \| \mathtt{g(x'', y'')}, \\ \mathtt{(x_1, x_2)} \| \mathtt{l(x_0, x_1, x_2, x_3, x_4)} \end{array} \Bigg)$$

Una possibile valutazione del LAM principale é la seguente:

$$\begin{array}{ll} \Big(\mathbb{V}, \mathtt{(x_1, x_2)} \| \mathtt{l(x_0, x_1, x_2, x_3, x_4)}\Big) & \longrightarrow \\ (\mathbb{V}_1, \mathtt{(x_1, x_2)} \| \mathtt{f(x_1, x_2, x_0, x_3, x_4, x_5, x_6)}) & \longrightarrow \\ (\mathbb{V}_1, \mathtt{(x_1, x_2)} \| \mathtt{(x_0, x_6)} \| \mathtt{(x_6, x_5)} \| \mathtt{g(x_3, x_1)} \| \mathtt{l(x_2, x_0, x_1, x_5, x_6)}) & \longrightarrow \\ (\mathbb{V}_1, \mathtt{(x_1, x_2)} \| \mathtt{(x_3, x_1)} \| \mathtt{(x_0, x_6)} \| \mathtt{(x_6, x_5)} \| \mathtt{g(x_3, x_1)} \| \mathtt{l(x_2, x_0, x_1, x_5, x_6)}) & \longrightarrow \\ \ldots & \end{array}$$

dove $\mathbb{V}_1$ é l'unione di $\mathbb{V}$ e un nuovo insieme, dove i nomi fresh sono messi in relazione con i nomi già esistenti.

Per illustrare la semantica del linguaggio dei LAM si propongono tre esempi:

1. $\Big(\mathtt{f}(x,y,z) = (x,y) \| \mathtt{g}(y,z)\,;\,(y,z),\ \mathtt{g}(u,v) = (u,v)\,;\,(v,u),\ \mathtt{f}(x,y,z)\Big)$ e $\mathbb{I} = \{(x,x),(y,y),(z,z)\}$. Allora

$$\begin{aligned}\Big\langle \mathbb{I},\ \mathtt{f}(x,y,z)\Big\rangle &\longrightarrow \Big\langle \mathbb{I},\ (x,y)\|\mathtt{g}(y,z)\,;\,(y,z)\Big\rangle \\ &\longrightarrow \Big\langle \mathbb{I},\ (x,y)\|(y,z);\ (x,y)\|(z,y)\,;\,(y,z)\Big\rangle\end{aligned}$$

   Il LAM nello stato finale *non contiene invocazioni di funzione*. Questo avviene a causa del fatto che il programma sopra riportato non é ricorsivo. In aggiunta, la valutazione di $\mathtt{f}(x,y,z)$ *non ha creato nomi*, dato che i tutti i nomi all'interno delle definizioni di $\mathtt{f}(x,y,z)$ e $\mathtt{g}(u,v)$ sono legati.

2. $\Big(\mathtt{f'}(x) = (x,y)\|\mathtt{f'}(y)\ ,\ \mathtt{f'}(x)\Big)$ e $\mathbb{V}_0 = \{(x_0,x_0)\}$. Allora

$$\begin{aligned}&\Big\langle \mathbb{V}_0,\ \mathtt{f'}(x_0)\Big\rangle \\ &\longrightarrow \Big\langle \mathbb{V}_1,\ (x,x_1)\|\mathtt{f'}(x_1)\Big\rangle \\ &\longrightarrow \Big\langle \mathbb{V}_2,\ (x,x_1)\|(x_1,x_2)\|\mathtt{f'}(x_2)\Big\rangle \\ &\longrightarrow^n \Big\langle \mathbb{V}_{n+2},\ (x,x_1)\|(x_1,x_2)\|\cdots\|(x_{n+1},x_{n+2})\|\mathtt{f'}(x_{n+2})\Big\rangle\end{aligned}$$

   dove $\mathbb{V}_{i+1} = \mathbb{V}_i \oplus x_i{<}x_{i+1}$. Nel caso sopra riportato, il numero di paralleli cresce man mano che la valutazione progredisce. Ciò avviene *per la presenza di nomi liberi* nella definizione di $\mathtt{f'}$ che, come già notato, corrispondono alla generazione di nomi fresh ad ogni invocazione ricorsiva.

3. $\Big(\mathtt{g'}(x) = (x,x')\,;\,(x,x')\|\mathtt{g'}(x'),\ \mathtt{g'}(x_0)\Big)$ e $\mathbb{V}_0 = \{(x_0,x_0)\}$. Allora

$$\begin{aligned}&\Big\langle \mathbb{V}_0,\ \mathtt{g'}(x_0)\Big\rangle \\ &\quad\longrightarrow \Big\langle \mathbb{V}_1, (x_0,x_1)\,;\,(x_0,x_1)\|\mathtt{g'}(x_1)\Big\rangle \\ &\quad\longrightarrow \Big\langle \mathbb{V}_2, (x_0,x_1)\,;\,(x_0,x_1)\|(x_1,x_2)\,; \\ &\qquad\qquad (x_0,x_1)\|(x_1,x_2)\|\mathtt{g'}(x_2)\Big\rangle \\ &\quad\longrightarrow^n \Big\langle \mathbb{V}_{n+2}, (x_0,x_1)\,;\,\cdots\,; \\ &\qquad\qquad ((x_0,x_1)\|\cdots\|(x_{n+1},x_{n+2})\|\mathtt{g'}(x_{n+2})\Big\rangle\end{aligned}$$

   Dove $\mathbb{V}_{i+1}$ sono gli stessi dell'esempio precedente. In questo caso, gli elementi a crescere di pari passo con il progresso della valutazione sono gli elementi in sequenza.

La semantica del linguaggio dei LAM é non-deterministica, a causa della scelta dell'invocazione da valutare. Nonostante ciò, nell'articolo originale [1] si é dimostrata l'esistenza di una proprietà a diamante dello stesso, *al più con una ridenominazione biiettiva dei nomi (liberi).*

La semantica operazionale informata. Per controllare la presenza di circolarità, la tecnica prevede di valutare un LAM fino a quando ogni funzione ivi contenuta sia stata adeguatamente espansa (fino all'ordine della mutazione associata). Ciò viene formalizzato passando ad una semantica operazionale "informata", dove i termini sono etichettati con le *storie*.

Definizione 2.5. Sia una *storia*, o *history*, definita su $\alpha, \beta, \cdots$, una sequenza di nomi di funzione $\mathtt{f}_{i_1}\mathtt{f}_{i_2}\cdots\mathtt{f}_{i_n}$. Si scrive $\mathtt{f} \in \alpha$ se $\mathtt{f}$ occorre α. Inoltre si scriverà α^n per $\underbrace{\alpha\cdots\alpha}_{n \text{ volte}}$. Sia $\alpha \preceq \beta$ se esiste α' tale che $\alpha\alpha' = \beta$. Il simbolo ε denota la storia vuota.

La semantica operazionale informata é un sistema di transizione dove gli stati sono tuple $\left\langle \mathbb{V},\, {}^{\natural}\mathbb{F},\, \mathbb{L} \right\rangle$ dove ${}^{\natural}\mathbb{F}$ é un insieme di invocazioni di funzione con storie e $\mathbb{L}$, chiamato *LAM informativo*, é un termine come $\mathtt{L}$, tranne per il fatto che le coppie e le invocazioni di funzione sono indicizzate attraverso le storie, ad esempio ${}^{\alpha}(x, y)$ e ${}^{\alpha}\mathtt{f}(\widetilde{u})$, rispettivamente.

Sia

$$addh(\alpha, \mathtt{L}) \stackrel{def}{=} \begin{cases} {}^{\alpha}(x, y) & \text{se } \mathtt{L} = (x, y) \\ {}^{\alpha}\mathtt{f}(\widetilde{x}) & \text{se } \mathtt{L} = \mathtt{f}(\widetilde{x}) \\ addh(\alpha, \mathtt{L}') \| addh(\alpha, \mathtt{L}'') & \text{se } \mathtt{L} = \mathtt{L}' \| \mathtt{L}'' \\ addh(\alpha, \mathtt{L}')\,;\, addh(\alpha, \mathtt{L}'') & \text{se } \mathtt{L} = \mathtt{L}'\,;\, \mathtt{L}'' \end{cases}$$

Per esempio $addh(\mathtt{fl}, (x_4, x_2) \| \mathtt{f}(x_2, x_3, x_4, x_5)) = {}^{\mathtt{fl}}(x_4, x_2) \| \, {}^{\mathtt{fl}}\mathtt{f}(x_2, x_3, x_4, x_5)$. La relazione di transizione informativa é la più piccola tale che

$$\frac{\begin{array}{c}\text{(Red+)} \\ \mathtt{f}(\widetilde{x}) = \mathtt{L} \qquad var(\mathtt{L}) \setminus \widetilde{x} = \widetilde{z} \qquad \widetilde{w} \text{ sono fresh} \\ \mathtt{L}[\widetilde{w}/\widetilde{z}][\widetilde{u}/\widetilde{x}] = \mathtt{L}'\end{array}}{\begin{array}{l}\left\langle \mathbb{V},\, {}^{\natural}\mathbb{F},\, {}^{\natural}\mathfrak{L}[{}^{\alpha}\mathtt{f}(\widetilde{u})] \right\rangle \quad \longrightarrow \\ \qquad \left\langle \mathbb{V} \oplus \widetilde{u} {<} \widetilde{w},\; {}^{\natural}\mathbb{F} \cup \{{}^{\alpha}\mathtt{f}(\widetilde{u})\},\; {}^{\natural}\mathfrak{L}[addh(\alpha\mathtt{f}, \mathtt{L}')] \right\rangle\end{array}}$$

dove ${}^{\natural}\mathfrak{L}[\ ]$ é un contesto LAM con storie (le coppie di dipendenze e le invocazioni di funzione sono etichettate con le storie). Quando $\left\langle \mathbb{V},\, {}^{\natural}\mathbb{F},\, \mathbb{L} \right\rangle \longrightarrow$

$\left\langle \mathbb{V}', {}^{\natural}\mathbb{F}', \mathbb{L}' \right\rangle$ tramite l'applicazione di (RED+) a ${}^{\alpha}\mathtt{f}(\widetilde{u})$, si dice che il termine ${}^{\alpha}\mathtt{f}(\widetilde{u})$ *é valutato nella riduzione*. Lo stato iniziale informativo di un programma con LAM principale $\mathtt{L}$ é $\left\langle \mathbb{I}_{\mathtt{L}}, \varnothing, \mathit{addh}(\varepsilon, \mathtt{L}) \right\rangle$.

Per esempio, il programma `flh`

$$\left(\begin{array}{lcl} \mathtt{f}(x,y,z,u) & = & (x,z) \| \mathtt{l}(u,y,z)\ , \\ \mathtt{l}(x,y,z) & = & (x,y) \| \mathtt{f}(y,z,x,u)\ , \\ \mathtt{h}(x,y,z,u) & = & (z,x) \| \mathtt{h}(x,y,z,u) \| \mathtt{f}(x,y,z,u)\ , \\ \mathtt{h}(x_1,x_2,x_3,x_4) & & \end{array}\right)$$

ha una valutazione (informativa)

$$\begin{array}{l} \left\langle \mathbb{I}_{\mathtt{L}}, \varnothing, {}^{\varepsilon}\mathtt{h}(x_1,x_2,x_3,x_4) \right\rangle \\ \longrightarrow \left\langle \mathbb{I}_{\mathtt{L}}, {}^{\natural}\mathbb{F}, \mathbb{L} \,\|\, {}^{\mathtt{h}}\mathtt{f}(x_1,x_2,x_3,x_4) \right\rangle \\ \longrightarrow \left\langle \mathbb{I}_{\mathtt{L}}, {}^{\natural}\mathbb{F}_1, \mathbb{L} \,\|\, {}^{\mathtt{hf}}(x_1,x_3) \,\|\, {}^{\mathtt{hf}}\mathtt{l}(x_4,x_2,x_3) \right\rangle \\ \longrightarrow \left\langle \mathbb{I}_{\mathtt{L}} \oplus x_4{<}x_5, {}^{\natural}\mathbb{F}_2, \mathbb{L}' \,\|\, {}^{\mathtt{hfl}}(x_4,x_2) \,\|\, {}^{\mathtt{hfl}}\mathtt{f}(x_2,x_3,x_4,x_5) \right\rangle \end{array}$$

dove $\mathbb{L} = {}^{\mathtt{h}}(x_3,x_1) \| {}^{\mathtt{h}}\mathtt{h}(x_1,x_2,x_3,x_4)$, $\mathbb{L}' = \mathbb{L} \| {}^{\mathtt{hf}}(x_1,x_3)$ e ${}^{\natural}\mathbb{F} = \{{}^{\varepsilon}\mathtt{h}(x_1,x_2,x_3,x_4)\}$, ${}^{\natural}\mathbb{F}_1 = {}^{\natural}\mathbb{F} \cup \{{}^{\mathtt{h}}\mathtt{f}(x_1,x_2,x_3,x_4)\}$, ${}^{\natural}\mathbb{F}_2 = {}^{\natural}\mathbb{F}_1 \cup \{{}^{\mathtt{hf}}\mathtt{l}(x_4,x_2,x_3)\}$.

La semantica informativa e non informativa di un programma sono in stretta relazione, fatto cruciale per la per correttezza formale descritta nel lavoro [1]. La corrispondenza formale non é nient'altro che un'operazione di cancellazione $[\![\cdot]\!]$ che rimuove le storie da un LAM informativo.

Circolarità. I LAM registrano insiemi di relazioni sui nomi. Per esplicitare le relazioni definite dai LAM, si usa la funzione $\flat(\cdot)$, chiamata *flattening*, che é definita induttivamente come segue

$$\begin{array}{c} \flat(\mathtt{0}) = \mathtt{0}, \qquad \flat((x,y)) = (x,y), \qquad \flat(\mathtt{f}(\widetilde{x})) = \mathtt{0}, \\ \flat(\mathtt{L} \| \mathtt{L}') = \flat(\mathtt{L}) \| \flat(\mathtt{L}'), \qquad \flat(\mathtt{L}\,;\,\mathtt{L}') = \flat(\mathtt{L})\,;\,\flat(\mathtt{L}'). \end{array}$$

Per esempio

$$\begin{array}{l} \mathtt{L} = \mathtt{f}(x,y,z)\,;\,(x,y) \| \mathtt{g}(y,z) \| \mathtt{f}(u,y,z)\,;\,\mathtt{g}(u,v) \| (u,v)\,;\,(v,u) \\ \flat(\mathtt{L}) = (x,y)\,;\,(u,v)\,;\,(v,u) \end{array}$$

cioé, ci sono tre relazioni in $\mathtt{L}$: $\{(x,y)\}$ e $\{(u,v)\}$ e $\{(v,u)\}$. Per la Proposizione 2.2.1, $\flat(\mathtt{L})$ ritorna, a meno degli assiomi sui LAM, sequenze di paralleli di dipendenze.

L'operazione $\flat(\cdot)$ può essere estesa ai LAM informativi $\mathbb{L}$ in maniera semplice:

$$\flat({}^{\natural}(x,y)) = (x,y), \qquad \flat({}^{\natural}\mathtt{f}(\widetilde{x})) = \mathtt{0}.$$

Definizione 2.6. Un LAM $\mathtt{L}$ *ha una circolarità* se

$$\flat(\mathtt{L}) = (x_1, x_2)\|(x_2, x_3)\| \cdots \|(x_m, x_1)\|\mathtt{T}'; \mathtt{T}''$$

per qualche $x_1, \cdots, x_m$. Uno stato $\left\langle \mathbb{V},\ \mathtt{L} \right\rangle$ ha una circolarità se $\mathtt{L}$ ha una circolarità. La definizione si applica con le modifiche ovvie anche ai LAM informativi $\mathbb{L}$.

Lo stato finale della computazione del programma `fgh` ha una circolarità. Altre funzioni che hanno circolarità sono le funzioni `f` e `g` del Capitolo 1. Nessuno degli stati negli esempi 1,2, 3 all'inizio di questa sezione ha una *circolarità*.

Capitolo 3

Individuare i deadlock in LAM lineari

Dato un modello LAM, realizzato con il linguaggio del capitolo precedente, è possibile individuare una computazione che manifesti un lock. In questo capitolo si analizzerà un metodo basato sull'interpretazione delle funzioni ricorsive e/o mutuamente ricorsive come *mutazioni*. Tale interpretazione è possibile solo nel caso di programmi *lineari*, nei quali tutte le funzioni hanno un'unica mutazione associata. L'algoritmo è il seguente:

1. Si valutino le invocazioni necessarie al raggiungimento di un cosiddetto *stato saturato*;

2. Se non c'è un lock nello *stato saturato*, allora il programma non manifesterà mai un lock (altrimenti, la computazione fino ad arrivare allo stato saturato mostra l'evidenza di un lock)

Informalmente, "saturazione" significa (1) calcolare tutti i cammini possibili per raggiungere una invocazione, ad esempio $\mathtt{f}(\widetilde{u})$, da un LAM iniziale, e da questi (2) valutare le funzioni (possono essere molteplici a causa della mutua ricorsione) che partecipano alla mutazione di $\mathtt{f}(\widetilde{u})$ stessa. La presenza di un lock è decisa tramite un analisi di circolarità: se nello stato saturato vi è una circolarità di dipendenze, allora nello stato c'è un lock.

Si restringe il campo di azione, senza perdere di generalità, ai modelli LAM mutuamente ricorsivi. In effetti, l'analisi di circolarità all'interno di programmi non ricorsivi è banale: è sufficiente valutare tutte le invocazioni fino allo stato finale, e poi valutare la presenza di circolarità in esso. In particolare si assume che ogni funzione sia (mutuamente) ricorsiva. Ci si può ricondurre facilmente a questo caso espandendo tutte le invocazioni delle funzioni non (mutuamente) ricorsive e rimuovendo le loro definizioni dal modello LAM.

3.1 Linearità

Si definisce *programma lineare* qualsiasi programma che rispetti la seguente definizione:

Definizione 3.1. *Un programma* $\left(\mathtt{f}(\widetilde{x_1}) = \mathtt{L}_1, \ldots, \mathtt{f}_l(\widetilde{x}_l) = \mathtt{L}_l, \mathtt{L}\right)$ *è* lineare *se, per ogni* $\mathtt{f}_i$ *, esiste una unica sequenza di nomi di funzione, a due a due differenti* $\mathtt{f}_i\mathtt{f}_{i_1} \ldots \mathtt{f}_{i_k}$ *tale che:*

1. $\mathtt{L}_i$ *contenga esattamente una invocazione di* $\mathtt{f}_{i_1}$*;*

2. $\mathtt{L}_{i_j}$*, con* $j \leq k-1$*, contenga esattamente una invocazione di* $\mathtt{f}_{i_{j+1}}$

3. $\mathtt{L}_{i_k}$ *contenga esattamente una invocazione di* $\mathtt{f}_i$

La sequenza $\mathtt{f}_i\mathtt{f}_{i_1}\ldots\mathtt{f}_{i_k}$ *è chiamata la* storia ricorsiva (oppure recursive history) *di* $\mathtt{f}_i$.

Esempio 3.1.1. *Il programma seguente è lineare:*
$\Big(\mathtt{f}(x_1,x_2)=(x_1,x_2)\|\mathtt{f}_1(x_2,x_3)\|\mathtt{f}_2(x_2);\mathtt{f}_2(x_3)\ ,\ \mathtt{f}_2(y)=(y,z)\|\mathtt{f}_2(z)\ ,\ \mathtt{L}\Big)$

Esempio 3.1.2. *Un altro esempio di programma non lineare è invece una delle tante possibili versioni che calcolano i numeri di Fibonacci, dove due invocazioni ricorsive sono gestite da due thread paralleli su nuovi oggetti. Segue il LAM associato ad esso:*
$\Big(mFib(x)=(x,y)\|mFib(y)\|(x,z)\|mFib(z)\ ,\ mFib(x)\Big)$

Esempio 3.1.3. *Come ultimo esempio, un programma che non è linare a causa di una doppia storia:*

$$\Big(\mathtt{f}(x)=(x,y)\|\mathtt{g}(x)\ ,\ \mathtt{g}(x)=(x,y)\|\mathtt{f}(x);\mathtt{l}(x)\ ,\ \mathtt{l}(x)=\mathtt{f}(x)\ ,\ \mathtt{L}\Big)$$

La funzione `f` *ha due storie:* `fg` *e* `fgl` *(anche se entrambe portano alla stessa mutazione, come si vedrà più avanti).*

Osservazione 3.1. Nei programmi lineari, l'insieme dei nomi di funzione può essere *partizionato* in un modo tale che `f` e `g` appartengono alla stessa partizione se e solo se essi sono nella stessa *recursive history.*

In un programma $\Big(\mathtt{f}(\widetilde{x_1})=L_1,\ldots,\mathtt{f}_l(\widetilde{x}_l)=\mathtt{L}_l,\mathtt{L}\Big)$, sia $closure(\mathtt{f})$ l'insieme più piccolo delle funzioni tali che

$$\frac{\mathtt{f}(\widetilde{x})=\mathfrak{L}[\mathtt{g}(\widetilde{u})]}{closure(\mathtt{g})\cup\{\mathtt{g}\}\subseteq closure(\mathtt{f})}$$

Si noti come

- Se `f` e `g` appartengono alla stessa partizione, allora $closure(\mathtt{f})=closure(\mathtt{g})$
- Se `f` e `g` sono in partizioni differenti, allora vale una delle seguenti (i) $closure(\mathtt{f})\cap closure(\mathtt{g})=\varnothing$ oppure (ii) $closure(\mathtt{f})\subsetneq closure(\mathtt{g})$ oppure (iii) $closure(\mathtt{g})\subsetneq closure(\mathtt{f})$.

É stato sviluppato un algoritmo efficiente per calcolare le partizioni di un programma, derivato dall'algoritmo *Depth First Search.* L'algoritmo, descritto in Tabella 3.1, tratta le funzioni come se fossero nodi di un grafo, in cui il punto di entrata è la funzione $Main$, e come archi le invocazioni tra di essi. Esso si basa su tre strutture dati: uno stack $\mathbb{S}$, su cui vengono salvati i

nodi in ordine di apparizione durante la *DFS* (è lo stack su cui lavora l'implementazione iterativa della *DFS*), un'associazione tra nodi $\mathbb{T}$ in cui vengono memorizzate le relazioni padre $\rightarrow$ figlio e infine un insieme, $\mathbb{C}$, dove vengono memorizzati i cicli (se esistono) trovati, i cui elementi rappresenteranno le partizioni. Dal punto di vista dell'algoritmo, una partizione altro non è che un ciclo all'interno del grafo composto dai nodi-funzione e archi-invocazione. L'algoritmo è progettato per fermarsi se incontra una funzione che rende il programma non lineare: nell'implementazione esso viene lasciato proseguire, accendendo però un flag di non linearità.

$$\text{(START)}\quad \frac{\mathtt{f} \in Main \quad \mathtt{f}(\widetilde{x}) = L \quad \mathfrak{M} \dot{\in} L}{\langle \epsilon, \varnothing, \mathbb{C} \rangle \longrightarrow \langle \mathtt{f}, [\mathtt{f} \mapsto \mathfrak{M}], \mathbb{C} \rangle}$$

$$\text{(NEXT)}\quad \frac{\mathbb{T}(\mathtt{f}) = \mathtt{g} \cdot \alpha' \quad \mathtt{g} \notin \alpha \cdot \mathtt{f} \quad \mathtt{g}(\widetilde{x}) = L \quad \mathfrak{M} \dot{\in} L}{\langle \alpha \cdot \mathtt{f}, \mathbb{T}, \mathbb{C} \rangle \longrightarrow \langle \alpha \cdot \mathtt{f} \cdot \mathtt{g}, \mathbb{T}[\mathtt{f} \mapsto \alpha'][\mathtt{g} \mapsto \mathfrak{M}], \mathbb{C} \rangle}$$

$$\text{(CYCLE)}\quad \frac{\begin{array}{c} \mathbb{T}(\mathtt{f}) = \mathtt{g} \cdot \alpha' \\ \gamma = \mathtt{g}_1 \dots \mathtt{g}_k \\ \forall C \in \mathbb{C}\ ,\ \{\mathtt{g} \cdot \gamma \cdot \mathtt{f}\} \cap \{C\} = \varnothing \\ \forall \mathtt{l} \in \mathtt{g} \cdot \gamma \cdot \mathtt{f}\ ,\ \mathtt{l} \notin \alpha' \wedge \mathtt{l} \notin \mathbb{T}(\mathtt{g}) \wedge \mathtt{l} \notin \mathbb{T}(\mathtt{g}_i) \quad i = 1 \dots k \end{array}}{\langle \beta \cdot \mathtt{g} \cdot \gamma \cdot \mathtt{f}, \mathbb{T}, \mathbb{C} \rangle \longrightarrow \langle \beta \cdot \mathtt{g} \cdot \gamma \cdot \mathtt{f}, \mathbb{T}[\mathtt{f} \mapsto \alpha'], \mathbb{C} \cup \{\mathtt{g} \cdot \gamma \cdot \mathtt{f}\} \rangle}$$

$$\text{(DELETE)}\quad \frac{\mathbb{T}(\mathtt{f}) = \epsilon}{\langle \alpha \cdot \mathtt{f}, \mathbb{T}, \mathbb{C} \rangle \longrightarrow \langle \alpha, \mathbb{T} \setminus \mathtt{f}, \mathbb{C} \rangle}$$

Tabella 3.1: Algoritmo per partizionare un programma

La regola (START) è responsabile di inizializzare l'intero algoritmo, ponendo una delle funzioni invocate nel $Main$ sullo stack $\mathbb{S}$ e i suoi figli in $\mathbb{T}$. I figli di una funzione, in altre parole le funzioni invocate dalla funzione stessa, sono trovati grazie all'operatore $\dot{\in}$.

Per avanzare all'interno del grafo, si utilizza la regola (NEXT), la quale estrae un figlio ancora da visitare dell'ultima funzione immessa nello stack, e se non è già presente sullo stack stesso la aggiunge ad esso, aggiornando ovviamente la relazione $\mathbb{T}$, perché ora il figlio deve considerarsi visitato. Se invece il figlio appare già sullo stack, si passa alla regola (CYCLE), in cui vengono effettuati controlli supplementari per controllare la linearità del programma. Un programma lineare non può avere più cicli (e quindi partizioni) che condividono

$$\varepsilon \mathrel{\dot\in} (x, y)$$
$$\mathtt{g} \mathrel{\dot\in} \mathtt{g}(\widetilde{x})$$
$$\frac{\mathfrak{M} \mathrel{\dot\in} L \quad \mathfrak{M}' \mathrel{\dot\in} L'}{\begin{array}{c}\mathfrak{M} \cdot \mathfrak{M}' \mathrel{\dot\in} L \| L' \\ \mathfrak{M} \cdot \mathfrak{M}' \mathrel{\dot\in} L; L'\end{array}}$$

Tabella 3.2: Definizione della relazione $\dot\in$

le stesse funzioni (a), e le funzioni che compongono un ciclo non possono avere archi che ricadono in un altro cammino che porta all'interno del ciclo stesso (b).

Esempio 3.1.4. *Per la precondizione (a), si veda il grafo delle invocazioni associato al programma*

$$\begin{array}{rl} \mathtt{f}(x,y)= & \mathtt{g}(x,y) \| (x,y) \quad , \\ \mathtt{g}(x,y)= & \mathtt{f}(x,y)\,;\,(y,z)\,;\,\mathtt{l}(z), \\ \mathtt{l}(x)= & \mathtt{f}(x,y) \quad , \\ \mathtt{f}(x_1,x_2) & \end{array}$$

rappresentato come

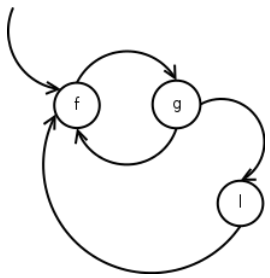


Figura 3.1: grafo delle invocazioni del programma precedente

in cui, una volta scoperto il ciclo `fg`*, l'intersezione con il ciclo successivo* `fgl` *non è vuota, ma contiene la funzione* `l`*. Questo comporta che il figlio di* `l`*, e cioè* `f`*, ha una doppia storia ricorsiva (e non è quindi lineare).*

La precondizione (b) è necessaria per catturare situazioni di non linearità simili a quella contenuta nel programma

$$\begin{array}{rl} \mathtt{f}(x)= & (x,y)\,;\,\mathtt{g}(y) \quad , \\ \mathtt{g}(x)= & (x,y) \| \mathtt{f}(x) \| \mathtt{f}(y), \\ \mathtt{f}(x_1) & \end{array}$$

il cui grafo delle invocazioni è:

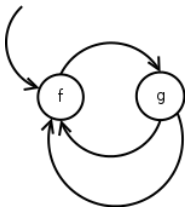


Figura 3.2: Grafo delle invocazioni del programma `fg`

È evidente come, una volta scoperto il ciclo `fg`*, rimanga ancora un arco da visitare per* `g`*, che però fa parte di un altro cammino che arriva al ciclo* `fg`*. Dunque,* `f` *ha ancora una volta una doppia storia ricorsiva, rendendo il modello LAM non lineare.*

La regola (DELETE) infine si occupa di liberare lo stack e la relazione $\mathbb{T}$ quando non ci sono più nodi da visitare sul cammino attuale, preparando le strutture dati all'esplorazione di un cammino alternativo.

Se si applica l'algoritmo della Tabella 3.1 al programma

$$M(x,y) = (x,y) \| H(x); N(y)\ ,\ N(x) = (x,y) \| O(y)\ ,\ O(x) = (x,y); M(x,y)\ ,$$
$$H(x) = (x,y) \| I(x,y)\ ,\ I(x,y) = (x,y); H(y)\ ,\ M(x_1, y_1)$$

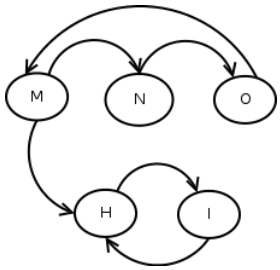


Figura 3.3: Un esempio di albero delle invocazioni

il cui albero delle invocazioni è rappresentato in Figura 3.3, l'evoluzione della tupla $\langle \mathbb{S}, \mathbb{T}, \mathbb{C} \rangle$ avviene tramite i seguenti passaggi:

$$
\begin{aligned}
&\langle \epsilon, \varnothing, \varnothing \rangle \overset{START}{\longrightarrow} \langle M, [M \mapsto N \cdot H], \varnothing \rangle \\
&\overset{NEXT}{\longrightarrow} \langle M \cdot N, [M \mapsto H, N \mapsto O], \varnothing \rangle \\
&\overset{NEXT}{\longrightarrow} \langle M \cdot N \cdot O, [M \mapsto H, N \mapsto \varnothing, O \mapsto M], \varnothing \rangle \\
&\overset{CYCLE}{\longrightarrow} \langle M \cdot N \cdot O, [M \mapsto H, N \mapsto \varnothing, O \mapsto \varnothing], \{(M \cdot N \cdot O)\} \rangle \\
&\overset{DELETE}{\longrightarrow} \langle M \cdot N, [M \mapsto H, N \mapsto \varnothing], \mathbb{C}_1 \rangle \\
&\overset{DELETE}{\longrightarrow} \langle M, [M \mapsto H], \mathbb{C}_1 \rangle \\
&\overset{NEXT}{\longrightarrow} \langle M \cdot H, [M \mapsto \varnothing, H \mapsto I], \mathbb{C}_1 \rangle \\
&\overset{NEXT}{\longrightarrow} \langle M \cdot H \cdot I, [M \mapsto \varnothing, H \mapsto \varnothing, I \mapsto M], \mathbb{C}_1 \rangle \\
&\overset{CYCLE}{\longrightarrow} \langle M \cdot H \cdot I, [M \mapsto \varnothing, H \mapsto \varnothing, I \mapsto \varnothing], \mathbb{C}_1 \cup \{(H \cdot I)\} \rangle \\
&\overset{DELETE}{\longrightarrow} \cdots \overset{DELETE}{\longrightarrow} \cdots \overset{DELETE}{\longrightarrow} \langle \epsilon, \varnothing, \{(M \cdot N \cdot O, H \cdot I)\} \rangle
\end{aligned}
$$

Dove $\mathbb{C}_1 = \{(M \cdot N \cdot O)\}$.

3.2 Assegnare una mutazione ad una funzione lineare

La linearità permette di definire un' *unica* mutazione per ogni funzione. Per calcolarla, dato un programma lineare $\Big(\mathtt{f}(\widetilde{x_1}) = \mathtt{L}_1, \ldots, \mathtt{f}_l(\widetilde{x}_l) = \mathtt{L}_l, \mathtt{L}\Big)$ si usano le seguenti regole:

$$
\frac{\mathtt{f}_i\alpha \models \varepsilon \quad \mathtt{f}_i(\widetilde{x}_i) = \mathtt{L}_i}{\alpha \models \mathtt{f}_i(\widetilde{x}_i)}
\qquad
\frac{\begin{array}{c}\mathtt{f}_j\alpha \models \mathtt{Hf}_i(\widetilde{x}) \quad \mathtt{f}_i(\widetilde{x}_i) = \mathtt{L}_i \\ var(\mathtt{L}_i) \setminus \widetilde{x}_i = \widetilde{z} \qquad \widetilde{w} \text{ sono fresh} \\ \mathfrak{L}[\mathtt{f}_j(\widetilde{y})] = \mathtt{L}_i[\widetilde{w}/\widetilde{z}][\widetilde{x}/\widetilde{x}_i]\end{array}}{\alpha \models \mathtt{Hf}_i(\widetilde{x})\mathtt{f}_j(\widetilde{y})}
$$

Dove $\mathtt{H}$ spazia su sequenze di invocazioni di funzione.

Sia $\varepsilon \models \mathtt{f}(x_1, \cdots, x_n) \cdots \mathtt{f}(x'_1, \cdots, x'_n)$ il giudizio finale dell'albero di dimostrazione che parte da $\mathtt{f}\alpha\mathtt{f} \models \varepsilon$, dove $\mathtt{f}\alpha$ è la storia ricorsiva di $\mathtt{f}$. Sia anche $x'_1, \cdots, x'_n \setminus x_1, \cdots, x_n = z_1, \cdots, z_k$. Allora la *mutazione* di $\mathtt{f}$, scritta come $\mu_{\mathtt{f}} = (\!| a_1, \cdots, a_n |\!)$ è definita da

$$
a_i = \begin{cases} j & \text{if } x'_i = x_j \\ \\ n + j & \text{if } x'_i = z_j \end{cases}
$$

Sia $\mathfrak{o}_{\mathtt{f}}$, *ordine della funzione* $\mathtt{f}$, l'ordine di $\mu_{\mathtt{f}}$. Per esempio, nel programma `flh`, definito come:

$$\Big(\begin{array}{rcc} \mathtt{f}(x,y,z,u) & = & (x,z)\|\mathtt{l}(u,y,z), \\ \mathtt{l}(x,y,z) & = & (x,y)\|\mathtt{f}(y,z,x,u), \\ \mathtt{h}(x,y,z,u) & = & (z,x)\|\mathtt{h}(x,y,z,u)\|\mathtt{f}(x,y,z,u),\ \mathtt{L}\end{array}\Big)$$

La storia ricorsiva di `f` è `fl`, ed applicando il precedente algoritmo si ottiene:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{flf} \models \varepsilon}}{\mathtt{lf} \models \mathtt{f}(x,y,z,u)}}{\mathtt{f} \models \mathtt{f}(x,y,z,u)\mathtt{l}(u,y,z)}}{\varepsilon \models \mathtt{f}(x,y,z,u)\mathtt{l}(u,y,z)\mathtt{f}(y,z,u,v)}$$

La mutazione di `f` è $\{\!\!\{2,3,4,5\}\!\!\}$ e $\mathfrak{o}_{\mathtt{f}} = 4$. Allo stesso modo si può calcolare $\mathfrak{o}_{\mathtt{l}} = 3$ e $\mathfrak{o}_{\mathtt{h}} = 1$.

3.3 Saturazione della storia ricorsiva

D'ora in poi, fino al termine del capitolo, si assumerà un programma lineare fissato $\Big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\Big)$ con $\mathfrak{o}_{\mathtt{f}_1}, \cdots, \mathfrak{o}_{\mathtt{f}_\ell}$ ordini delle funzioni corrispondenti.

Definizione 3.2. Una storia α è

`f`-*completa*
: se $\alpha = \beta^{\mathfrak{o}_{\mathtt{f}}}$, dove β è la storia ricorsiva di `f`. Si dice che α è *complete* quando è `f`-completa, per qualche `f`.

`f`-*saturata*
: se $\alpha = \beta_1 \cdots \beta_{n-1}\alpha_n^2$, dove $\beta_i \preceq (\alpha_i)^2$, con α_i completa, e α_n `f`-completa. Si dice che α è *saturata* quando è `f`-saturata, per qualche `f`.

Nel programma `flh`, $\mathfrak{o}_{\mathtt{f}} = 4$, $\mathfrak{o}_{\mathtt{l}} = 3$, e $\mathfrak{o}_{\mathtt{h}} = 1$, e le storie ricorsive di `f`, `l` e `h` sono `fl`, `lf` e `h`, rispettivamente. Dunque $\alpha = (\mathtt{fl})^4$ è la storia `f`-completa, e $\mathtt{h}^2(\mathtt{fl})^8$ e $\mathtt{h}(\mathtt{fl})^8$ sono `f`-saturate.

La nozione di `f`-saturazione verrà usata per definire uno stato "saturato", i quali sono stati in cui la ricerca di circolarità può fermarsi.

Definizione 3.3. Un LAM informativo $\Big\langle \mathbb{V}, {}^{\mathfrak{h}}\mathbb{F}, \mathbb{L} \Big\rangle$ è *saturato* quando, per ogni ${}^{\mathfrak{h}}\mathfrak{L}[\ ]$ e $\mathtt{f}(\widetilde{u})$ tali che $\mathbb{L} = {}^{\mathfrak{h}}\mathfrak{L}[{}^{\alpha}\mathtt{f}(\widetilde{u})]$, allora α ha un prefisso saturante.

É facile controllare che il seguente LAM informativo generato dalla computazione del programma `flh` è saturato:

$$\Big\langle \mathbb{V}_7,\ {}^{\natural}\mathbb{F},\ {}^{\mathtt{h}^2}\mathtt{h}(x_1,x_2,x_3,x_4) \quad \| \prod_{0\le i\le 8} {}^{\mathtt{hf}(\mathtt{lf})^i}(x_{i+1},x_{i+3}) \\ \| \prod_{0\le i\le 8} {}^{\mathtt{h}(\mathtt{fl})^i}(x_{i+3},x_{i+1}) \\ \| {}^{\mathtt{h}(\mathtt{fl})^8}\mathtt{f}(x_9,x_{10},x_{11},x_{12})\Big\rangle,$$

dove $\mathbb{V}_{i+1} = \mathbb{V}_i \oplus x_{i+4}{<}x_{i+5}$, e

$$\begin{aligned} {}^{\natural}\mathbb{F} = \ & \{{}^{\varepsilon}\mathtt{h}(x_1,x_2,x_3,x_4), {}^{\mathtt{h}}\mathtt{h}(x_1,x_2,x_3,x_4)\} \\ & \cup \{{}^{\mathtt{h}(\mathtt{fl})^i}\mathtt{f}(x_{i+1},x_{i+2},x_{i+3},x_{i+4}) \mid 0\le i\le 7\} \\ & \cup \{{}^{\mathtt{hf}(\mathtt{lf})^i}\mathtt{l}(x_{i+4},x_{i+2},x_{i+3}) \mid 0\le i\le 7\}. \end{aligned}$$

Si noti come la saturazione è più complicata quando i programmi hanno anche funzioni non ricorsive. L'espansione di funzioni non ricorsive può essere effettuata in qualsiasi stato di una valutazione (sempre che la loro invocazione sia nello stato), rompendo di fatto la regolarità delle storie.

Teorema 3.3.1. *Sia* $\langle \mathbb{I}_{\mathtt{L}},\ \varnothing,\ addh(\varepsilon,\mathtt{L})\rangle \longrightarrow^* \langle \mathbb{V},\ {}^{\natural}\mathbb{F},\ \mathbb{L}\rangle$ *e* $\langle \mathbb{V},\ {}^{\natural}\mathbb{F},\ \mathbb{L}\rangle$ *sia saturato. Se* $\langle \mathbb{V},\ {}^{\natural}\mathbb{F},\ \mathbb{L}\rangle \longrightarrow \langle \mathbb{V}',\ {}^{\natural}\mathbb{F}',\ \mathbb{L}'\rangle$ *allora*

1. $\langle \mathbb{V}',\ {}^{\natural}\mathbb{F}',\ \mathbb{L}'\rangle$ *è saturato;*
2. *se* $\mathbb{L}'$ *ha una circolarità allora* $\mathbb{L}$ *già aveva una circolarità.*

La teoria delle mutazioni è un elemento basilare nella prova del teorema, che si può trovare nell'articolo [1].

Teorema 3.3.2. *Trovare circolarità in un programma LAM è un problema decidibile quando il programma stesso è lineare.*

L'algoritmo è il seguente:

1. Per ogni invocazione nel LAM principale, si calcoli l'opportuna storia saturata e si eseguano le transizioni corrispondenti (il numero di transizioni è finito, e ogni transizione è *essa stessa finita*);
2. Le transizioni nate per il primo passo dell'algoritmo, riguardanti `f`, possono a loro volta generare invocazioni di "funzioni di livello inferiore" `g` tali che *closure*(`g`) $\subsetneq$ *closure*(`f`) (si guardi l'Osservazione 3.1). Queste sono finite, e per ogni invocazione, si riapplichi il primo passo dell'algoritmo;

3. La procedura termina perché le sequenze $closure(\mathtt{f}_1) \subsetneq closure(\mathtt{f}_2) \subsetneq \cdots$ sono finite (a causa del fatto che il programma è finito).

Questo algoritmo porta alla saturazione del LAM principale, dal quale si possono estrarre degli insiemi di stati, in cui devono essere controllate le assenze di circolarità tra dipendenze. La correttezza segue dal Teorema 3.3.1 e dalla *diamond property* (possono esistere ovviamente altre computazioni, che termineranno in uno stato uguale al più per una biiezione allo stato saturato trovato).

3.4 Implementazione

Ogni funzione viene rappresentata, oltre che testualmente, attraverso un albero di sintassi astratta, creato automaticamente a partire dalla rappresentazione scritta da un parser. Sono memorizzati anche alcuni punti importanti dell'albero stesso, come ad esempio il punto esatto di ogni occorrenza di una variabile libera o di una variabile legata, oppure l'invocazione di una funzione, tramite una visita iniziale, effettuata una sola volta. L'utilità di questa operazione di *caching* diverrà chiara quando si presenterà l'operatore di *merge*. Il parser stesso viene generato automaticamente da SableCC[1], un framework completamente ad oggetti descritto in [26], per la creazione di compilatori ed interpreti. In particolare, il codice generato per l'albero di sintassi astratta è fortemente tipato, includendo anche algoritmi per la visita dell'albero stesso. SableCC crea il codice partendo dalla seguente grammatica formale, che si definirà "astratta" per la mancanza di regole precise per la gestione della precedenza degli operatori e per gli elementi terminali, fatta eccezione per il terminale "0".

$$
\begin{array}{lll}
\langle expr \rangle & ::= & \langle expr \rangle \text{ `;' } \langle expr \rangle \\
& | & \langle expr \rangle \text{ `||' } \langle expr \rangle \\
& | & \text{`(' } \langle expr \rangle \text{ `)'} \\
& | & \text{`(' } \langle id \rangle \text{ `,' } \langle id \rangle \text{ `)'} \\
& | & \langle id \rangle \text{ `(' } \langle id \rangle \text{ [, } \langle id \rangle \text{]* `)'} \\
& | & \text{`0'}
\end{array}
$$

Per gestire le precedenze, si ricorre all'utilizzo di una grammatica concreta, la quale specifica in ogni dettaglio un modello LAM. Una volta generato l'albero di sintassi concreto, esso viene trasformato in un albero di sintassi

[1] Reperibile su `http://sablecc.org/`

astratto (che rispetta dunque la grammatica astratta) attraverso una procedura automatica gestita anch'essa da SableCC, la quale mappa produzioni concrete in produzioni astratte.

⟨*expr*⟩ ::= ⟨*subexpr*⟩
| ⟨*subexpr*⟩ ‘;’ ⟨*expr*⟩

⟨*subexpr*⟩ ::= ⟨*term*⟩
| ⟨*term*⟩ ‘||’ ⟨*subexpr*⟩

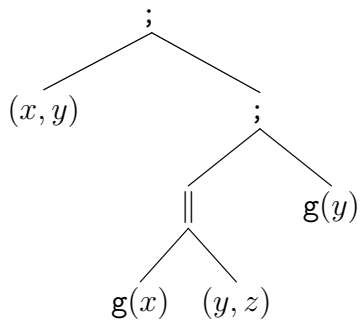
⟨*term*⟩ ::= ‘(’ ⟨*expr*⟩ ‘)’
| ‘(’ ⟨*id*⟩ ‘,’ ⟨*id*⟩ ‘)’
| ‘0’
| ⟨*id*⟩ ‘(’ ⟨*arglist*⟩ ‘)’

⟨*arglist*⟩ ::= ⟨*id*⟩
| ⟨*id*⟩ ‘,’ ⟨*arglist*⟩

⟨*id*⟩ ::= [A-Za-z]+ [0-9]*

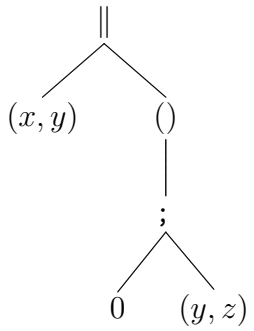
Viene rispettata la gerarchia tra gli operatori, in cui “$\|$” ha maggiore precedenza rispetto a “; ”, a meno di espressioni tra parentesi. La mappatura da albero concreto ad albero astratto avviene tramite semplici ed ovvie regole, che indicano a SableCC come trasformare ogni nodo (o sottoalbero) dell'albero concreto in un nodo dell'albero astratto.

Esempio 3.4.1. *Per la funzione* $\mathtt{f}(x,y) = (x,y); \mathtt{g}(x)||(y,z); \mathtt{g}(y)$ *viene generato il seguente albero:*



che corrisponde all' interpretazione corretta, che è $\mathtt{f}(x,y) = (x,y); ((\mathtt{g}(x)||(y,z)); \mathtt{g}(y))$.

Esempio 3.4.2. *La funzione* $\mathtt{g}(x) = (x, y)||(0; (y, z))$*, per la presenza delle parentesi, genera:*



Il primo passo pratico è dunque quello di generare gli alberi astratti per tutte le funzioni. Nel farlo, occorre memorizzare le funzioni chiamate (i "figli" del chiamante) attraverso un dizionario. Ogni chiamata di funzione è, nell'albero astratto, una foglia, la quale sarà memorizzata nel dizionario come chiave, con il valore corrispondente rappresentato dalla funzione stessa.

A questo punto si rende necessaria la scoperta delle partizioni del programma, che permette di determinare se il programma in input sia lineare (grazie alle precondizioni nelle regole "next" e "cycle"). In più, le partizioni verranno riutilizzate quando si dovranno calcolare gli alberi saturati.

Grazie alla rappresentazione che tiene traccia delle funzioni "figlie", è facile implementare l'algoritmo descritto in Tabella 3.1, e una sua descrizione in pseudocodice si trova nell'Algoritmo 1.

Con le partizioni del programma memorizzate, si può iniziare a costruire la storia saturata delle funzioni invocate nel Main del LAM. L'intero procedimento ha inizio da una funzione che rappresenta la "radice" dell'albero delle invocazioni, e cioè il Main stesso.

Ricavare la storia saturata di una funzione `f`, se il programma è lineare, in sè non è un compito estremamente difficile (si riveda la Definizione 3.2): occorre trovare la mutazione associata alla funzione stessa (procedimento illustrato in Sezione 3.2), calcolarne l'ordine, e ripetere la storia di `f` per un numero pari a $\mathfrak{o}_{\mathtt{f}} * 2$ volte. Il caso in cui il programma non sia lineare verrà trattato nei capitoli successivi: si assumeranno quindi tutti programmi lineari. La conoscenza delle partizioni si rivela estremamente utile per calcolare la mutazione associata, perché è possibile ignorare le chiamate a funzioni appartenenti a livelli inferiori, dato che non influiscono sulla mutazione. In altre parole, sulla mutazione influiscono solo le funzioni appartenenti alla stessa partizione, e nella fattispecie gli argomenti passati a queste funzioni. Con un numero di iterazioni pari alla dimensione della partizione, si riesce ad avere

Algorithm 1 Partizionare un programma

```
function getPartition(root)
    𝕊.push(root)
    𝕋.map(root, root.children)                    ▷ Regola 1: Start
    while not 𝕊.isEmpty() do
        f ← firstElementOf(𝕊)
        children ← 𝕋.get(f)
        if children is empty then                 ▷ Regola 4: delete
            eliminazione da 𝕊 di f
            eliminazione da 𝕋 di f
        end if
        g ← children.getFirst()
        if g ∈ 𝕊 then                             ▷ Regola 3: cycle
            Inserisci il ciclo in ℂ
        else                                      ▷ Regola 2: next
            𝕊.push(g)
            𝕋.map(g, g.children)
        end if
    end while
    return ℂ
end function
```

le due sequenze (quella di partenza x_i e quella di arrivo x'_i) da confrontare per poterne ricavare la mutazione associata.

Durante la saturazione, i problemi sorgono quando, all'interno della storia di `f` esistono invocazioni a funzioni mutuamente ricorsive di livello inferiore (e quindi appartenenti ad altre partizioni) le quali si portano dietro la loro storia, che deve essere ripetuta per un numero di volte pari al loro ordine moltiplicato per due. Ciò significa che ogni invocazione viene duplicata per $\mathfrak{o}_{\mathtt{f}} * 2$ volte, ed ognuna di esse al suo interno può chiamare altre partizioni, sempre però di livello inferiore. Chiaramente, occorre un approccio ragionato al problema per evitare di crescere troppo come spazio e tempo di computazione quando si andrà a tenere traccia delle dipendenze generate da ogni invocazione.

Esempio 3.4.3. *Per calcolare la storia saturata del programma $MNOP$, definito come*

$$\begin{pmatrix} M(x,y) & = (x,y) \| N(x)\,;\,(y,z) \| H(z,y)\,;\,H(x,y), \\ N(x) & = (x,y)\,;\,O(y), \\ O(x) & = (x,y) \| P(y), \\ P(x) & = M(x,y) \| F(y,z,w), \\ H(x,y) & = I(y)\,;\,(x,y), \\ I(x) & = G(x) \| (x,y), \\ G(x) & = H(x,y)\,;\,F(y,z,w) \\ F(x,y,z) & = (x,y)\,;\,E(z,y,x,w) \\ E(x,y,z,w) & = (x,w) \| F(w,z,x) \\ M(a,b) & \end{pmatrix}$$

si esegue l'algoritmo per partizionare, che restituisce come partizioni $[MNOP, HIG, FE]$. Dato che il Main del programma contiene esclusivamente M, essa diventa il nodo root dell'albero delle invocazioni. Calcolando gli ordini delle funzioni in esso presenti, si vede come

$$\begin{aligned} \mathfrak{o}_M = \mathfrak{o}_N = \mathfrak{o}_O = \mathfrak{o}_P &= 1 \\ \mathfrak{o}_H = \mathfrak{o}_I = \mathfrak{o}_G &= 1 \\ \mathfrak{o}_F &= 2 \\ \mathfrak{o}_E &= 3 \end{aligned}$$

Per calcolare gli ordini, si esegue esattamente lo stesso procedimento illustrato in Sezione 3.2. Si prenda ad esempio il metodo E: la mutazione ad

esso associata si trova arrivando al giudizio finale dell'albero di dimostrazione che parte da $EFE \models \varepsilon$

$$\dfrac{\dfrac{\dfrac{EFE \models \varepsilon}{EF \models E(x,y,z,w)}}{E \models E(x,y,z,w)F(w,z,x)}}{\varepsilon \models E(x,y,z,w)F(w,z,x)E(x,z,w,w')}$$

ed, avendo la foglia $\varepsilon \models E(x,y,z,w)F(w,z,x)E(x,z,w,w')$*, è facile arrivare alla mutazione, che è* $\mu_E = [\![1,3,4,5]\!]$*. Da qui, occorre calcolarne l'ordine: è facile vedere la presenza di un* free sink *di lunghezza* 3 *ed un ciclo la cui lunghezza è* 1*. Secondo il Teorema 2.1.1, l'ordine risulta* 3*. Lo stesso procedimento si applica alle altre funzioni.*

Avendo quindi $\mathfrak{o}_M = 1$*, occorre ripetere la storia di* M*, che equivale alla sua partizione* $MNOP$*, per un numero pari a* $\mathfrak{o}_M * 2 = 2$ *volte.*

Figura 3.4: Storia di M ripetuta per due volte

Se si considerano le chiamate a funzioni di partizioni di livello inferiore, si deve tenere conto che ogni M *chiama due volte la funzione* H*, mentre la funzione* P *chiama una volta sola* F*. Inoltre, all'interno della partizione di* H *esiste una invocazione di* F*. Ognuna di queste occorrenze deve essere saturata, dunque la storia deve essere ripetuta per un numero di volte pari al loro ordine moltiplicato per due (nel caso delle invocazioni di* H*, la ripetizione di* HIG *avviene per due volte, mentre per le invocazioni di* F *la saturazione impone di ripetere la storia* FE *per sei volte).*

Notare che i nodi contrassegnati con $\cdots$ *contengono dei sottoalberi, esclusi per ragioni di spazio e di visibilità, uguali a quelli già visti (ad esempio, dai nodi* P *parte la sequenza* $FEFEFE$*). Si omettono inoltre gli archi tra nodi disposti sulla stessa linea orizzontale, che rappresentano le chiamate che avvengono tra una funzione e l'altra. A questo punto si può iniziare a visitare l'albero e tenere traccia delle dipendenze.*

Per evitare di ripetere calcoli inutili e dispendiosi, si è scelto di approcciare il problema con un metodo *divide et impera*. Occorre innanzitutto dare una definizione di *partizione espansa* a partire da un rappresentante canonico e dell'operatore di *merge*, su cui essa è interamente basata.

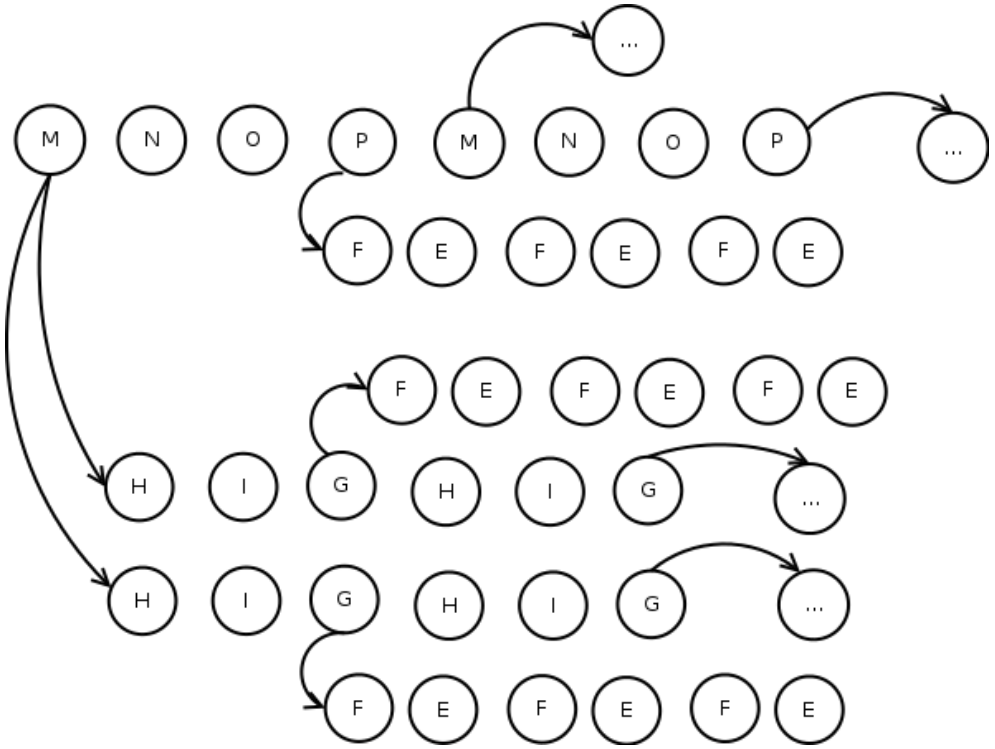


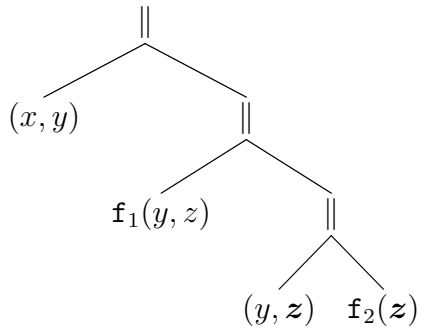
Figura 3.5: Espansione delle invocazioni nella storia

Definizione 3.4. Date due funzioni `f`, `g` l'operatore di *merge* su `f` e `g`, sritto come `f` ∪ `g` restituisce la funzione `f` in cui tutte le chiamate a `g` sono rimpiazzate con il corpo di `g` stesso opportunamente istanziato, rispettando le regole di precedenza tra operatori (quindi se necessario instanziando il corpo all'interno di parentesi).

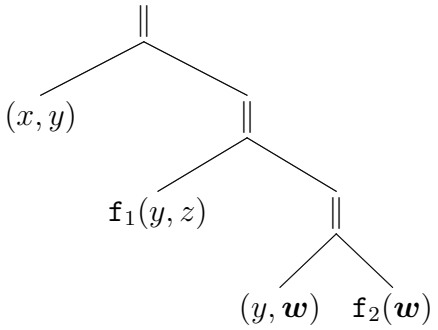
L'implementazione dell'operatore di merge è il punto più delicato, essendo il mattone fondamentale di tutti gli algoritmi presentati in seguito. All'interno di un albero di sintassi astratta, le chiamate a funzione sono delle foglie, una per ogni invocazione. Una volta scelta la foglia, si può procedere alla sostituzione della foglia stessa con l'albero della funzione invocata, opportunamente duplicato. Il passaggio successivo è quello dell'instanziazione corretta delle variabili all'interno del sottoalbero appena aggiunto. Esso contiene ancora l'instanziazione precedente, quella con i parametri formali (e le variabili libere) della funzione originale: si procede dunque con l'assegnazione di nuove variabili libere e con la sostituzione di quelle legate con i parametri dell'invocazione, recuperati dalla foglia appena sostituita. È evidente come occorra duplicare l'albero originale, ed effettuare le sostituzioni esclusivamente nell'albero duplicato, perché se si avesse la necessità di unire

più volte lo stesso albero, e non duplicandolo ma semplicemente usando lo stesso puntatore più volte, cambiare variabili in uno equivarrebbe a cambiarle in tutti, rovinando così l'intera procedura. Per la gestione delle variabili libere, si ricorre ad una classe "generatrice di nomi". Essa scorre l'albero della funzione, alla ricerca dei nomi utilizzati (sia quelli legati sia quelli liberi) e, all'occorrenza, restituisce una lista di nomi *fresh*, nuovi, e cioè mai utilizzati all'interno della funzione stessa. In realtà, invece di scorrere l'intero albero, essa conosce già la locazione esatta delle variabili libere (ma anche di quelle legate, se ne necessita) grazie all'operazione di *caching* dei puntatori dopo la visita iniziale, presentata all'inizio della sottosezione. L'assegnazione delle variabili libere avviene quindi sostituendo ogni occorrenza con la nuova variabile, creata dalla classe generatrice. Questo assicura che, anche unendo due alberi con le stesse variabili libere, esse rimangano distinte nell'instanziazione finale.

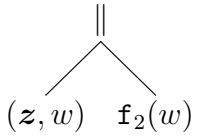
Esempio 3.4.4. *Nel programma* $\Big(\mathtt{f}_1(x,y) = (x,y) \| \mathtt{f}_1(y,z) \| \mathtt{f}_2(z)\ ,\ \mathtt{f}_2(y) = (y,z) \| \mathtt{f}_2(z)\ ,\ \mathtt{L}\Big)$ *L'operazione di* merge *tra* M_1 *e* M_2 *senza una sostituzione delle variabili libere (e neanche delle variabili legate) porterebbe al seguente albero:*



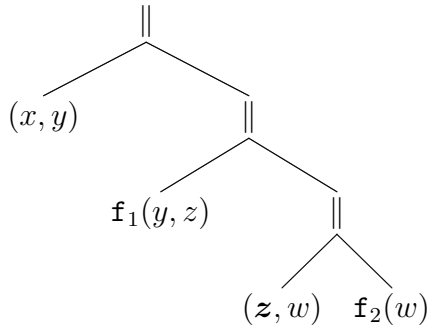
Le variabili $\boldsymbol{z}$ *in grassetto, erano variabili libere all'interno di* $\mathtt{f}_2$*. Nella sua instanziazione, esse si devono in qualche modo distinguere dalle variabili* z *di* $\mathtt{f}_1$*, pena la non correttezza dell'albero instanziato. Il passaggio corretto è, come evidenziato precedentemente, di farsi creare un nome nuovo per ogni variabile libera dalla classe generatrice di nomi, la quale sa che in* $\mathtt{f}_1$ *sono stati usati i nomi* x, y, z *e ne restituisce, ad esempio,* w*. L'albero corretto, che distingue tra le* z *di* $\mathtt{f}_1$ *e le* z *di* $\mathtt{f}_2$ *risulta:*



L'esempio 3.4.4 mette in luce anche la necessità di una corretta istanziazione delle variabili legate. Infatti, la funzione $\mathtt{f}_1$ chiama $\mathtt{f}_2$ passandogli il parametro z, mentre nell'albero instanziato rimane il vecchio valore legato, y, che non è assolutamente corretto. Si provvede alla sostituzione delle variabili legate tramite una semplice mappatura tra i parametri formali e quelli reali, modificando opportunamente le foglie che contengono le variabili legate all'interno del sottoalbero appena unito. Avendo come mappatura $y \to z$, il sottoalbero di $\mathtt{f}_2$ da unire a $\mathtt{f}_1$ dell'esempio precedente diventa

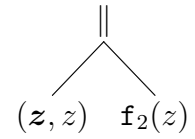


Mentre tutta l'instanziazione di M_1 risulta

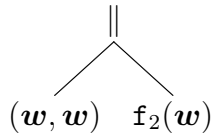


L'esempio 3.4.4 è utile anche per capire come mai occorre sostituire prima le occorrenze libere e poi quelle legate: facendo l'inverso può capitare di trovarsi nella spiacevole situazione di una collisione di nomi. Nell'esempio 3.4.5 si procede all'instanziazione *errata* che sostituisce prima le variabili legate e poi quelle libere.

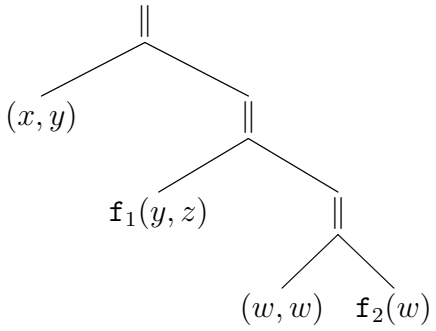
Esempio 3.4.5. *Con riferimento al programma dell'esempio 3.4.4, si instanzia dapprima la variabile legata all'interno del sottoalbero di* $\mathtt{f}_2$*, con il parametro reale z:*



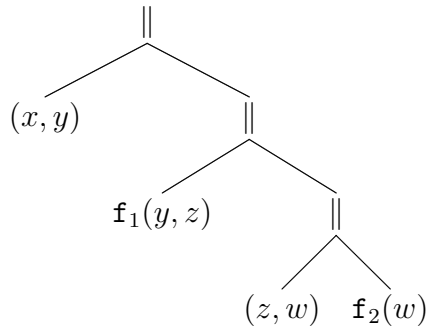
e poi si procede all'instanziazione delle variabili libere, che in M_2 sono tutte le z:



È avvenuta la collisione tra nomi: la variabile z, che era legata, è stata interpretata come libera, e sostituita (erroneamente) con w. L'albero che ne risulta è



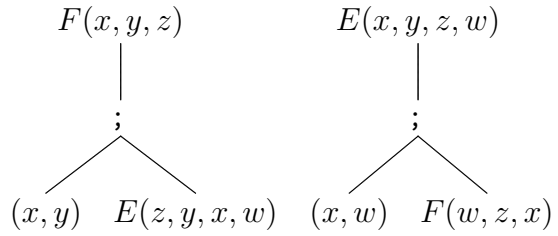
totalmente sbagliato e non corretto. Eseguendo una corretta instanziazione sia per le variabili libere che quelle legate, il risultato che ne deriva è il seguente:



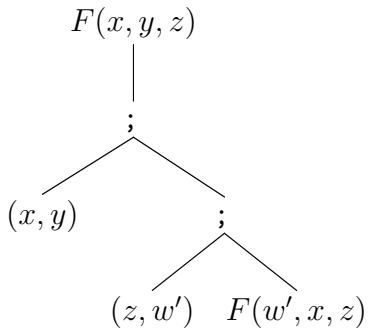
Definizione 3.5. Data una partizione $P = [\mathtt{f}_0, \mathtt{f}_1, \cdots, \mathtt{f}_k]$ e il suo rappresentante $\mathtt{f}_i$, la partizione espansa a partire da $\mathtt{f}_i$, scritta come $P_{\mathtt{f}_i}$, è definita ricorsivamente come

- $P_{\mathtt{f}_k} = \mathtt{f}_k$
- $P_{\mathtt{f}_i} = \mathtt{f}_i \cup P_{\mathtt{f}_{i+1}}$

Si noti che, qualunque sia il rappresentante canonico scelto, la partizione espansa contiene esattamente una invocazione della prima funzione della partizione (come da definizione). Ciò significa che, al termine della procedura, essa conterrà esattamente una invocazione ad $\mathtt{f}_0$, dato che $\mathtt{f}_k$ contiene esattamente una invocazione ad $\mathtt{f}_0$. Altre invocazioni, riferite a funzioni contenute in partizioni inferiori o funzioni non (mutuamente) ricorsive, non vengono prese in considerazione e dunque non sono espanse. Riguardo all'esempio 3.4.3, tenendo in considerazione soltanto le funzioni F ed E, le quali appartegono alla partizione FE



Si può calcolare la partizione espansa partendo dal rappresentante F:



Un altra nozione importante ai fini della costruzione dell'albero saturato è quella della *forma normale* di una partizione.

Definizione 3.6. L'ordine di una partizione $P = [\mathtt{f}_0, \mathtt{f}_1, \cdots, \mathtt{f}_k]$, scritto come $\mathfrak{o}_P$, è l'ordine massimo delle funzioni appartenenti a P:

$$\mathfrak{o}_P = \max\{\mathfrak{o}_{\mathtt{f}_0}, \cdots, \mathfrak{o}_{\mathtt{f}_k}\}$$

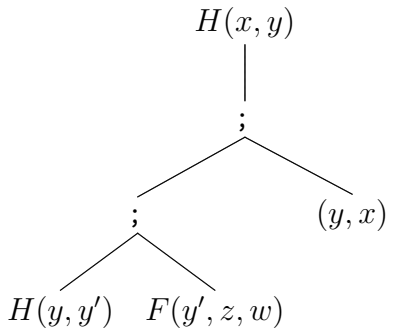
Definizione 3.7. La forma normale di una partizione $P = [\mathtt{f}_0, \mathtt{f}_1, \cdots, \mathtt{f}_k]$ è definita come

$$\Big(\bigcup_{i=1}^{(2*\mathfrak{o}_P)} P_{\mathtt{f}_0}\Big)[^0/_{\mathtt{f}_0(\widetilde{x})}]$$

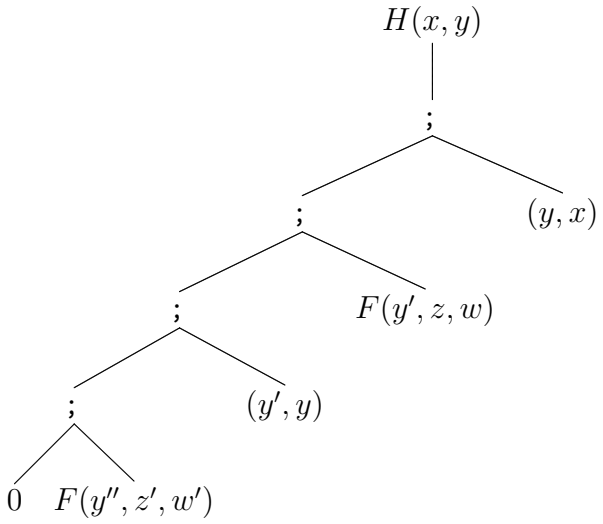
In altre parole occorre unire, tramite l'operazione di *merge*, $2 * \mathfrak{o}_P$ volte la partizione espansa dal rappresentante canonico $\mathtt{f}_0$, ed infine sostituire l'ultima invocazione ad $\mathtt{f}_0$ stessa con il terminale "0". Anche qui invocazioni di funzione al di fuori di quelle contenute nella partizione non vengono espanse.

Esempio 3.4.6. *Se la partizione $P = [MNOP]$ ha ordine 3, la forma normale della partizione è $[MNOPMNOPMNOPMNOPMNOPMNOP][^0/_{M(\widetilde{x})}]$*

Riguardo all'esempio 3.4.3, avendo a disposizione la partizione di HIG espansa a partire dal rappresentante H:



È facile ottenere la forma normale della partizione HIG, considerando che si ha $\mathfrak{o}_{HIG} = 1$:



Cioè, come si può vedere, la chiamata $H(y, y')$ è stata espansa all'interno della chiamata $H(x, y)$, istanziando opportunamente le variabili in esso contenute.

Nell'Algoritmo 2 è presentato il procedimento per la saturazione di una funzione `f`, il quale è principalmente una successione di operazioni di *merge*.

Algorithm 2 Saturare una funzione

```
1:  function SATURATE(f)
2:      if f has not children then
3:          return f
4:      end if
5:      saturated ← null
6:      if f ∈ partition p then
7:          EP_f ← p.getEPFrom(f)              ▷ partizione espansa da f
8:          NF_p ← p.getNF()                   ▷ forma normale di p
9:          saturated ← EP_f ∪ NF_p
10:     else
11:         saturated ← f.clone()
12:     end if
13:     for child in saturated.children() do
14:         saturated ← saturated ∪ SATURATE(child)
15:     end for
16:     return saturated
17: end function
```

L'algoritmo è ricorsivo, ed ha come caso base (linea 3) una funzione `f` che non ha figli. Essa non ha una storia ricorsiva, e quindi risulta già saturata, senza bisogno di ulteriori modifiche. Se invece la funzione è contenuta all'interno di una partizione, chiamata p nello pseudocodice, occorre generare la partizione espansa che parte da `f`, e ad essa unire, alla linea 9, la forma normale di p. Si noti come, da Definizione 3.7, mancando l'invocazione ricorsiva all'interno della forma normale, il risultato dell'unione non è ricorsivo. Quando si arriva alla linea 13, si è sicuri che ogni figlio di `f` o non è ricorsivo, o è contenuto in una partizione di livello inferiore (che quindi non chiamerà mai la partizione di cui si è appena unita la forma normale). Si itera dunque sui figli stessi, i quali sono finiti, unendo la loro versione saturata. La funzione ritorna un albero che conseguentemente non contiene piú invocazioni di funzione, e per i motivi esposti precedentemente termina sempre, assumendo che tutte le funzioni siano lineari.

Osservazione 3.2. *In realtà viene unito un numero superiore di funzioni rispetto al numero minimo previsto dalla teoria. Precisamente, se la partizione ha dimensione k e il rappresentante canonico è in posizione i, vengono*

aggiunte $k - i$ funzioni in più, le quali sono le ultime contenute all'interno della forma normale, ma di cui non ci sarebbe bisogno perché alla forma normale viene prefissa la partizione espansa dal rappresentante in posizione i. La precisione della tecnica non cambia, e nemmeno la complessità computazionale se non nella costante moltiplicativa, dato che al più vengono aggiunte k funzioni nel caso peggiore, ovvero quando il rappresentante canonico è in posizione 0. Il vantaggio consiste nel dover calcolare solo una volta la partizione in forma normale, che sarà poi valida per tutti i membri della partizione stessa, variando solo la partizione espansa a cui essa verrà unita (la costruzione della partizione espansa è un'operazione molto meno onerosa in termini computazionali, dato che richiede al più k unioni, rispetto alla costruzione della forma normale, che ne richiede $k + \mathfrak{o}_P * 2$).

Esempio 3.4.7. *Si prenda un programma il cui grafo delle invocazioni è rappresentato nella Figura seguente:*

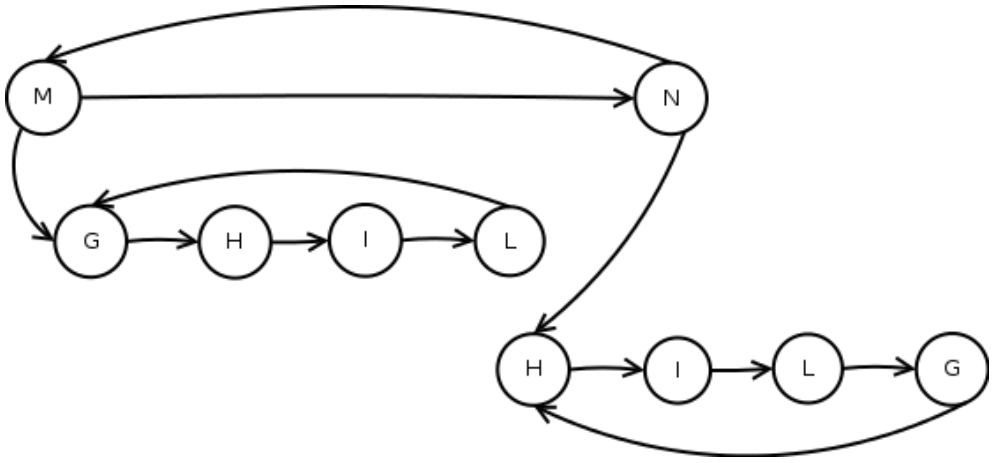


Figura 3.6: Grafo delle invocazioni

Le due sottopartizioni $[GHIL]$ e $[HILG]$ sono in realtà la stessa, cambiano solo i rappresentanti canonici da cui è invocata. Per saturare M usando il metodo sopra riportato, quando si incontra il figlio G si calcola la forma normale di $P = [GHIL]$ e la si unisce alla partizione espansa da G:

$$[GHIL] \cup (\underbrace{[GHIL \cup \cdots \cup \boldsymbol{GHIL}]}_{\mathfrak{o}_P * 2 \text{ volte}}[^0/_{G(\widetilde{x})}])$$

E quando si incontra il figlio H, la forma normale della partizione P è riutilizzata, unendola però alla partizione espansa da H:

$$[HIL] \cup (\underbrace{[GHIL \cup \cdots \cup G\boldsymbol{HIL}]}_{\mathtt{o}_P * 2 \text{ volte}}[^0/_{G(\widetilde{x})}])$$

Come già accennato, al costo di poche invocazioni eccedenti il numero teorico (evidenziate in grassetto), si è evitata la costruzione della forma normale della partizione $[HILG]$*.*

Dopo aver ottenuto un albero saturato, si può procedere all'estrazione degli stati presenti nello stesso. Se l'albero fosse in forma normale (Proposizione 2.2.1), cioè quando si ha $\mathtt{T} = \mathtt{T}_1\,;\ \cdots\ ;\,\mathtt{T}_n$, in cui ogni $\mathtt{T}_i$ è un parallelo tra dipendenze, si avrebbero n stati, dove lo stato i risulta composto esclusivamente dalle relazioni di dipendenza contenute in $\mathtt{T}_i$. Quando l'albero non è in forma normale, ci si può ricondurre ad essa tramite un assioma dei programmi LAM, che potrebbe essere definito come assioma della moltiplicazione, il quale afferma che

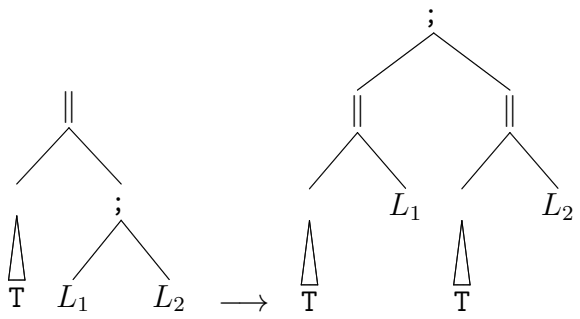
$$\mathtt{T} \| (\mathtt{L}'\,;\,\mathtt{L}'') = \mathtt{T} \| \mathtt{L}'\,;\,\mathtt{T} \| \mathtt{L}''$$

quando T non contiene invocazioni di funzione. Si noti come T venga duplicato, mentre le parentesi spariscono.

Si hanno dunque due possibilità:

- La prima consiste nell' effettuare una doppia visita, in cui nella prima l'albero viene portato in forma normale, mentre nella seconda viene tenuta traccia degli stati;
- Oppure eseguire una sola visita, lavorando sull'albero non in forma normale, tenendo però in conto semanticamente dell'assioma della moltiplicazione.

A ben vedere, l'approccio della trasformazione in forma normale è oneroso in termini di spazio, perché ad esempio, nel caso in cui T sia un sottoalbero complesso, che contiene sequenze di paralleli senza invocazioni di funzione, la sua duplicazione impone un utilizzo raddoppiato di memoria. Un esempio di duplicazione è il seguente::



Trasformare in forma normale quindi implica, oltre all'aumento dello spazio occupato dall'albero a causa delle duplicazioni (bilanciato in prima battuta da una duplicazione di puntatori rispetto ad una duplicazione di oggetti) anche l'aumento del tempo di visita dello stesso, perché quando si andrà a tenere traccia degli stati occorrerà visitare non solo il sottoalbero `T`, ma anche i suoi duplicati, di cui a priori non si conosce il numero.

L'altro approccio, cioè quello di lavorare sull'albero non in forma normale, è praticabile solamente impiegando una tecnica che utilizzi l'assioma per duplicare esclusivamente l'insieme di stati, e non gli interi sottoalberi.

L'operazione corretta da un punto di vista semantico, all'interno della situazione descritta dall'assioma, sarebbe quella di prodotto cartesiano: dato lo stato $\mathtt{T} \| (\mathtt{L}' \,;\, \mathtt{L}'')$, l'insieme di stati corretto, cioè quello generato tramite la duplicazione di `T`, è in realtà il prodotto cartesiano tra gli stati di `T` e quelli di $\mathtt{L}' \,;\, \mathtt{L}''$.

Esempio 3.4.8. *Sia* $S = (x,y) \| ((y,z) \,;\, (z,w))$. *Duplicando tramite l'assioma, si ha che* $S' = (x,y) \| (y,z) \,;\, (x,y) \| (z,w)$. S' *è in forma normale, e i suoi stati sono due:*

$$\{\ \underbrace{\{(x,y),(y,z)\}}_{\text{generato da (x,y)}\|\text{(y,z)}}\ ,\ \underbrace{\{(x,y),(z,w)\}}_{\text{generato da (x,y)}\|\text{(z,w)}}\ \}$$

Gli stessi identici stati possono essere ottenuti tramite l'operazione di prodotto cartesiano, partendo da S *che non è in forma normale:*

$$\underbrace{\{\{(x,y)\}\}}_{\text{generato da (x,y)}} \times \underbrace{\{\{(y,z)\},\{(z,w)\}\}}_{\text{generato da (y,z) ; (z,w)}} = \{\{(x,y),(y,z)\},\{(x,y),(z,w)\}\}$$

Formalmente, la visita dell'albero non in forma normale per tenere traccia degli stati in esso contenuti avviene tramite un operatore, chiamato *process* ed indicato con ω, che restituisce la lista di stati (dove uno stato è in realtà una lista di coppie) di un nodo qualunque:

(Pair)
$$\frac{S = (x,y)}{\omega(S) \to \{\{(x,y)\}\}}$$

(Parenthesis)
$$\frac{S = (L_1)}{\omega(S) \to \omega(L_1)}$$

(Semicolon)
$$\frac{S = L_1 \,;\, L_2}{\omega(S) \to \mathrm{cons}(\omega(L_1), \omega(L_2))}$$

(Parallel)
$$\frac{S = L_1 \| L_2}{\omega(S) \to \omega(L_1) \times \omega(L_2)}$$

Il caso base è rappresentato dalla regola (Pair), dove ω restituisce un insieme formato da un solo stato, il quale contiene esclusivamente la coppia del nodo in input. Il caso ricorsivo più semplice si ha con la regola (Parenthesis), in cui si applica ω al contenuto delle parentesi stesse. Se invece si è nel caso di una sequenza tra due nodi, tramite la regola (Semicolon) si restituisce la concatenazione tra il risultato dell'applicazione di ω al sottoalbero sinistro, e quello dell'applicazione al sottoalbero destro. In pratica, se il sottoalbero sinistro ha 5 stati e quello destro ne ha 3, verrà restituita una lista composta da 8 stati. L'ultimo caso è quello della regola (Parallel), in cui viene restituito il prodotto cartesiano tra i risultati ricorsivi sui due sottoalberi.

Avendo la lista degli stati in output da ω, si può procedere alla ricerca di circolarità all'interno degli stati. Si è deciso di utilizzare l'idea contenuta in [27], che si basa su una struttura dati (chiamata ST) contenente:

- Una *testa*, che corrisponde ad un elemento che si trova ad occupare la posizione di sinistra in almeno una coppia di dipendenze
- Una *lista* di elementi, i quali si trovano nella posizione destra nelle coppie in cui l'elemento di sinistra è la *testa*.

Passare dalla rappresentazione di stato come lista di coppie ad una lista di queste struttura dati è estremamente semplice, e inoltre si può evitare modificando opportunamente l'operatore *process*.

Esempio 3.4.9. *Si ha lo stato* $\{(x,y),(y,z),(z,x)\}$. *La struttura dati corrispondente per l'algoritmo di analisi di circolarità è*

$$\begin{array}{rl} x: & [y] \\ y: & [z] \\ z: & [x] \end{array}$$

Avendo invece $\{(x,y),(x,z),(y,w)\}$ *si ha*

$$\begin{array}{rl} x: & [y,z] \\ y: & [w] \end{array}$$

Per cercare una circolarità in uno stato, viene eseguito dunque l'Algoritmo 3, che lavora sulla lista di strutture dati (introdotte precedentemente e il cui tipo viene identificato da ST nello pseudocodice) derivate dallo stato stesso.

Nell'implementazione è stata utilizzata una versione leggermente ottimizzata dell'Algoritmo 3, che pur mantenendo la stessa complessità computazionale nel caso pessimo (quando cioè non ci sono circolarità) migliora leggermente le prestazioni in caso di deadlock. Infatti, alla linea 19, se si scopre

Algorithm 3 Ricerca di circolarità

```
    function checkCircularities(List<ST> state)
 2:     for each u in state do
            if find(u.head, u) then
 4:             return true
            end if
 6:         for each v in state do
                v.visited ← false
 8:         end for
        end for
10:     return false
    end function
12: function find(String head, ST u)
        for each element in u.tail do
14:         if head == element.head then
                return true
16:         end if
        end for
18:     for each element in u.tail do
            if size(element.tail) > 0 and not element.visited then
20:             element.visited ← true
                if find(head, element) then
22:                 return true
                end if
24:         end if
        end for
26:     return false
    end function
```

che *element.visited* è impostato a *true*, si è in presenza di una circolarità. Questa conclusione si può trarre solo se ogni chiamata ricorsiva di *FIND*, al suo termine, reimposti a *false* tutti i valori di *visited* che ha impostato a *true*, per evitare che un cammino in comune tra due teste, contenute nella stessa coda di qualche altra testa, seppur non circolare venga scambiato per circolare, a causa del fatto che il secondo trova già visitata una parte di esso (da parte del primo cammino). L'algoritmo viene invocato ad ogni saturazione di funzione, permettendo di conoscere all'interno di quale saturazione avviene il deadlock stesso. Come esempio, si è eseguita l'intera procedura di ricerca deadlock sul programma dell'Esempio 3.4.3. Viene segnalato correttamente un deadlock, nel caso specifico generato da $(x, y) \| H(y, x)$ nel corpo di M.

Capitolo 4

Il caso di programmi non lineari

Quando il programma non è lineare, non è possibile associare una mutazione univoca ad una funzione. Nel caso generale, la tecnica sviluppata per verificare la presenza di circolarità consiste nel trasformare un programma non lineare in uno lineare, e poi di eseguire l'algoritmo illustrato nel capitolo precedente. Come si vedrà, la trasformazione porta ad una perdita di precisione, introducendo coppie di dipendenze non presenti nel programma originario.

Nei programmi non lineari, le storie ricorsive non sono più adeguate per catturare le mutazioni definite dalle funzioni. Per esempio, nel seguente programma non lineare (chiamato $\mathtt{f'g'}$)

$$\Big(\mathtt{f}'(x,y,z) = (x,y) \| \mathtt{g}'(y,z),\ \mathtt{g}'(x,y) = \mathtt{g}'(x,z) \| \mathtt{f}'(z,y,y),\ \mathtt{L}\Big).$$

la storia ricorsiva di $\mathtt{f}'$ è $\mathtt{f'g'}$. La sequenza $\mathtt{f'g'g'}$ *non è* una storia ricorsiva perché contiene occorrenze multiple della funzione $\mathtt{g}'$. Dunque, se si calcolano le sequenze di invocazioni $\mathtt{f}'(x,y,z) \cdots \mathtt{f}'(\widetilde{u})$, è possibile derivare due sequenze $\mathtt{f}'(x,y,z)\mathtt{g}'(y,z)\mathtt{f}'(z',z,z)$ e $\mathtt{f}'(x,y,z)\mathtt{g}'(y,z)\ \mathtt{g}'(y,u)\mathtt{f}'(u',u,u)$ le quali definiscono due mutazioni differenti, nella fattispecie $[\![\, 4,3,3 \,]\!]$ e $[\![\, 6,5,5 \,]\!]$.

Definizione 4.1. Un programma $\Big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\Big)$ è *pseudo-lineare* se, per ogni $\mathtt{f}_i$, l'insieme di funzioni $\{\mathtt{f} \mid \mathit{closure}(\mathtt{f}) = \mathit{closure}(\mathtt{f}_i)\}$ contiene al più una funzione con un numero di storie ricorsive maggiore di 1.

Il programma $\mathtt{f'g'}$ definito precedentemente è pseudo-lineare. Il programma dell'esempio 3.1.2 della sezione precedente, e il seguente programma $\mathtt{l}'$

$$\Big(\mathtt{l}'(x,y,z) = (x,y) \| \mathtt{l}'(y,z,x)\,;\,(x,u) \| \mathtt{l}'(u,u,y),\ \mathtt{L}\Big)$$

sono altresì pseudo-lineari. In questi casi, le funzioni hanno un'unica storia ricorsiva, ma con invocazioni ricorsive multiple. Al contrario, il programma non lineare $\mathtt{f''g''}$ definito come

$$\left(\begin{array}{l} \mathtt{f}''(x,y) = (x,z) \| \mathtt{f}''(y,z)\,;\,\mathtt{g}''(y,x)\ , \\ \mathtt{g}''(x,y) = (y,x) \| \mathtt{f}''(y,z) \| \mathtt{g}''(z,x)\ , \\ \mathtt{f}''(x_1,x_2) \end{array}\right)$$

non è pseudo-lineare, perché entrambe le funzioni hanno più storie ricorsive.

La nozione di pseudo-linearità è stata introdotta per la facilità con cui programmi pseudo-lineari si possono trasformare in programmi lineari: essa avviene in tre passaggi, specificati in Tabella 4.1, che verranno discussi di seguito. Sia $\Big(\mathtt{f}_1(\widetilde{x_1}) = \mathtt{L}_1, \cdots, \mathtt{f}_\ell(\widetilde{x_\ell}) = \mathtt{L}_\ell, \mathtt{L}\Big)$ un programma LAM, e sia $\mathit{rechis}(\mathtt{f}_i)$ l'insieme delle storie ricorsive di $\mathtt{f}_i$, $\mathit{head}(\varepsilon) = \varepsilon$ e $\mathit{head}(\mathtt{f}\alpha) = \mathtt{f}$.

$$\frac{\begin{array}{c} \mathit{rechis}(\mathtt{f}_i) = \{\mathtt{f}_i\mathtt{f}_k\alpha, \mathtt{f}_i\beta_0, \cdots, \mathtt{f}_i\beta_n\} \qquad \{\mathit{head}(\beta_0), \cdots, \mathit{head}(\beta_n)\} \setminus \mathtt{f}_k \neq \varnothing \\ \mathtt{L}_i = \mathfrak{L}[\mathtt{f}_k(\widetilde{u})] \qquad \mathit{var}(\mathtt{L}_k) \setminus \widetilde{x_k} = \widetilde{z} \qquad \widetilde{w} \text{ sono fresh} \end{array}}{\big(\cdots \mathtt{f}_i(\widetilde{x_i}) = \mathtt{L}_i, \cdots, \mathtt{L}\big) \overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1 \big(\cdots \mathtt{f}_i(\widetilde{x_i}) = \mathfrak{L}[\mathtt{L}_k[\widetilde{w}/\widetilde{z}][\widetilde{u}/\widetilde{x_i}]], \cdots, \mathtt{L}\big)}$$

$$\frac{\begin{array}{c} \mathit{rechis}(\mathtt{f}_i) = \{\mathtt{f}_i\alpha\} \qquad \mathtt{f}_k = \mathit{head}(\alpha) \\ \mathtt{L}_i = \mathfrak{L}[\mathtt{f}_k(\widetilde{u_0})] \cdots [\mathtt{f}_k(\widetilde{u_{n+1}})] \qquad \mathtt{f}_k \notin \mathfrak{L} \\ \mathit{var}(\mathtt{L}_k) \setminus \widetilde{x_k} = \widetilde{z} \qquad \widetilde{w_0}, \cdots, \widetilde{w_{n+1}} \text{ sono fresh} \\ \mathfrak{L}[\mathtt{L}_k[\widetilde{w_0}/\widetilde{z}][\widetilde{u_0}/\widetilde{x_k}]] \cdots [\mathtt{L}_k[\widetilde{w_{n+1}}/\widetilde{z}][\widetilde{u_{n+1}}/\widetilde{x_k}]] = \mathtt{L}'_i \end{array}}{\big(\cdots \mathtt{f}_i(\widetilde{x_i}) = \mathtt{L}_i, \cdots, \mathtt{L}\big) \overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2 \big(\cdots \mathtt{f}_i(\widetilde{x_i}) = \mathtt{L}'_i, \cdots, \mathtt{L}\big)}$$

$$\frac{\begin{array}{c} \mathtt{L}_i = \mathfrak{L}[\mathtt{f}_i(\widetilde{u_0})] \cdots [\mathtt{f}_i(\widetilde{u_{n+1}})] \qquad \mathtt{f}_i \notin \mathfrak{L} \\ \mathit{var}(\mathtt{L}_i) \setminus \widetilde{x_i} = \widetilde{z} \qquad \widetilde{w_0}, \cdots, \widetilde{w_{n+1}} \text{ sono fresh} \\ \mathtt{L}_i^{aux} = \mathtt{f}_i^{aux}(\widetilde{u_0}[\widetilde{w_0}/\widetilde{x_i}], \cdots, \widetilde{u_{n+1}}[\widetilde{w_{n+1}}/\widetilde{x_i}]) \| \prod_{j\in 0..n+1} \flat_{\mathtt{f}_i}(\mathtt{L}_i)[\widetilde{w_j}/\widetilde{x_i}] \end{array}}{\begin{array}{l} \big(\cdots \mathtt{f}_i(\widetilde{x_i}) = \mathtt{L}_i, \cdots, \mathtt{L}\big) \overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3 \\ \quad \big(\cdots \mathtt{f}_i(\widetilde{x_i}) = \mathtt{f}_i^{aux}(\underbrace{\widetilde{x_i}, \cdots, \widetilde{x_i}}_{n+2 \text{ volte}}),\ \mathtt{f}_i^{aux}(\widetilde{w_0}, \cdots, \widetilde{w_{n+1}}) = \mathtt{L}_i^{aux}, \cdots, \mathtt{L}\big) \end{array}}$$

Tabella 4.1: Trasformazioni di programmi pseudo-lineari in lineari

Trasformazione $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1$*: Rimozione delle storie ricorsive multiple.* Come primo passaggio, si applica ripetutamente la regola $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1$. Ogni instanziazione della regola seleziona una funzione $\mathtt{f}_i$ con un numero di storie ricorsive maggiore di uno, attraverso le ipotesi $\mathit{rechis}(\mathtt{f}_i) = \{\mathtt{f}_i\mathtt{f}_k\alpha, \mathtt{f}_i\beta_0, \cdots, \mathtt{f}_i\beta_n\}$ e $\{\mathit{head}(\beta_0), \cdots, \mathit{head}(\beta_n)\} \setminus \mathtt{f}_k \neq \varnothing$ espandendo l'invocazione di $\mathtt{f}_k$, con $\mathtt{f}_k \neq \mathtt{f}_i$. Da definizione di pseudo-linearità, le altre funzioni in $\mathit{rechis}(\mathtt{f}_i)$ hanno una sola storia ricorsiva, da cui segue che ad ogni applicazione della regola a $\mathtt{f}_i$, la somma delle lunghezze delle sue storie diminuisce, fino al punto in cui espandendo la sua ultima invocazione (mutuamente) ricorsiva si rende unica la storia ricorsiva associata. L'espansione avviene grazie all'operatore di *merge*, trattato nella Definizione 3.4. Per esempio, il programma

$$\Big(\mathtt{f}(x) = (x, y) \| \mathtt{g}(x)\ ,\ \mathtt{g}(x) = (x, y) \| \mathtt{f}(x)\,;\, \mathtt{g}(y)\ ,\ \mathtt{L}\ \Big)$$

è trasformato in

$$\Big(\mathtt{f}(x) = (x, y) \| \mathtt{g}(x)\ ,\ \mathtt{g}(x) = (x, y) \| (x, z) \| \mathtt{g}(x)\,;\, \mathtt{g}(y)\ ,\ \mathtt{L}\ \Big)$$

espandendo all' interno del corpo di $\mathtt{g}$ la chiamata ad $\mathtt{f}(x)$, rimpiazzando opportunamente le variabili libere e quelle legate.

Transformazione $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2$*: Riduzione delle storie delle funzioni pseudo-lineari.* Attraverso $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_1$, ci si è ridotti a funzioni che hanno una sola storia ricorsiva, che non è ancora abbastanza per rendere un programma lineare. Il programma $\mathtt{l}'$ oppure il seguente programma $\mathtt{h}''\mathtt{l}''$

$$\left(\begin{array}{l} \mathtt{h}''(x,y) = (x,z) \| \mathtt{l}''(y,z)\,;\, \mathtt{l}''(y,x)\ , \\ \mathtt{l}''(x,y) = (y,x) \| \mathtt{h}''(y,z) \| \mathtt{h}''(z,x)\ , \\ \mathtt{h}''(x_1,x_2) \end{array}\right)$$

non sono ancora lineari, perché i corpi hanno più invocazioni di funzione appartenenti alla loro stessa partizione, che portano quindi a diverse invocazioni mutuamente ricorsive. La regola $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2$ espande i corpi delle funzioni pseudo-lineari fino a quando non è rimasta al più una sola invocazione mutuamente ricorsiva.

Per le ipotesi della regola, essa si applica alle funzioni $\mathtt{f}_i$ nelle quali il secondo elemento della storia ricorsiva è $\mathtt{f}_k$ (per definizione di storia ricorsiva, $\mathtt{f}_i \neq \mathtt{f}_k$) e il corpo di $\mathtt{f}_i$ contiene *almeno* due invocazioni di $\mathtt{f}_k$. Nelle conclusioni si trasforma dunque il corpo, espandendo ogni invocazione di $\mathtt{f}_k$ tenendo conto della corretta instanziazione delle variabili legate e libere.. Per esempio, le funzioni $\mathtt{h}''$ e $\mathtt{l}''$ nel programma $\mathtt{h}''\mathtt{l}''$ sono trasformate in

$$\begin{array}{ll} \mathtt{h}''(x,y) = & (x,z) \| \mathtt{l}''(y,z)\,;\, \mathtt{l}''(y,x)\ , \\ \mathtt{l}''(x,y) = & (y,x) \| ((y,z') \| \mathtt{l}''(z,z')\,;\, \mathtt{l}''(z,y)) \| ((z,z'') \| \mathtt{l}''(x,z'')\,;\, \mathtt{l}''(x,z)). \end{array}$$

Si noti come in output si abbiano delle funzioni nella stessa forma di `fib`, in cui cioè la mancanza di linearità è dovuta alla presenza di invocazioni ricorsive multiple.

Trasformazione $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$*: Rimozione delle invocazioni ricorsive non lineari.* Attraverso $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_2$ ci si è ridotti a programmi pseudo-linari in cui la non linearità è dovuta a funzioni ricorsive, ma non mutuamente ricorsive (come `fib`). La trasformazione successiva, e cioè $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$, rimuove le invocazioni ricorsive multiple dai programmi non lineari. Essa è anche responsabile dell'introduzione di inaccuratezze (e di conseguenza della perdita di precisione della tecnica), data dall'introduzione di dipendenze fasulle, non presenti nel programma originale. All'interno della regola $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$ si utilizza l'operatore ausiliario $\flat_{\mathtt{f}}(\mathtt{L})$, chiamato *flattening* e definito come segue:

$$\begin{array}{ll} \flat_{\mathtt{f}}(\mathtt{0}) = \mathtt{0}, & \flat_{\mathtt{f}}((x,y)) = (x,y), \\ \flat_{\mathtt{f}}(\mathtt{f}(\widetilde{x})) = \mathtt{0}, & \flat_{\mathtt{f}}(\mathtt{g}(\widetilde{x})) = \mathtt{g}(\widetilde{x}),\ \text{se } (\mathtt{f} \neq \mathtt{g}), \\ \flat_{\mathtt{f}}(\mathtt{L} \| \mathtt{L}') = \flat_{\mathtt{f}}(\mathtt{L}) \| \flat_{\mathtt{f}}(\mathtt{L}'), & \flat_{\mathtt{f}}(\mathtt{L}\,;\,\mathtt{L}') = \flat_{\mathtt{f}}(\mathtt{L})\,;\,\flat_{\mathtt{f}}(\mathtt{L}'). \end{array}$$

Analizzando $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$, si vede come essa seleziona una funzione $\mathtt{f}_i$ con invocazioni ricorsive multiple, ed estraendole attraverso il termine $\flat_{\mathtt{f}_i}(\mathtt{L}_i)$, che viene messo in parallelo con una invocazione ad una funzione ausiliaria, chiamata $\mathtt{f}_i^{aux}$, che colleziona gli argomenti di ciascuna invocazione ricorsiva $\mathtt{f}_i$ (con opportune instanziazioni dei nomi). Il termine risultante, chiamato $\mathtt{L}_i^{aux}$ è il corpo della nuova funzione $\mathtt{f}_i^{aux}$, invocata da $\mathtt{f}_i$ nel programma trasformato. Per esempio, la funzione di fibonacci è trasformata in

$$\begin{array}{l}\mathtt{fib}(x) = \mathtt{fib}^{aux}(x,x),\\ \mathtt{fib}^{aux}(x,x') = (x,y)\|(x,z)\|(x',y)\|(x',z)\|\mathtt{fib}^{aux}(y,z)\end{array}$$

dove le differenti invocazioni ($\mathtt{fib}(y)$ e $\mathtt{fib}(z)$) nel programma originale sono state contratte nell'unica invocazione della funzione ausiliaria ($\mathtt{fib}^{aux}(y,z)$). Come accennato in precedenza, contraendo le invocazioni ricorsive in un'unica invocazione, i nomi creati diminuiscono. Aumentano invece le dipendenze, introdotte per "simulare" le diverse chiamate ricorsive e per garantire quindi la correttezza dell'analisi. È possibile che le inaccuratezze introdotte da $\overset{\mathtt{pl}\mapsto\mathtt{l}}{\Longmapsto}_3$ possano essere ridotte utilizzando diverse tecniche, ma agli autori dell'articolo sembra che non possano essere eliminate del tutto a causa del problema della cardinalità: la valutazione di una invocazione $\mathtt{f}(\widetilde{u})$ in un programma lineare può produrre al più una invocazione di $\mathtt{f}$, mentre una invocazione di $\mathtt{f}(\widetilde{u})$ in un programma non lineare ne può produrre due, o addirittura di più.

Eseguendo il programma di trasformazione con input

$$\begin{array}{l}M(x)= \quad (x,y)||N(x) \quad ,\\ N2(x)=(x,y)||M2(x);N2(y),\\ M(x_1)\end{array}$$

Come output si ha

```
First transformation on N2(x)=(x,y)||M2(x);N2(y)
Result: N2(x)=(x,y)||((x,x1)||N2(x));N2(y)
Third transformation on N2(x)=(x,y)||((x,x1)||N2(x));N2(y)
Result: N2(x)=NAUX2(x,x)
NAUX2(x,y)=(x,x1)||((x,x2)||0);0||(y,x1)||((y,x2)||0);0||NAUX2(x,x2)
```

E l'unica partizione che viene trovata contiene esclusivamente $NAUX2$.

Capitolo 5

Conclusioni

Si è definita una tecnica per l'analisi statica di deadlock e la relativa implementazione. I maggiori benefici che si possono trarre dall'utilizzo di essa sono (i) non c'è necessità di usare alcun ordine pre-definito (ii) gestisce perfettamente la creazione dinamica di nomi e infine (iii) grazie all'implementazione tramite libreria è possibile inserirla dovunque ce ne sia la necessità.

Attualmente si sta testando l'intero lavoro su programmi scritti in un linguaggio orientato agli oggetti con tipi di dato futuri [4, 3].

Per la teoria, esistono diverse direzioni di ricerca:

1. I modelli LAM e la teoria delle mutazioni è stata usata per scoprire i *deadlock di allocazione di risorse*, così come sono visti ad esempio nei sistemi operativi. D'altro canto, la tecnica sembra adeguata per gestire anche i deadlock dovuti alla sincronizzazione di processi, come quelli in pi-calcolo [17, 5]. In questo caso, i nomi sono *nomi di canale*, ed esiste già un prototipo di analizzatore per i deadlock, che usa però una tecnica differente [28].

2. La tecnica è stata impiegata per analizzare un linguaggio basilare, orientato agli oggetti, con tipi di dato futuri. Per questo linguaggio si è definito un sistema di inferenza [4], ma la sua estensione per supportare altri costrutti, come assegnamenti, strutture dati ed ereditarietà non è banale e comporta ulteriori problemi.

Per quanto riguarda l'implementazione, l'unico passo non ancora trattato (ma in via di sviluppo) è relativo al caso generale, quando cioè un programma LAM non è lineare nè pseudo-lineare. Esiste, ed è dimostrata corretta, una trasformazione che, dato in input un programma non pseudo-lineare, ne restituisce la versione pseudo-lineare in output.

Bibliografia

[1] E. Giachino and C. Laneve, "Mutations, flashbacks and deadlocks," 2013, submitted. Available at www.cs.unibo.it/~laneve.

[2] L. Comtet, *Advanced Combinatorics: The Art of Finite and Infinite Expansions*. Netherlands: Dordrecht, 1974.

[3] E. Giachino and C. Laneve, "A beginner's guide to the deadLock Analysis Model," in *TGC*, ser. LNCS. Springer-Verlag, 2013.

[4] E. Giachino, C. A. Grazia, C. Laneve, M. Lienhardt, and P. Y. H. Wong, "Deadlock analysis of concurrent objects: Theory and practice," 2013, submitted. Available at www.cs.unibo.it/~laneve.

[5] N. Kobayashi, "A new type system for deadlock-free processes," in *CONCUR*, ser. LNCS, vol. 4137. Springer, 2006, pp. 233–247.

[6] K. Suenaga and N. Kobayashi, "Type-based analysis of deadlock for a concurrent calculus with interrupts," in *ESOP*, ser. LNCS. Springer, 2007, vol. 4421, pp. 490–504.

[7] K. Suenaga, "Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references," in *APLAS*, ser. LNCS. Springer, 2008, vol. 5356, pp. 155–170.

[8] V. T. Vasconcelos, F. Martins, and T. Cogumbreiro, "Type inference for deadlock detection in a multithreaded polymorphic typed assembly language," in *PLACES*, ser. EPTCS, vol. 17, 2009, pp. 95–109.

[9] C. Boyapati, R. Lee, and M. Rinard, "Ownership types for safe program.: preventing data races and deadlocks," in *OOPSLA*. ACM, 2002, pp. 211–230.

[10] C. Flanagan and S. Qadeer, "A type and effect system for atomicity," in *PLDI*. ACM, 2003, pp. 338–349.

[11] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for Java," *TOPLAS*, vol. 28, 2006.

[12] N. Kobayashi, "A partially deadlock-free typed process calculus," *TOPLAS*, vol. 20, no. 2, pp. 436–482, 1998.

[13] A. Igarashi and N. Kobayashi, "A generic type system for the pi-calculus," *Theor. Comput. Sci.*, vol. 311, no. 1-3, pp. 121–163, 2004.

[14] C. Laneve and L. Padovani, "The *must* preorder revisited," in *CONCUR*, ser. LNCS, vol. 4703. Springer, 2007, pp. 212–225.

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

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

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

[18] H. R. Nielson and F. Nielson, "Higher-order concurrent programs with finite communication topology," in *POPL*. ACM, 1994, pp. 84–97.

[19] S. Chaki, S. K. Rajamani, and J. Rehof, "Types as models: model checking message-passing programs," *SIGPLAN Not.*, vol. 37, no. 1, pp. 45–57, 2002.

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

[21] S. P. Masticola, "Static detection of deadlocks in polynomial time," Ph.D. dissertation, 1993.

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

[23] E. Giachino and C. Laneve, "Deadlock and livelock analysis in concurrent objects with futures," www.cs.unibo.it/~laneve, 2012.

[24] E. B. Johnsen and O. Owe, "An asynchronous communication model for distributed concurrent objects," *Software and System Modeling*, vol. 6, no. 1, pp. 39–58, 2007.

[25] A. Welc, S. Jagannathan, and A. Hosking, "Safe futures for Java," in *OOPSLA*. New York, NY, USA: ACM, 2005, pp. 439–453.

[26] E. Gagnon, "Sablecc, an object-oriented compiler framework," 1998, lavoro di tesi su SableCC, reperibile su http://sablecc.sourceforge.net/downloads/thesis.pdf.

[27] C. A. Grazia, "Analisi statica dei deadlock in featherweight java con futuri," 2013, lavoro di tesi presso Università di Bologna.

[28] N. Kobayashi, "Type systems for concurrent programs," in *10th Anniversary Colloquium of UNU/IIST*, ser. LNCS, vol. 2757. Springer, 2003, pp. 439–453.