ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE
Corso di Laurea Magistrale in Informatica

Macchine di Schönhage e Complessità Computazionale Implicita

Relatore:
Chiar.mo Prof.
Simone Martini

Presentata da:
Domiziana Suprani

Sessione III
Anno Accademico 2012-2013

"I never am really satisfied that I understand anything;
because, understand it well as I may, my comprehension can only be
an infinitesimal fraction of all I want to understand
about the many connections and relations which occur to me,
how the matter in question was first thought of or arrived at, etc., etc."
— Ada Lovelace

"Toute autre science est dommageable
celui qui n'a pas la science de la bonté."
— Michel de Montaigne

Indice

1 Macchine di Schönhage **5**
1.1 Strutture Dati e Istruzioni 5
1.1.1 Δ-struttura 5
1.1.2 Programma e Istruzioni 7
1.1.3 Esempio di SMM 8
1.2 Alcuni Risultati di Simulazione 11
1.2.1 Simulazione in Tempo Reale di una Macchina di Turing 12
1.2.2 Moltiplicazione tra Interi in Tempo Lineare 13
1.3 Term Graph Rewriting e Macchine di Schönhage 14
1.3.1 Riscrittura su Term Graph 14
1.3.2 Strutture e Funzioni Ausiliarie per Macchine di Schönhage 16

2 Gurevich Abstract State Machine **23**
2.1 Algebre Statiche e Update 24
2.1.1 Definizione di Algebre Statiche 24
2.1.2 Termini 25
2.1.3 Locazioni e Update 25
2.1.4 Insiemi di Update e Famiglie di Insiemi di Update . . 26
2.2 Sequential Evolving Algebra 26
2.2.1 Basic Transition Rules 26
2.2.2 Importare Nuovi Elementi 28
2.2.3 Programmi e Run 29
2.3 Lock-step Equivalenza tra SMM e ASM 30
2.3.1 Simulazione Lock-step e Lock-step Equivalenza 30
2.3.2 Simulazione di una SMM Utilizzando una ASM Unaria 31
2.3.3 Simulazione di una ASM Tramite SMM 35

3 Evolving Graph Structures **39**
3.1 Evolving Graph Structures 40
3.1.1 Sorted Partial Structures 40
3.1.2 Strutture Grafo 40
3.1.3 Espressioni 41

3.1.4 Programmi 41
3.1.5 Evolving Structures 42
3.2 Programmi Ramificabili 44
3.2.1 Loop Stazionari e Strettamente Modificanti 45
3.3 Ramificazione Stretta e Polinomialità 46
3.3.1 Non-Interferenza 47
3.3.2 Limiti Polinomiali 48
3.3.3 Risultati Aggiuntivi 49

4 EGS e SMM: simulazione 51
4.1 Insiemi ed Elementi Base 51
4.2 Espressioni 56
4.2.1 Semantica delle Espressioni 57
4.3 Programmi 59
4.4 Run time: Confronto tra Strutture 61
4.4.1 Costruzione della Struttura 65

5 Tiering e SMM 67
5.1 Simulazione e Ramificazione 68
5.1.1 Tiering di Espressioni 68
5.1.2 Tiering di Programmi 69
5.1.3 Proprietà 70
5.2 Teorema di Caratterizzazione per SMM 71
5.2.1 Ramificazione Stretta Implica Polinomialità 73
5.2.2 Polinomialità Implica Ramificazione Stretta 77

Introduzione

Nel 1970 Arnold Schönhage descrisse per la prima volta le macchine a modificazione della memoria, dedicando allo stesso concetto studi più approfonditi negli anni successivi. Questo modello di computazione, che prende il nome del suo creatore, appartiene alla classe delle macchine a puntatori. Con alcune di esse, come le Macchine di Kolmogorov e le Macchine a Stati Astratti di Gurevich, condivide caratteristiche e similitudini. Nel corso degli anni il modello di Schönhage è stato parte di un numero consistente di studi sulla simulazione tra macchine, i cui risultati ne hanno sottolineato la flessibilità.

Oggetto della presente tesi è lo studio di un'ulteriore simulazione e delle sue conseguenze in termini di complessità computazionale. Elemento di interesse per questa trattazione è il contenuto di una recente pubblicazione di Leivant e Marion. In essa gli autori descrivono un nuovo modello, a cui si riferiscono con il nome di Evolving Graph Structures (in breve EGS), fortemente influenzato dalle Macchine a Stati Astratti. In questa tesi verrà dimostrato che per ogni EGS esiste una opportuna Macchina di Schönhage in grado di simularla, preservando l'eventuale polinomialità del tempo di esecuzione del programma sulla struttura. Le caratteristiche delle macchine a modificazione della memoria capaci di compiere tale simulazione permettono di ereditare una delle proprietà più interessanti delle EGS:

> *Una funzione su una struttura grafo è computabile in tempo polinomiale se e solo se è computabile da un programma terminante e strettamente ramificabile*

Nella prima sezione di questa tesi vengono presentati i modelli alla base dello studio effettuato. Il Capitolo 1 è dedicato unicamente alle Macchine di Schönhage. Nonostante esse abbiano fornito nel corso degli anni un solido punto di riferimento per l'analisi e la realizzazione di nuovi modelli, la documentazione originale che le riguarda è estremamente scarna. Oltre alla definizione formale di queste macchine e dei loro componenti, vengono descritti alcuni risultati di simulazione ritenuti interessanti. Infine, si riporta brevemente lo studio compiuto da chi scrive in merito al rapporto tra Macchine di Schönhage e term graph rewriting. Il Capitolo 2 descrive

le Macchine a Stati Astratti di Gurevich, la cui importanza ai fini di questa trattazione è duplice. Esse, infatti, non solo rappresentano l'ispirazione principale per il modello di Leivant e Marion, ma si legano strettamente alle macchine a modificazione della memoria a cui sono lock-step equivalenti. Nel Capitolo 3 vengono ripresi i contenuti dell'articolo che descrive lo studio delle Evolving Graph Structures, introducendole e ponendo particolare attenzione su alcuni interessanti teoremi riguardanti la loro complessità computazionale.

La seconda parte della trattazione presenta i risultati ottenuti nel corso del presente studio, descrivendo una modalità di simulazione delle EGS per mezzo di opportune Macchine di Schönhage ed alcuni teoremi ad essa applicabili. Il Capitolo 4 descrive le strutture ausiliarie e le sequenze di istruzioni su cui si basa la simulazione. La complessità del programma simulante viene quindi analizzata dal punto di vista del costo delle istruzioni utilizzate, dimostrando che la polinomialità dell'esecuzione viene preservata. Infine, il Capitolo 5 fornisce una definizione dei concetti di tiering applicati alle SMM, individuando un sottoinsieme di macchine su cui vengono dimostrati teoremi ed assuzioni di complessità già validi per le EGS.

Capitolo 1

Macchine di Schönhage

Il concetto di macchina a modificazione della memoria (Storage Modification Machine, SMM in seguito) fu presentato per la prima volta da Arnold Schönhage nel 1970 [01] come modello generale di computazione. Dieci anni dopo, lo stesso autore dedicò un articolo intero allo studio più completo di queste macchine [02] nel quale si occupò non solo di descrivere le caratteristiche del suo modello, ma anche di fornire una prova dettagliata della possibile simulazione in tempo reale di una Macchina di Turing multidimensionale [04]. Lo stesso Schönhage sottolineò come le sue macchine godessero di estrema flessibilità e potessero pertanto fungere da base per lo studio di un'adeguata nozione di complessità in termini di tempo per problemi algoritmici, che fino a quel momento mancava di uniformità. I seguenti paragrafi illustreranno la struttura delle macchine a modificazione della memoria e i principali risultati su questo modello ottenuti precedentemente da chi scrive [03].

1.1 Strutture Dati e Istruzioni

Una macchina a modificazione della memoria è identificata dai seguenti componenti: una Δ-struttura, un programma, una stringa sequenziale in input (genericamente binaria) e una stringa sequenziale in output. Questa sezione si occupa di descrivere le varie componenti e le relazioni che intercorrono tra esse.

1.1.1 Δ-struttura

Le macchine di Schönhage si basano su una particolare struttura dati dinamica, detta Δ-struttura, che prende il nome dall'alfabeto Δ associato ad ogni singola macchina. Δ è un inseme finito i cui elementi possono essere considerati come valori di direzione. Essi sono i corrispondenti delle istruzioni di spostamento delle testine nelle macchine di Turing - ad esempio

{N, W, S, E} su un piano o {R, L} su nastro. Una Δ-struttura è un grafo orientato finito i cui archi sono etichettati con elementi di Δ. Esiste una corrispondenza biunivoca tra gli archi uscenti da ogni nodo della struttura e gli elementi dell'alfabeto Δ, dal momento che ogni elemento viene usato una ed una sola volta come etichetta del relativo arco uscente da ognuno dei nodi; per questa ragione il numero di archi uscenti da ogni nodo è fisso per ogni particolare macchina ed è uguale alla dimensione $\mid \Delta \mid$ dell'alfabeto.
In ogni grafo è inoltre presente un particolare nodo scelto, detto *centro* della struttura. Esso rappresenta il punto di accesso per il mondo esterno, diventando quindi l'analogo delle testine delle macchine di Turing. A differenza delle celle di queste ultime, che come è noto contengono un singolo simbolo dell'alfabeto della macchina, i nodi delle Δ-strutture non forniscono alcun genere di informazione. L'informazione è espressa mediante i possibili diversi pattern realizzabili con gli archi della struttura.

Definizione 1.1 (Δ-struttura)**.** Una Δ-struttura è una tripla $S = (X, a, p)$, dove X è un insieme finito di nodi, $a \in X$ è il centro di S e $p = (\{p_\alpha\} \mid \alpha \in \Delta)$ è un insieme di funzioni $X \to X$ indicizzato dagli elementi di Δ. In particolare $p_\alpha(x) = y$ indica che il puntatore con etichetta α che parte dal nodo x si dirige verso il nodo y. È possibile quindi considerare il centro come destinazione di un arco aggiuntivo proveniente dall'esterno.

Si indichi con Δ^* l'insieme delle parole sull'alfabeto Δ e con $\Box$ la parola vuota. Per ogni Δ-struttura $S = (X, a, p)$, si definisce la mappatura $p^* : \Delta^* \to X$ in maniera ricorsiva:

- $p^*(\Box) = a$
- $p^*(W\alpha) = p_\alpha(p^*(W))$ per ogni $\alpha \in \Delta, W \in \Delta^*$

La funzione p^* associa quindi parole di Δ^* ad elementi di X, indicando il percorso di archi necessario a raggiungerli. Si noti come i casi in cui l'immagine di $p^*(\Delta^*)$ sia un sottoinsieme proprio di X indichino la presenza di una parte della struttura non accessibile a partire dal centro. Saranno dunque ammesse solo strutture che garantiscono la piena accessibilità ad ognuno dei nodi (cioè tali che $p^*(\Delta^*) = X$).
A questo punto è possibile introdurre una relazione di equivalenza in Δ^* definendola nella seguente maniera:

$$U \sim V \Leftrightarrow p^*(U) = p^*(V)$$

da cui deriva l'ovvia proprietà:

$$U \sim V \Rightarrow UW \sim VW$$

per ogni $U, V, W \in \Delta^*$. Si noti come ciò sia riconducibile al teorema di Myhill-Nerode: dato un linguaggio L definiamo una relazione R_L su Σ^* come segue:

$$xR_Ly \leftrightarrow (\forall z \in \Sigma^* : xz \in L \leftrightarrow yz \in L)$$

La relazione d'equivalenza R_L è una relazione di congruenza. Per il teorema di Myhill-Nerode, $L \subseteq \Sigma^*$ è regolare sse R_L ha indice finito.
Infine, ogni relazione di equivalenza finita su Δ^* per cui vale questa ultima proprietà può essere a sua volta vista come una Δ-struttura: i nodi corrispondono alle classi di equivalenza, la classe contenente $\Box$ è il centro e per mezzo dell'isomorfismo ogni Δ-struttura può essere rappresentata in questo modo.

1.1.2 Programma e Istruzioni

La memoria di una SMM con alfabeto interno Δ è rappresentata da una relativa Δ-struttura accessibile dal suo centro. Essa viene manipolata per mezzo di un programma di controllo finito, scritto in un linguaggio formale simile ad ALGOL. Il programma legge dalla stringa in input e scrive sulla stringa in output in maniera sequenziale. Lo stato di una SMM è descritto in ogni istante dall'input rimanente, dall'output precedentemente scritto, dall'istruzione corrente e dalla Δ-struttura. Nello stato iniziale, l'input rimanente è uguale alla stringa di input completa, l'output accumulato è nullo e l'istruzione corrente è la prima del programma. La Δ-struttura iniziale contiene il singolo nodo centrale a, i cui archi uscenti puntano tutti ad a stesso.
Per quanto riguarda il programma di controllo, esso può essere visto come una sequenza di etichette e istruzioni. Le etichette, scritte in maniera analoga a quelle del linguaggio ALGOL, si possono utilizzare per trasferire il controllo da un punto all'altro del programma tramite istruzioni come `goto`. Se due istruzioni posseggono la stessa etichetta, la prima viene considerata come l'unica avente tale nome di riferimento. Ogni SMM permette l'utilizzo di due principali categorie di istruzioni: le istruzioni comuni, che sono uguali per ogni macchina presa in considerazione, e le istruzioni interne, il cui significato dipende dal particolare Δ considerato. Le istruzioni comuni sono le seguenti:

- `input` λ_0 λ_1; viene letto dalla stringa in input il primo bit $\beta \in \{0, 1\}$. A seconda del valore letto, il controllo viene trasferito all'istruzione avente etichetta λ_β. Se l'input è vuoto, il controllo passa all'istruzione successiva del programma;
- `output` β; dato $\beta \in \{0, 1\}$, esso viene stampato sulla stringa in output;
- `goto` λ; il controllo viene trasferito all'istruzione etichettata λ;
- `halt`; la macchina smette di funzionare. Ciò avviene anche nel caso in cui il controllo passi alla fine del programma.

Le istruzioni interne sono invece le seguenti:

- `new` W; data la struttura $S = (X, a, p)$, questa istruzione la modifica trasformandola in $S' = (X', a', p')$, con $X' = X \cup \{y\}$; viene cioè creato

un nuovo nodo y, la cui posizione rispetto agli altri nodi e archi dipende da $W \in \Delta^*$. In particolare, distinguiamo i due seguenti casi:

– $W = \Box$; $a' = y$ e $p'_\delta(y) = a$ per ogni $\delta \in \Delta$. Si tratta del caso base. Se la parola da creare è vuota viene aggiunto un nuovo nodo centrale i cui puntatori sono tutti diretti verso il suo predecessore. Gli altri puntatori della struttura non subiscono modifiche;

– $W = U\alpha$ con $U \in \Delta^*$ e $\alpha \in \Delta$; $a' = a$, $p'_\alpha(p^*(U)) = y$ e $p'_\delta(y) = p^*(W)$ per ogni $\delta \in \Delta$. In altre parole, i puntatori del nuovo nodo portano a quello che era precedentemente descritto da W (solitamente il nodo centrale, ma non sempre). Ogni altro puntatore non viene modificato.

Si noti la natura iterativa di questa definizione: se il nuovo nodo da inserire è rappresentato da una stringa di caratteri esso viene posizionato al termine del percorso di puntatori etichettati con i singoli caratteri della stringa;

- `set` W `to` V; data la struttura $S = (X, a, p)$, questa istruzione la modifica trasformandola in $S' = (X', a', p')^*$, causando l'assegnamento di un puntatore. L'asterisco indica che S' è ottenuto tramite riduzione della parte di X' e p' accessibile da a'. X' rimane uguale a X, mentre la scelta del puntatore e la sua direzione dipendono da W e $V \in \Delta^*$. Anche per questa istruzione si distinguono due casi:

  * $W = \Box$; $a' = p^*(V)$, cioè il nodo indicato da V diventa il nuovo centro, mentre p' rimane invariato;

  * $W = \alpha$; $a = a'$ e $p_\alpha(p^*(U)) = p^*(V)$, cioè il puntatore con etichetta α uscente da U viene indirizzato sul nodo indicato da V, mentre gli altri puntatori non subiscono modifiche;

- `if` $U = V$ `then` σ e `if` $U \neq V$ `then` σ; σ è una istruzione diversa da un `if` che viene eseguita se e solo se $p^*(U) = p^*(V)$ e viceversa, rispettivamente.

1.1.3 Esempio di SMM

Le immagini riportate in seguito mostrano un esempio delle variazioni subite da una Δ-struttura in seguito all'esecuzione di alcune istruzioni. Sia $\Delta = \{0, 1\}$ e si supponga che inizialmente il grafo contenga soltanto il nodo centrale, indicato con la lettera A (come illustrato in Figura 1.1).

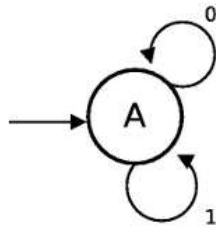

Fig. 1.1: Condizione iniziale

Tutti gli archi puntano al nodo centrale A, compreso il puntatore esterno che permette l'accesso alla struttura. Si vuole eseguire sulla macchina appena descritta il seguente programma:

```
new 1
new 10
new 11
set 111 to 10
```

Come si può intuire ad un primo sguardo, il programma in questione crea tre nuovi nodi, spostando infine l'arco uscente da uno di essi.

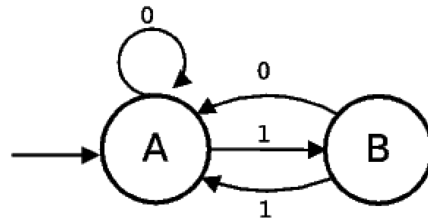

Fig. 1.2: Creazione di un nodo

La prima istruzione comporta la creazione di un nuovo nodo B, puntato da A per mezzo dell'arco etichettato con 1. Tutti gli archi uscenti da B puntano al nodo centrale, come mostrato nella Figura 1.2. Eseguendo la seconda istruzione viene creato il nodo indicato con C. Esso è codificato dalla stringa 10, che identifica il percorso compiuto partendo dal nodo centrale e spostandosi di arco in arco (Figura 1.3).

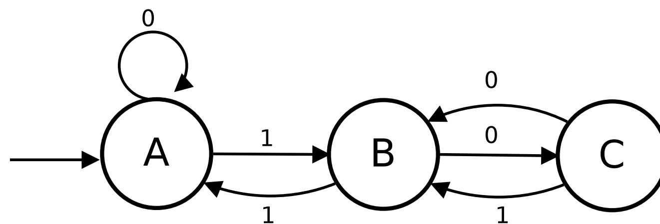

Fig. 1.3: Creazione di un nodo

In maniera analoga l'istruzione successiva crea il nodo D. Si noti come l'informazione rappresentata dalla struttura è totalmente costituita dalla disposizione dei puntatori. L'ultima istruzione è di tipo `set` e la sua esecuzione provoca lo spostamento di un puntatore: la sua destinazione cambia da B a C.

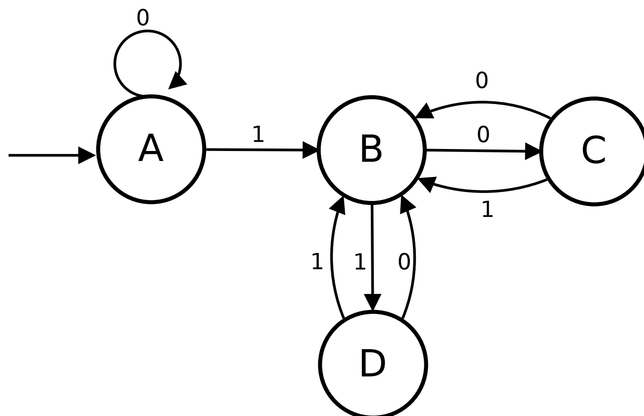

Fig. 1.4: Creazione di un nodo

Si può definire il tempo di esecuzione di un programma come il numero di istruzioni eseguite includendo `halt`. Lo stesso Schönhage fa notare che sebbene si possa ottenere una misura più precisa utilizzando pesi diversi per diversi tipi di istruzioni - ad esempio facendo valere `set` W `to` V come un numero di istruzioni pari a uno più la somma delle lunghezze di V e W - questo stratagemma farebbe aumentare il tempo di un fattore costante per ogni particolare programma, portando a un risultato concettualmente equivalente.

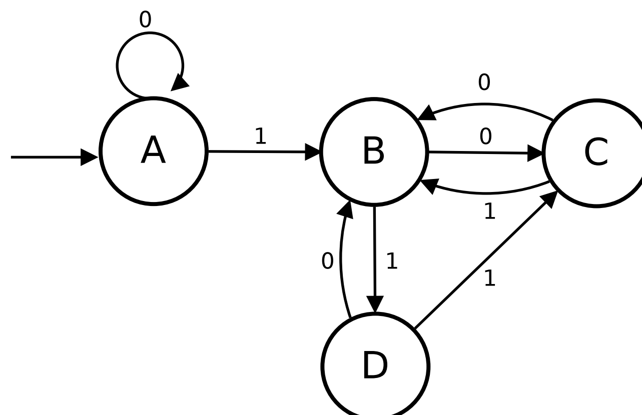

Fig. 1.5: Spostamento di un puntatore

1.2 Alcuni Risultati di Simulazione

Schönhage decise di introdurre questo nuovo modello alla luce del fatto che fino a quel momento non era stata stabilita "*una misura unificata e generalmente accettata per misurare la complessità in termini di tempo di problemi algoritmici*" [05]. Egli voleva quindi fornire un modello sufficientemente flessibile da servire come base per la simulazione in tempo reale di algoritmi sequenziali arbitrari. Sebbene non fosse stata suggerita dall'autore una definizione di flessibilità, Yuri Gurevich suppose che si riferisse alla capacità delle SMM di simulare in tempo reale ogni macchina sequenziale [06]. Rimane comunque da stabilire cosa si intenda per macchina sequenziale. Supponendo che, all'atto di paragonare modelli differenti di macchine, parlando di simulazione ci si limiti ad analizzare i comportamenti di input e output, Schönhage [02] suggerisce la seguente definizione di simulazione in tempo reale:

Definizione 1.2 (Simulazione in tempo reale)**.** Si dice che una macchina M' simula in tempo reale un'altra macchina M, e si denota con $M \xrightarrow{\tau} M'$, se esiste una costante c tale che per ogni sequnza in input x vale la seguente affermazione: se x causa a M la lettura di un simbolo in input, o la stampa di un simbolo in output, o l'halt agli istanti $0 = t_0 < t_1 < \ldots < t_l$ rispettivamente, allora x causerà a M' di agire nella stessa maniera rispetto alle stesse istruzioni esterne negli istanti $0 = t'_0 < t'_1 < \ldots < t'_l$, dove $t'_j - t'_{j-1} \leq c(t_j - t_{j-1})$ con $1 \leq j \leq l$.

Definizione 1.3 (Riducibilità in tempo reale)**.** Per ogni classe di macchine $\mathcal{M}$, $\mathcal{M}'$ la riducibilità in tempo reale (real time reducibility) $\mathcal{M} \xrightarrow{\tau} \mathcal{M}'$ è definita tramite la seguente condizione: per ogni M appartenente a $\mathcal{M}$ esiste una macchina M' appartenente a $\mathcal{M}'$ tale che $M \xrightarrow{\tau} M'$. Equivalenza

in tempo reale (real time equivalence) $\mathcal{M} \overset{\mathcal{T}}{\leftrightarrow} \mathcal{M}'$ significa che che $\mathcal{M} \overset{\mathcal{T}}{\rightarrow} \mathcal{M}'$ e $\mathcal{M}' \overset{\mathcal{T}}{\rightarrow} \mathcal{M}$.

Date queste definizioni è possibile formulare la principale tesi di Schönhage nella seguente maniera: per ogni modello di macchine atomistiche $\mathcal{M}$ vale che $\mathcal{M} \overset{\mathcal{T}}{\rightarrow}$ SMM. Amir M. Ben Amram fornì una chiara descrizione di cosa Schönhage indendesse per macchina atomistica [07]: l'insieme delle macchine astratte aventi le seguenti proprietà:

- Tutti i passi della computazione vengono compiuti in passi discreti e ogni passo utilizza solo una parte limitata del risultato delle precedenti operazioni;
- L'illimitatezza della memoria è solo quantitativa: si ipotizza l'esistenza di un numero illimitato di elementi, ma essi vengono costruiti basandosi su un insieme finito di tipi e le relazioni che intercorrono tra essi hanno complessità limitata.

L'insieme delle macchine atomistiche include tutte le macchine a puntatori, tra cui i vari tipi di RAM. La relativa dimostrazione della simulazione in tempo reale da parte delle macchine di Schönhage è disponibile nella pubblicazione dedicata al suo modello [02]. Si rimanda al secondo capitolo per la definizione di equivalenza lock-step e la dimostrazione di simulazione inerente le Abstract State Machine.

1.2.1 Simulazione in Tempo Reale di una Macchina di Turing

Interessante è anche il rapporto tra SMM e Macchine di Turing, analizzato da Schönhage stesso nel suo articolo sopra citato. In esso, viene preso in considerazione il particolare modello di macchina che consiste di una o più componenti finite a ognuna delle quali si accede tramite una o più testine. Inizialmente tutta la memoria è vuota e ogni testina è posizionata all'inizio del proprio componente. Viene utilizzato $\{0, 1\}$ come alfabeto input/output e la posizione delle testine può essere incrementata o decrementata di una sola posizione per passo. Il controllo è dato da una sequenza finita di istruzioni ed etichette. In aggiunta a input, output, goto e halt abbiamo le istruzioni head (per stabilire quale delle testine è attiva), write, read e move.

Volendo realizzare un simulatore di macchine di Turing (abbreviato con MdT), il problema principale che si pone riguarda la simulazione della classe di macchine con componenti di memoria multidimensionali. In particolare, si ricordi che ogni simbolo in input descrive un percorso su queste componenti utilizzando le direzioni finite dell'alfabeto relativo (comunemente {N, S, W, E}). Richiedere a una macchina che legge l'input sequenzialmente di simulare questo comportamento può presentare numerosi problemi. Come può, ad esempio, stabilire in tempo reale se la posizione corrente sia già

stata visitata o meno, e da quale testina? Questo quesito è conosciuto con il nome di *Self Crossing problem.*

Il teorema dimostrato da Schönhage in [02] è il seguente:

Teorema 1.2.1. *Ogni macchina di Turing appartenente alla classe appena descritta è simulabile da una idonea SMM in tempo reale.*

La dimostrazione, disponibile nell'articolo originale, si basa tre particolari strutture ausiliarie, realizzate per mezzo di una SMM il cui alfabeto è $\Delta = \{$N, S, W, E, U, D$\}$, dove U e D stanno per up and down:

- Struttura piramidale: memorizza il contenuto del piano della MdT, le informazioni riguardanti le celle che sono già state visitate e da quali testine;

- Contatori Cj: ve n'è presente uno per ogni testina ($j = 1, \ldots, n$). I contatori vengono utilizzati per accedere alla struttura piramidale;

- Struttura centrale H: contiene il centro della struttura A, i nodi base $B_1, B_2, \ldots, B_r$ e due nodi A_0 e A_1 usati per codificare l'informazione contenuta nel grafo. Serve per selezionare e accedere al contatore della testina attiva della MdT ogni volta che viene simulata una istruzione interna della stessa MdT.

1.2.2 Moltiplicazione tra Interi in Tempo Lineare

Per concludere si riporta ora un teorema riguardante la complessità di computazioni eseguite dal modello di macchine a modificazione della memoria, formulato da Schönhage stesso [02]:

Teorema 1.2.2. *Esiste una macchina a modificazione di memoria capace di eseguire moltiplicazioni tra interi in tempo lineare. Ciò significa che esiste una costante c tale che 2 numeri interi di N bit l'uno x e y letti in input in maniera consecutiva producono in output* $z = xy$ *dopo al massimo* cN *passi. Il numero z è espresso in forma binaria.*

In particolare viene dimostrato che:

Teorema 1.2.3. *Esiste una macchina a modificazione di memoria capace di eseguire moltiplicazioni tra interi in tempo lineare. Ciò significa che esiste una costante c tale che 2 numeri interi di N bit l'uno x e y letti in input in maniera consecutiva producono in output* $z = xy$ *dopo al massimo* cN *passi. Il numero z è espresso in forma binaria.*

Questo risultato sembra essere un'ulteriore dettaglio volto a confermare l'ipotesi della superiorità della potenza di calcolo delle SMM rispetto a

quella delle MdT per quanto riguarda alcune tipologie di algoritmi. Infatti, come descritto in [16], al momento il miglior upper bound conosciuto per la moltiplicazione di due numeri a N bit su una MdT multitape è $\mathcal{O}(NlogNloglogN)$. Questo ultimo risultato, viene leggermente migliorato utilizzando certi modelli di macchine, compresi tutti i tipi di RAM. La dimostrazione del seguente teorema è consultabile in [02].

Teorema 1.2.4. *Esiste una successor RAM capace di moltiplicare interi a N bit al costo logaritmico di $\mathcal{O}(NlogN)$.*

1.3 Term Graph Rewriting e Macchine di Schönhage

In questa sezione è possibile trovare un breve riassunto della precedente trattazione della presente autrice, dedicata al rapporto tra macchine di Schönhage e riscrittura su term graph. In particolare, riguarda la dimostrazione della possibilità di realizzare una SMM, opportunamente programmata, capace di applicare in maniera automatica una regola di riscrittura ad un term graph dato.

1.3.1 Riscrittura su Term Graph

Plump introduce la riscrittura su term graph nella sua pubblicazione [09] come segue:

> "La riscrittura su term graph riguarda la rappresentazione di espressioni funzionali per mezzo di grafi, e la valutazione di queste espressioni tramite trasformazioni del grafo basate su regole. Il voler rappresentare espressioni come grafi è motivato da considerazioni di efficienza."

Infatti, l'applicazione di regole su un term graph permette di ridurre il numero di occorrenze di un termine, attraverso l'uso di puntatori che conducano ad una di esse. In questo modo l'unica occorrenza risulta *condivisa* e viene valutata solo una volta risparmiando quindi non solo tempo ma anche spazio.

Definizione 1.4 (Ipergrafo)**.** Un ipergrafo è un grafo i cui archi posso connettere un numero arbitrario di vertici. Formalmente, è una coppia (V, E) dove V è un insieme di nodi ed E è un insieme di sottoinsiemi non vuoti di V.

Definizione 1.5 (Term Graph)**.** Un term graph G è un ipergrafo tale che:

- G è aciclico;

- esiste un nodo $root_G$ appartenente a V_G dal quale ogni altro nodo è raggiungibile. Questo nodo è detto *radice*;

- ogni nodo è il risultato di un unico arco.

Il nome term graph deriva dalla presenza in questi grafi dei cosiddetti *term*. Un term è una variabile, una costante o una stringa $f(t_1, \ldots, t_n)$ dove $f \in \Sigma$ e $(t_1, \ldots, t_n)$ sono a loro volta term. Il nodo v in un term graph G rappresenta il seguente valore:

$$term_G(v) = lab_G(e)(term_G(v_1), \ldots, term_G(v_n))$$

dove e è l'unico arco tale che il suo nodo risultato è v ed i suoi nodi argomenti sono $v_1, \ldots, v_n$. Se $arg(e)$ è vuoto allora $term_G(v) = lab_G(e)$, dove lab è la funzione che etichetta gli archi.

Il sistema di riduzione su term è un sistema di riduzione descritto dalla coppia $E = (\Sigma_E, R_E)$. In particolare gli elementi di Σ_E sono chiamati *funzioni*. Ogni elemento $t \in \Sigma_E$ ha una sua arietà. Dato un insieme di variabili non numerabile Υ è possibile definire come **closed term** i termini costituiti da simboli di funzioni; essi sono identificati con $T(E)$. Sono invece identificati con V(E,Υ) i termini costituiti da simboli di funzioni ed elementi di Υ. Essi vengono chiamati **term**.

R_E è invece un insieme di regole nella forma $t \rightarrow_E u$ dove t e u sono entrambi term. Tutti gli elementi di R_E sono diversi uno dall'altro e ogni variabile che appare in t è ripetuta una sola volta (dunque si tratta di regole lineari). Si consideri ora un term graph definito sull'insieme di funzioni Σ e l'insieme di variabili X. Sia $T_{\Sigma,X}$ l'insieme di tutti i term su Σ e t. Si dice *sostituzione* la mappatura $\sigma : T_{\Sigma,X} \rightarrow T_{\Sigma,X}$ tale che $\Sigma(c) = c$ per ogni costante c e $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$.
Una *regola di riscrittura su term* è una coppia di term $\langle l, r \rangle$ tale che:

- l non è una variabile;

- tutte le variabili di r sono contenute anche in l.

Si possono esprimere le singole regole nella forma $l \rightarrow r$. Una *relazione di riscrittura* $\rightarrow$ su $T_{\Sigma,X}$ indotta da R è definita come $t \rightarrow u$ se esistono una regola $l \rightarrow_E r$ in R e una sostituzione σ tali che $\sigma(l)$ è un sottotermine di t e u è ottenuto da t sostituendo ogni occorenza di $\sigma(l)$ con $\sigma(r)$. Sia un grafo morfismo un mapping che preserva la struttura dei grafi a cui è applicato.

Definizione 1.6 (Istanza)**.** Un term graph L è detto istanza del term l se esiste un grafo morfismo $f : \overline{l'} \rightarrow L$ tale che $f(root_{l'}) = root_L$.

Definizione 1.7 (Redex)**.** Dato un nodo v appartenente al term graph G e la regola di riscrittura di term $l \to r$, la coppia $\langle v, l \to r \rangle$ è detta redex se $G \downarrow v$ è istanza di l.

Infine, è possibile dare una definizione di riscrittura su term graph.

Definizione 1.8 (Riscrittura su term graph)**.** Sia G un term graph contenente il redex $\langle v, l \to r \rangle$. Allora esiste un *passaggio di riscrittura proprio* $G \Rightarrow_{v,l\to r} H$, dove H è il term graph costruito come segue:

- G_1 è il grafo ottenuto da G rimuovendo l'unico arco e tale che $res(e) = v$;

- G_2 è il grafo ottenuto dall'unione disgiunta di G_1 e $\overline{r'}$ in maniera tale che v sia la radice di $\overline{r'}$ e che vengano aggiunti anche gli archi che devono intercorrere tra $\overline{r'}$ e $\overline{l'}$;

- H è il term graph ottenuto da G_2 rimuovendo tutti i nodi e gli archi non più raggiungibili dalla radice.

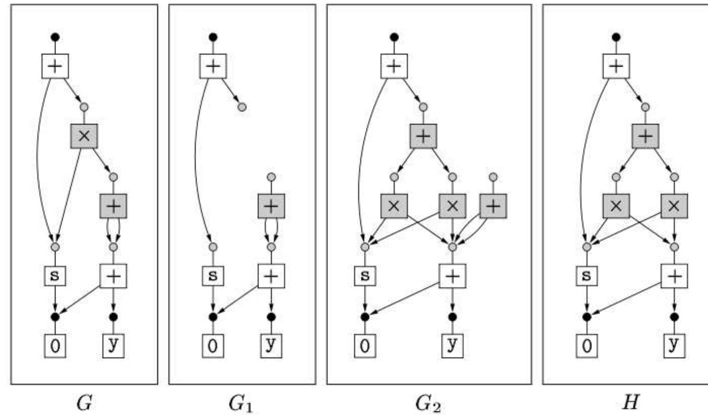

Fig. 1.6: L'applicazione della regola

La Figura 1.6 rappresenta l'applicazione al grafo G della regola $x \times (y+z) \to (x \times y) + (x \times z)$. Le sezioni di grafo grigie in G e H rappresentano i term graph rispettivamente di $x \times (y + z)$ e di $(x \times y) + (x \times z)$.

1.3.2 Strutture e Funzioni Ausiliarie per Macchine di Schönhage

Prima di discutere della simulazione è necessario introdurre il concetto di *strutture ausiliarie*, utili per astrarre i dettagli implementativi più complessi. L'insieme di istruzioni che realizzano il linguaggio delle macchine di

Schönhage è infatti sostanzialmente primitivo e limitato. In particolare, la mancanza di iterazione rende la realizzazione di certi procedimenti estremamente meccanica e ripetitiva. Si noti inoltre la mancanza di istruzioni riguardanti gli spostamenti all'interno del grafo. L'unica maniera per accedere ad un nodo è utilizzare il particolare percorso che porta ad esso e che altro non è che il valore del nodo stesso. Nei grafi di Schönhage, infatti, l'informazione è memorizzata non nei singoli nodi ma nella serie di etichette degli archi utilizzati per accedervi. Per un operatore umano con la possibilità di osservare la struttura del grafo questo valore è semplice da individuare e di immediato utilizzo. La macchina stessa, però, non mantiene una lista dei valori inseriti e riesce a trovare un nodo solo nel momento in cui un'istruzione richiede la sua manipolazione fornendone l'esatta posizione. Tutte le etichette introdotte per l'utilizzo delle strutture ausiliarie appartengono al sottoinsieme ausiliario $\Delta_a \in \Delta$. Supponiamo per comodità di lavorare con strutture aventi un numero massimo finito di nodi (e dunque una profondità massima). Si rimanda alla trattazione [03] per le dimostrazioni complete.

- **Input ed Output** Le istruzioni `input` e `output` delle SMM utilizzano sia in lettura che in scrittura singoli bit appartenenti all'insieme $\{0, 1\}$. Sarebbe utile poter lavorare su caratteri più complessi, come ad esempio nomi di funzioni. Si stabilisce quindi che ogni funzione che possa essere gestita da una particolare SMM debba essere espressa con un simbolo di grandezza e lunghezza finite, ad esempio il carattere di un byte. Come è noto, Δ è finito e pertanto l'insieme Δ_f delle possibili funzioni utilizzabili è noto prima della compilazione (ed è definito come $\Delta_f = \{f_0, f_1, \ldots, f_n\} \in \Delta$). È sufficiente realizzare un programma che abbia come effetto la lettura di un numero di valori corrispondente alla lunghezza della codifica binaria del nome della funzione. Questo programma è facilmente realizzabile per mezzo di sole istruzioni `input` e di un'etichetta λ_{err} corrispondente all'istruzione da eseguire in caso di lettura di un carattere non valido o inatteso;

- **Esistenza di un nodo** Si introduce ora il nodo ausiliario `exist`. Esso viene utilizzato per stabilire se un nodo identificato da un particolare percorso di etichette sia effettivamente esistente all'interno del grafo. Esso è puntato direttamente dal centro della struttura con un arco dedicato, etichettato con l'elemento `exist`. La stessa etichetta viene utilizzata ugualmente da tutti gli altri nodi della struttura fin dalla loro creazione.

- **Non esistenza di un nodo** Si ipotizzi di avere un grafo con un nodo associato alla stringa f (che per brevità chiameremo f) ed un nodo associato alla stringa fg. Tutti i puntatori uscenti nodo fg non inizializzati puntano a f. Si supponga di voler sapere se esiste fgf. Intuitivamente si tratta di un problema di facile risoluzione, poiché

si vede chiaramente che l'elemento non è presente nel grafo. Ciò nonostante, il percorso fgf conduce ad uno dei nodi effettivamente esistenti, la cui identità può variare a seconda delle istruzioni eseguite precedentemente. Per questa ragione si è deciso di introdurre il nodo `null`, a cui indirizzano tutti i puntatori non utilizzati fin dal momento della creazione di un nodo. Ció è possibile da implementare poichè tutti gli elementi di Δ sono noti fin dal principio;

- **Rappresentazione di funzioni** Si noti come le funzioni dell'insieme Δ_f possano avere arietà maggiore di uno. Per indicare gli argomenti delle relative funzioni viene aggiunto quindi un nuovo sottoinsieme Δ_{fa}. Si supponga ad esempio che esistano $f, g \in \Delta_f$ dove f è funzione binaria e g funzione unaria. Vengono create quindi le etichette $f1, f2, g1 \in \Delta_{af}$, che verranno utilizzate solo dai nodi di arrivo degli archi etichettati con i relativi nomi di funzione;

- `curr` e `curr'`: l'etichetta `curr` viene utilizzata solamente dal nodo centrale. Il relativo puntatore è diretto al nodo corrente ed è aggiornato ad ogni istruzione;

- `root`: è un puntatore al nodo radice del grafo vero e proprio. Serve a distinguerlo dagli elementi ausiliari;

- `level`: il puntatore etichettato con `level` viene utilizzato unicamente dal centro. Esso a sua volta sfrutta i puntatori $1, 2, \dots, p$, dove p è la profondità massima dell'albero. Inizialmente tutti gli archi di `level` conducono a `null`;

- `pending`: è un puntatore analogo a `level`. Anche esso utilizza puntatori etichettati con $1, 2, \dots, n$ dove n è il numero massimo di nodi;

- `parent`: l'arco di ogni nodo etichettato con `parent` punta al nodo genitore. Viene istanziato al momento della creazione del nodo stesso;

- `visited`: simile ad `exist`, è utilizzato per stabilire se un nodo è già stato visitato o meno.

Utilizzando le strutture ausiliare appena definite, non solo è possibile usare l'output di una macchina come input di un'altra macchina (purché il suo alfabeto sia compatibile), ma anche realizzare la visita di un grafo fornendo la sua struttura in output sotto forma di stringa composta da funzioni e caratteri ausiliari. È possibile inserire nella stringa anche informazioni riguardanti lo stato della struttura ausiliaria al momento della stampa.

Applicazione di una Regola di Term Rewriting ad una Δ-struttura

A questo punto si dimostra che è possibile scrivere un programma per SMM capace di riconoscere una data regola di term rewriting su un grafo in input e applicarla. Il term graph e la regola di rescrizione utilizzati nell'esempio (ed illustrati in Figura 1.7) sono tratti dalla pubblicazione di Dal Lago e Martini "Derivational Complexity is an Invariant Cost Model" [09]. Il programma è strutturato in tre parti: la fase di individuazione, la fase di costruzione e la fase di redirezione.

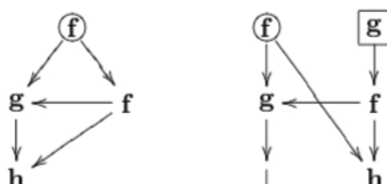

Fig. 1.7: Term graph e regola di rescrizione

Nella fase di individuazione si analizza il grafo mano a mano che esso viene inserito nella Δ-struttura, cercando di riconoscere gli elementi appartenenti alla parte sinistra della regola da applicare. In ogni momento dell'esecuzone il programma si trova in uno dei suoi possibili stati.

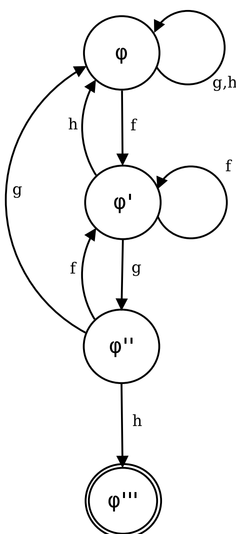

Fig. 1.8: Diagramma degli stati

Ogni volta che uno di questi elementi viene riconosciuto, lo si punta con un arco ausiliario σ_i, diretto a partire dal centro, e si aggiorna lo stato. Per quanto riguarda l'esempio trattato, la Figura 1.8 rappresenta l'albero degli stati del sistema, e quali particolari input causano cambiamenti di stato. La struttura al termine di questa fase è rappresentata dalla Figura 1.9

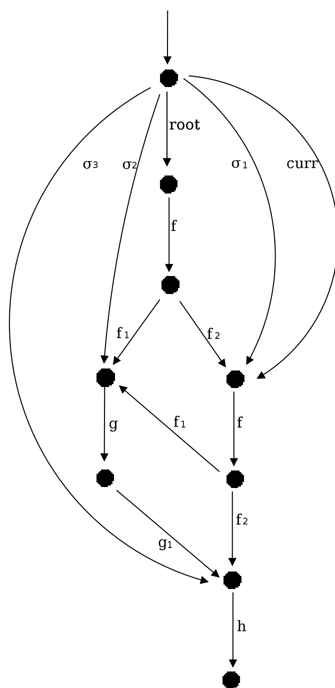

Fig. 1.9: Fase di individuazione

Segue poi la fase di costruzione, nella quale si inseriscono i nodi da aggiungere al momento dell'applicazione della regola. Essi non vengono inseriti nella parte di struttura identificata con `root`, ma da un ulteriore sottografo con origine dal centro puntato dall'arco `rx`. L'aspetto della struttura al termine di questa fase è visibile nella Figura 1.10.

A questo punto l'unica operazione rimanente, in quella chiamata fase di redirezione, consiste nel'inizializzare i puntatori dalla struttura `rx` al grafo originale. Poichè i nodi che vengono utilizzati in questa fase sono già stati individuati si tratta di un processo molto rapido. Serve inoltre togliere il puntatore alla vecchia testa del grafo in modo da non potervi più accedere. L'aspetto finale della struttura è rappresentato dalla Figura 1.11

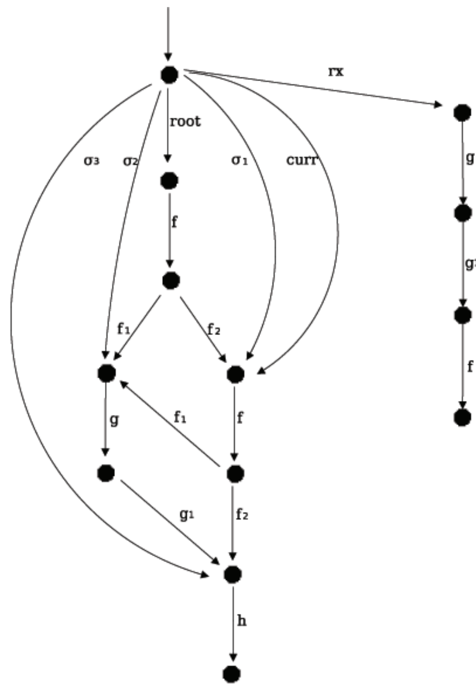

Fig. 1.10: Fase di costruzione

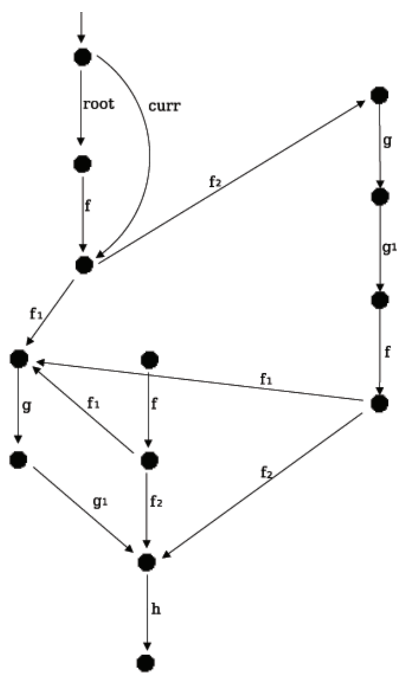

Fig. 1.11: Fase di redirezione

Capitolo 2

Gurevich Abstract State Machine

Il modello descritto da Leivant e Marion nel loro studio sulla complessità computazionale implicita [11] prende ispirazione da due modelli: le macchine di Schönhage, già introdotte nel precedente capitolo, e le Abstract State Machine (in seguito ASM) ideate da Yuri Gurevich. Nei paragrafi che seguiranno verranno quindi presentate le caratteristiche principali di queste macchine, basandosi in particolare sulle definizioni fornite in [12] e [13]. Un interessante risultato meritevole di approfondimento è quello descritto in [08]: i modelli di Schönhage e Gurevich sono lock-step equivalenti.

Le macchine di Turing forniscono una semantica operazionale per gli algoritmi. Tuttavia, per vari motivi, si può considerare questa semantica non ottimale: un semplice passo di un dato algoritmo può richiedere una lunga sequenza di istruzioni alla MdT che lo simula. Gurevich propone quindi un modello più versatile: nate in un primo momento con il nome di evolving algebra (o ealgebra), le sue macchine a stati astratti possono adattarsi ai diversi livelli di astrazione richiesti da un algoritmo, fornendone una semantica operazionale più semplice ed intuitiva. Il termine *algebra* viene adottato dalla disciplina dell'algebra universale, in cui indica le strutture del prim'ordine senza relazioni.

Pur non fornendone una dimostrazione, Gurevich si basa su numerosi risultati ottenuti per dichiarare che è probabile che ogni algoritmo possa essere simulato in lock-step (di cui verrà fornita una definizioni nei paragrafi successivi) da una opportuna macchina a stati astratti, cioè esiste un'algebra con simile livello di astrazione che usa un quantitativo di risorse analogo a quello dell'algoritmo di partenza.

Le ealgebra sono state applicate anche in contesti differenti da quello di

pura simulazione. L'International Standard Organization ISO ha adottato una semantica basata sulle evolving algebra per descrivere Prolog [14]. In un'altra occasione, esse sono state utilizzate per specificare le caratteristiche semantiche di modelli per database object-oriented [15]. Lo stesso Gurevich introduce in [08] versioni delle ealgebra per computazioni parallele, non deterministiche e distribuite.

2.1 Algebre Statiche e Update

Come precedentemente accennato, un'algebra è una struttura senza relazioni. Un'ulteriore restrizione concentra l'attenzione sulle algebre con due operazioni nullarie distinte *true* e *false*, le operazioni di base su booleani e la relazione di equivalenza. Delle strutture così ottenute verranno prese in considerazione quelle multi-sorte con operazioni parziali. Le sorte possono essere rappresentate per mezzo di relazioni unarie, chiamate **universi**.
Una **segnatura** è un insieme finito di nomi di funzioni, ognuno di essi con arietà fissa. Ogni segnatura contiene il segno uguale, i nomi di funzioni nullarie *true, false* e *undef* (dette costanti logiche) e il nome delle operazioni booleane base.

2.1.1 Definizione di Algebre Statiche

Un'**algebra statica** o **stato** S di una segnatura Υ, o *Fun*(S), è un insieme non vuoto X chiamato **superuniverso** di S insieme alle interpretazioni su X dei nomi di funzioni contenuti in Υ. Mentre il superuniverso è un insieme fisso, l'interpretazione delle funzioni può cambiare nel corso dell'esecuzione (solo se si tratta di funzioni dinamiche, le funzioni statiche vengono interpretate sempre nella stessa maniera).
Una funzione n-aria è interpretata come funzione da X^n a X ed è chiamata **funzione base** di S. Allo stesso modo una **relazione base** è interpretata come una funzione da X^n a $\{true,false\}$. Le interpretazioni di *true*, *false*, *undef* sono elementi distinti di X. Intuitivamente, l'elemento *undef* permette di rappresentare funzioni parziali: se una funzione f non è definita su $\overline{a}$ allora $f(\overline{a}) = undef$ (ad esempio quando un operatore booleano riceve come argomento un non booleano). Il dominio di una funzione n-aria f viene visto come l'insieme di n-tuple $\overline{x}$ per le quali $f(\overline{x})$ è diverso da *undef*, anche se in realtà le funzioni base sono totali grazie all'utilizzo di *undef*.

Come già accennato, si vogliono considerare algebre con multi-sorte. È possibile vedere una relazione base f come un insieme di tuple che valutano a *true*. Se f è unaria allora essa viene chiamata **universo**. Ad esempio, si può considerare l'universo *Nodes* e una relazione binaria *Edge* su di esso, tale che $Edge(x, y)$ vale solo se sia x che y appartengono a *Nodes*. Da questo momento in poi si supporrà che ogni algebra statica abbia la relazione di

identità, un universo *Bool* comprendente *true* e *false* e le solite operazioni booleane. Il loro risultato è *undef* se almeno uno degli argomenti non appartiene a *Bool*. Si noti che *undef* non appartiene a nessun universo.

2.1.2 Termini

Così come nella logica del primo ordine, i termini sono definiti ricorsivamente:

- Una variabile è un termine;
- Se f è il nome di una funzione $n-$aria e $t_1, \ldots, t_n$ sono termini allora $f(t_1, \ldots, t_n)$ è un termine.

I **termini ground** sono termini senza variabili. I **termini booleani atomici** sono nella forma $f(\bar{t})$ con f nome di relazione. Di conseguenza, i **termini booleani** sono costruiti per mezzo delle operazioni booleane partendo dai termini booleani atomici. Uno stato S è **appropriato** per un dato termine s se *Fun*(S) include l'insieme dei nomi di funzioni che occorrono in s (indicato con *Fun*(S)). In uno stato appropriato S un ground term $t = f(t_1, \ldots, t_n)$ valuta ad un elemento $Val_S(t) = f(Val_S(t_1), \ldots, Val_S(t_n)$.

Si consideri ora un esempio di algebra statica, la cui segnatura contenga i nomi di funzioni nullarie 0 e 1 e i nomi di funzioni binarie $+$, $\times$ e $\div$. Il suo superuniverso è l'insieme degli interi e i nomi di funzioni possono essere interpretati nell'ovvia maniera. Questa algebra ha inoltre l'universo *Integer* che comprende i numeri interi tra cui 0 e 1 e $a \div b =$*undef* a meno che a e b non siano interi tali che a divide b. Ecco alcuni esempi di termini per questa segnatura: $0, 1+1, (1+1) \times (1+1)$.

2.1.3 Locazioni e Update

Definizione 2.1 (Ridotto)**.** Data una segnatura Υ , il ridotto di un suo stato S slla segnatura Υ' è lo stato S' ottenuto da S rimuovendo le interpretazioni dei nomi di funzioni contenuti in $\Upsilon - \Upsilon'$. S viene detto *espansione* di S'.

Definizione 2.2 (Carrier e locazioni)**.** Un carrier è uno stato la cui segnatura contiene solo nomi di funzioni statiche, ed è possibile ridurre uno stato S al suo carrier $\mid S \mid$ lasciando solo le parti statiche di *Fun*(S). Una *locazione* su un carrier C è una coppia $l = (f, \overline{x})$ dove f è un nome di funzione non contenuto in $Fun(C)$ e $\overline{x}$ è una tupla di elementi di C di lunghezza opportuna. La locazione è detta *relazionale* se f è il simbolo di una relazione.
Loc$_\Upsilon(C)$ è l'insieme di locazioni su C con nomi di funzioni in Υ e gli stati S su Υ con carrier C vengono talvolta visti come funzioni da *Loc*$_\Upsilon(C)$ a C. Se uno stato S è appropriato per un termine ground $t_0 = f(\bar{t})$ allora la locazione di t_0 in S è la locazione $(f, Val_Z(\bar{t}))$.

Definizione 2.3 (Update)**.** Un update di uno stato S è una coppia $\alpha = (l, y)$ dove l è una locazione su S e $y \in\mid S \mid$. Se l è relazionale allora y è booleano. La locazione l è la locazione $Loc(\alpha)$ di α e y è il valore $Val(\alpha)$ di α. Per eseguire α su S si ridefinisce S affinché mappi l a y. Il risultato è un nuovo stato S' tale che $Fun(S') = Fun(S)$, $\mid S' \mid=\mid S \mid$, $S'(l) = y$ e $S'(l') = S(l')$ per ogni locazione l' su S differente da l.

2.1.4 Insiemi di Update e Famiglie di Insiemi di Update

Sia β un **insieme di update** su uno stato S. $Loc(\beta) = \{Loc(\alpha : \alpha \in \beta\}$. Per ogni $l \in Loc(\beta), Val_\beta(l) = \{Val(\alpha : \alpha \in \beta \wedge Loc(\alpha) = l\}$. Un insieme di update β è **consistente** per lo stato S se ogni $Val_\beta(l)$ è un insieme composto da un solo elemento. Eseguendo un insieme di update consistenti, essi vengono eseguiti simultaneamente. Il risultato è un nuovo stato S' con stessi segnatura e carrier di S. Se $l \in Loc(\beta)$ allora $S'(l)$ è l'unico elemento di $Val_\beta(l)$, altrimenti $S'(l) = S(l)$. Se l'insieme di update eseguiti non è consistente il risultato sarà uno stato equivalente a quello di partenza.
Per eseguire una **famiglia** γ di insiemi di update su S, si sceglie un insieme di update $\beta \in \gamma$ e lo si esegue su S. Se la famiglia è un insieme vuoto essa indica inconsistenza e non ha efetti su S.

2.2 Sequential Evolving Algebra

In questa sezione verranno presentate le regole di transizione base, soffermandosi sul problema dell'estensione degli universi, presentando poi programmi ed esecuzioni. Tutti i termini verranno sottointesi come ground.

2.2.1 Basic Transition Rules

Una istruzione **update** R è una espressione nella forma $f(\bar{t}) := t_0$, dove f è il nome di una funzione non statica, detto *soggetto* dell'istruzione, $\bar{t}$ è una tupla di termini di lunghezza opportuna e t_0 un altro termine (che deve essere booleano se f è il nome di una relazione). La regola indica che si valutano i termini t_i e se a_i è il valore di t_i si imposta $f(a_1, \ldots, a_n)$ ad a_0, cioè si modifica il valore di $f(\bar{t})$ in t_0 nello stato successivo. Dunque, per eseguire R su un appropriato stato S, si lancia l'update $\alpha = (l, y)$ su S, dove $l = f, Val_S(t)$ e $y = Val_S(t_0)$. Da questo momento si indicherà $\{\alpha\}$ con Updates(R, S).

Fondamento delle regole base sono le **istruzioni update**, create ricorsivamente per mezzo di due costruttori: blocco e condizionale. Per ogni regola R ed ogni stato S ad essa appropriato viene definito un insieme di update Updates(R, S) su S, lanciato per eseguire R su S.

Il **costruttore blocco** indica che una sequenza R di regole $R_1, \ldots, R_k$ è una regola, cioè: Updates(R, S)=Updates$(R_1, S) \cup \ldots \cup$ Updates(R_k, S)
In altre parole, per eseguire un insieme di regole bisogna lanciarle simultaneamente. Updates(R, S) è inconsistente se lo è qualsiasi R_i, o se una coppia di queste regole modifica il valore di uno stesso elemento. In tal caso, nessuna delle regole viene eseguita.

Il **costruttore condizionale** si presenta nella forma:

if g_0 then R_0
elseif g_1 then R_1
$\vdots$
elseif g_k then R_k
endif

Dove k è un numero naturale, $g_0, \ldots, g_k$ sono termini booleani e $R_0, \ldots, R_k$ sono regole. Gli elementi $g_0, g_1, \ldots$ detti *sentinelle* vengono valutati sequenzialmente fino al primo g_i valutato true. Dopodiché viene eseguita la transizione identificata con R_i, cioè Updates(R, S)=Updates(R_i, S). Se ogni sentinella viene valutata come false allora l'update è inconsistente.

Una istruzione multi-update è una sequenza di istruzioni update. Una **istruzione update con sentinelle** (e il suo corrispettivo multi-update) è una regola nella forma:

if g then R endif

dove R è un'istruzione update (o multi-update, rispettivamente).

Lemma 2.1 (Update)
Per ogni regola R esiste una sequenza R' di update con sentinelle tale che $Fun(R') = Fun(R)$ e Updates(R', S)=Updates(R, S) per tutti gli stati appropriati S.

Versioni più recenti di evolving algebra permettono l'utilizzo di due ulteriori costruttori:

- Il costruttore **case**, che permette un'esecuzione molto più efficiente, permettendo di generare più velocemente il relativo insieme di update. Esso rende inoltre più semplice la programmazione di un'esecuzione sequenziale di regole;

- Il costruttore let $x = t$ in R, che evita successive valutazioni del termine t in R.

2.2.2 Importare Nuovi Elementi

Un algoritmo sequenziale potrebbe aver bisogno di aggiungere un nuovo elemento. Serve dunque una regola che lo permetta. Invece che creare nuovi elementi, viene introdotto un universo speciale chiamato *Reserve*, da cui poter prendere nuovi elementi. Questo universo non è statico e non è necessario che sia presente nella segnatura di ogni algebra statica. Se però lo contiene, allora l'insieme $\{x{:}S \mid x \in Reserve\}$ è chiamato *riserva di S*.
Ogni stato deve soddisfare le seguenti condizioni:

- Ogni relazione base, ad eccezione di Reserve e dell'eguaglianza, restituisce $false$ se almeno uno degli argomenti appartiene alla riserva;

- Ogni altra funzione base restituisce *undef* se almeno uno degli argomenti appartiene alla riserva;

- Nessuna funzione base restituisce elementi della riserva

A questo punto è necessario aggiornare le definizioni di term e update. Le **variabili** sono nomi di funzioni nullarie ausiliarie che non possono essere soggetto di un'istruzione update. Le regole sono costruite dalle istruzioni update per mezzo di tre costruttori: sequenza, condizionale e import (in aggiunta ai facoltativi case e let).

Il **costrutto import** viene espresso per mezzo della seguente formula, contenente la variabile v detta *main existential variable* e dalla regola R_0 detta *corpo*:

import v R_0 endimport

Chiamiamo una regola *perspicua* se nessuna variabile ha sia occorrenze libere che legate e nessuna variabile legata è dichiarata più di una volta. Solitamente v è un elemento appartenente all'universo *Reserve*.

Sia Free(R) l'insieme di variabili libere della regola R. Analogamente si definisca Bound(R). Se R è una regola import con main existential variable v e corpo R_0 allora:

$$\text{Free}(R) = \text{Free}(R_0) - \{v\}, \wedge \ \text{Bound}(R) = \text{Bound}(R_0) \cup \{v\}$$

Per comodità è possibile considerare le variabili libere delle regole come funzioni nullarie. Viene quindi introdotto il concetto di **segnatura ausiliaria** Υ', composto dalla segnatura originale Υ e dall'insieme finito di variabili V. Si ricordi che se S è uno stato con segnatura ausiliaria Υ' allora Fun(S) $= \Upsilon'$. Inoltre S è *appropriato* per una regola R se la sua segnatura ausiliaria contiene tutti i nomi delle funzioni e le variabili libere di R. R è S*-perspicua*

se è perspicua e le sue variabili legate non occorrono in V.

Importante è sottolineare che importando un elemento della riserva esso viene rimosso da essa. Una successiva importazione della stessa variabile creerà due elementi distinti. Analogamente a quanto visto in precedenza, per ogni regola R e stato per essa appropriato S è possibile definire Updates(R, S). Si rimanda a [13] per i dettagli. Infine, esistono abbreviazioni che permettono un uso più pratico di import. Ad esempio:

import $v_1, \ldots, v_k$
R_0
endimport

viene utilizzato per indicare l'importazione consecutiva di k variabili. Allo stesso modo

import $v_1, \ldots, v_k$
$U(v_1) := true$
$\vdots \quad U(v_k) := true$
R_0
endimport

indica l'aggiunta di k elementi all'universo U.

2.2.3 Programmi e Run

Un **programma** P è una regola senza variabili, che diventa base se anche la regola in questione lo è. Solitamente un programma è composto da una sequenza di regole. Per lanciare un programma P su uno stato S è sufficiente lanciare Update(P, S). Un **pure run** di P è una sequenza di stati S_n (con n numero naturale) la cui segnatura è Fun(P) tale che per ogni i lo stato S_{i+1} è ottenuto dallo stato S_i lanciando P su di esso.

A questo punto si vuole considerare l'ipotesi in cui sia necessario interagire con l'ambiente esterno, definendo le **funzioni esterne** (in opposizione alle funzioni interne viste fino a questo punto). Esse non possono essere cambiate dalle regole dell'algebra, ma possono comunque restituire valori diversi a seconda dello stato in cui vengono invocate. Dal punto di vista dell'algebra, una funzione esterna è come un oracolo dinamico che può venire utilizzato per ricevere output su dati argomenti ma non può venire controllato. Per garantire consistenza, si richiede che non sia possibile annidare funzioni esterne. Allo stato d'arte, non sono note applicazioni in cui sia impossibile fare a meno di questo costrutto e si può dunque considerare questa restri-

zione non limitante, almeno per il momento.

Il **run** di un programma P è quindi una sequenza di stati S_n tale che:

- Ogni stato S_i intermedio è appropriato per P. Lo stato finale, se esiste, è uno stato della segnatura interna;
- Ogni stato S_{i+1} è ottenuto lanciando P su S_n.

Esistono anche casi in cui una funzione è gestita in parte esternamente ed in parte internamente. In questi casi si preferisce parlare di locazioni interne ed esterne.

2.3 Lock-step Equivalenza tra SMM e ASM

Una macchina a stati astratti $\mathcal{A}$ è definita in [08] come una tupla $\{V, p, A_0\}$ dove V è la segnatura, p è un programma e A_0 è lo stato iniziale.

Ogni macchina a stati astratti contiene l'universo Modes=$\{$Initial, Working, Final$\}$ nella sua segnatura. L'elemento Mode identifica la modalità nella quale il programma si trova in ogni istante. L'input è una stringa binaria rappresentata dall'universo InputPositions, dagli elementi 0 e Last e dalle funzioni unarie Succ e Bit. Utilizzando questi elementi è possibile ordinare l'universo: Succ(0) diventa 1, Succ(1) diventa 2 e così via fino ad arrivare a Succ(Last)=undef. La funzione Bit associa InputPositions all'insieme $\{0, 1\}$. Pertanto, poiché la stringa in input è rappresentata dall'elemento InputString, allora Bit(InputString) rappresenta il bit corrente della stringa in input. L'output viene rappresentato con l'omonima funzione nullaria Output. Dato $\langle A_i : i \in \Lambda \rangle$ allora Output assume il valore (β) nello stato A_i. Nello stato A_0 Output ha valore undef.

L'esecuzione di un programma è identificata con una successione $A_0, \ldots, A_i$ di stati. Ogni stato A_{i+1} è calcolato a partire dallo stato A_i applicando gli update determinati dal programma.

2.3.1 Simulazione Lock-step e Lock-step Equivalenza

Per comprendere meglio la definizione di lock-step equivalenza servono alcune chiarificazioni. L'espressione $\langle A_i : i \in \Lambda \rangle$ indica una esecuzione della macchina $\mathcal{A}$. Λ è una sequenza non vuota di valori appartenenti a $\mathbb{N}$. A_0 è lo stato iniziale e A_j è lo stato finale (o hang state). Se Λ è finito è $j = \max(\Lambda)$ per nessun $k < j$ A_k è finale. Il concetto di lock-step venne definito da Gurevich stesso nella seguente maniera:

Definizione 2.4 (Simulazione lock-step)**.** Una macchina $\mathcal{B}$ simula una macchina $\mathcal{A}$ in lock-step con fattore di lag c se esiste una mappatura ϕ tra lo

stato A e lo stato B tale che per ogni run $\langle A_i : i \in \Lambda \rangle$ di $\mathcal{A}$ esiste un run $B_0, B_1, \ldots$ di $\mathcal{B}$ e una funzione monotona $J : \Lambda \to \mathbb{N}$ tale che:

- $J(0) = 0$; inoltre A_0 e B_0 hanno lo stesso input;

- $B_{J(i)} = \phi(A_i)$ e se x è l'input di A_i allora x è esattamente l'input di $B_{J(i)}$;

- Se un output β è generato nel passaggio da A_i a A_{i+1} allora esiste ed è unico $l \in [J(i), J(i-1)]$ tale che β sia l'output generato nel passaggio da B_l a B_{l+1}. Se β è nullo nel passaggio da A_i a A_{i+1} allora lo è anche da B_l a B_{l+1};

- Se $0 < \max(\Lambda)$ allora $J(i) - J(i-1) \leq c$. In altre parole, il numero di passi necessari per passare dallo stato i allo stato j non supera mai una certa costante c;

- Se Λ è finito, $i = \max(\Lambda)$ e A_i è lo stato finale allora anche $B_{J(i)}$ lo è.

Si noti che dando questa definizione Gurevich si riferiva a una particolare categoria di computer device, le cui caratteristiche riproponiamo brevemente in seguito: si tratta di device deterministici, il cui input viene fornito prima dell'esecuzione. L'output può essere generato in qualsiasi momento e non vi è altro tipo di interazione con l'ambiente esterno. Sono macchine che lavorano con sequenze ordinate di stati e hanno quindi uno stato iniziale e uno stato finale. Ogni stato è caratterizzato da un input (una stringa binaria) e da un output (un singolo bit) che può essere nullo. A questo punto è possibile dare le seguente definizioni:

Definizione 2.5. $\mathcal{A}$ simula in lock-step $\mathcal{B}$ strettamente se $c = 1$.

Definizione 2.6 (Lock-step equivalenza)**.** Due modelli di macchine $\mathbb{A}$ e $\mathbb{B}$ sono lock-step equivalenti se per ogni macchina $\mathcal{A}$ descritta da $\mathbb{A}$ esiste una macchina $\mathcal{B}$ descritta da $\mathbb{B}$ che simuli $\mathcal{A}$ in lock-step con fattore di lag finito e per ogni macchina $\mathcal{B}$ descritta da $\mathbb{B}$ esiste una macchina $\mathcal{A}$ descritta da $\mathbb{A}$ che simuli $\mathcal{B}$ in lock-step con fattore di lag finito.

2.3.2 Simulazione di una SMM Utilizzando una ASM Unaria

La dimostrazione di lock-step equivalenza tra SMM e ASM si divide in due parti: nella prima si vuole dimostrare che per ogni macchina a modificazione di memoria $\mathbb{A}$ esiste una macchina a stati astratti $\mathbb{B}$ capace di simularla in lock-step. Nella seconda parte ci si occuperà della dimostrazione speculare.

Simulazione della Struttura Dati di una SMM

Come è già stato detto, una macchina di Schönhage è descritta nel suo stato iniziale da una stringa in input, una stringa vuota in output, da un programma e da una Δ-struttura contenente un particolare nodo chiamato centro. Serve quindi trovare un corrispettivo per ognuno di questi elementi. Il caso più elementare è quello dell'input/output, che può essere simulato con gli analoghi meccanismi delle ASM. Per quanto riguarda il programma, esso può essere visto come una lista di istruzioni. Per poterle enumerare, vengono utilizzati elementi dell'universo InputPositions, uno dei quali prende il nome di CurInst e indica l'istruzione corrente. Anche per simulare la Δ-struttura viene utilizzato un apposito universo Nodes che contiene l'elemento Center. Il valore iniziale di Center è undef; esso viene utilizzato come corrispondente del nodo centrale. Inizialmente l'universo è vuoto, ma verrà riempito mano a mano che il programma viene eseguito. Gli archi vengono rappresentati utilizzando delle funzioni unarie f: Nodes $\rightarrow$ Nodes. Per ogni $\delta \in \Delta$ esiste una funzione omonima. In aggiunta a questi costrutti viene utilizzato l'elemento Mode per specificare la fase di simulazione: nella fase iniziale Mode assume valore 0 o Initial, nella fase di esecuzione ha valore 1 o Working, al termine dell'esecuzione 2 o Halt. Nello stato iniziale, la ASM simulante ha le seguenti caratteristiche:

- Mode è settato a Inital, CurInst a 0;
- Output ha valore undef;
- Bit(n) $\in \{0, 1\}$ per ogni $n \in$ InputPositions;
- InputString ha valore 0.

Simulazione delle Istruzioni per SMM

Serve adesso capire come tradurre un programma scritto per una SMM in un programma utilizzabile da una ASM. Lo stato iniziale precedentemente definito può essere descritto come segue:

```
if CurInst = 0 and Mode = Initial then
        import y
                Nodes(y) := true
                Center := y
                δ_1(y) := y
                ⋮
                δ_m(y) := y
        endimport
        CurInst := 1; Mode := Working
endif
```

Il centro del nodo viene quindi importato e ogni puntatore viene settato in modo da puntare al nodo stesso. Dopodiché si incrementa il puntatore all'istruzione corrente e la modalità di simulazione passa su Working. Per quanto riguarda le singole istruzioni, Gurevich descrive la modalità di simulazione di ognuna di esse. Sono riportate in seguito le più interessanti, insieme a una loro breve spiegazione. Per cominciare, il seguente blocco di istruzioni simula `input` λ_0 λ_1:

```
if CurInst = i and Mode = Working then
        if Bit(InputString) = undef then
            CurInst := Succ(CurInst)
        elseif Bit(InputString) = 0 then
            CurInst := Lambda0
        else CurInst := Lambda1
endif
```

Se il bit letto dalla stringa in input non è stato definito, si passa all'istruzione successiva. In caso contrario, viene assegnato come valore all'elemento CurInst quello corrispettivo all'istruzione da eseguire. Dopodiché si aggiorna anche il puntatore alla stringa in input facendolo scorrere di una posizione.

```
if CurInst = i and Mode = Working then
        Output := Beta
        CurInst := Succ(CurInst)
endif
```

Questo blocco di istruzioni simula `output` β. Viene modificato il valore di Output e incrementato il puntatore all'istruzione corrente. Per quanto riguarda l'output, è necessario integrare l'istruzione appena vista con il seguente controllo:

```
if CurInst ≠ i_1 and ... and CurInst ≠ i_k then
        if Output≠ undef then
                Output := undef
        endif
endif
```

Questa parte di codice viene infatti aggiunta al programma per garantire che Output abbia valore non definito per ogni istruzione che non generi output. Il seguente blocco di codice indica il termine dell'esecuzione corrispondente allo stato di `halt`.

if CurInst $= i$ and Mode $=$ Working then
Mode := Final;
endif

Si passa ora ad un tipo di istruzione più complesso. I due blocchi qui illustrati corrispondono alle istruzioni `new` $\Box$ e `new` $w_1 \ldots w_k$ $(k \geq 1)$.

if CurInst $= i$ and Mode $=$ Working
then
import y
Nodes(y) := True
Center := y
$\delta_1(y)$:= Center
$\vdots$
$\delta_m(y)$:= Center
endimport
CurInst := Succ(CurInst)
endif

if CurInst $= i$ and Mode $=$ Working
then
import y
Nodes(y) := True
β(PARENT) := y
$\delta_1(y)$:= δ_1(PARENT)
$\vdots$
$\delta_m(y)$:= δ_m(PARENT)
endimport
CurInst := Succ(CurInst)
endif

Il caso base della creazione di un nuovo nodo prevede il recupero dell'elemento y nell'universo Reserve e il suo inserimento nell'universo Nodes. Come visto nella descrizione delle SMM, se il nuovo nodo da creare è vuoto esso viene settato come centro e tutti i puntatori da esso uscenti indicano il centro stesso. In caso contrario, i puntatori del nuovo nodo porteranno al nodo genitore (indicato con PARENT), che a sua volta utilizzerà un puntatore per riferirsi al figlio.

if CurInst $= i$ and Mode $=$ Working
then
Center := Center.$\mathsf{v}_1.\mathsf{v}_2 \ldots \mathsf{v}_j$
CurInst := Succ(CurInst)
endif

if CurInst $= i$ and Mode $=$ Working
then
w_k(Center.$\mathsf{w}_1.\mathsf{w}_2 \ldots \mathsf{w}_{k-1}$) :=
Center.$\mathsf{v}_1.\mathsf{v}_2 \ldots \mathsf{v}_j$
CurInst := Succ(CurInst)
endif

Il valore della funzione w_k applicata all'elemento $w_{k-1}(w_{k-2}(\ldots(w_1(\mathsf{Center}))\ldots)$ diventa uguale a quello dell'elemento indicato da $v_j(v_{j-1}(v_{j-2}(\ldots(v_1(\mathsf{Center}))\ldots)$. Nel caso base il centro diventa l'elemento indicato da $v_j(v_{j-1}(v_{j-2}(\ldots(v_1(\mathsf{Center}))\ldots)$.

if CurInst $= i$ and Mode $=$ Working then
if Center.$\mathsf{u}_1\mathsf{u}_2 \ldots \mathsf{u}_k$ $=$ Center.$\mathsf{v}_1\mathsf{v}_2 \ldots \mathsf{v}_j$ then
R_σ
else
CurInst := Succ(CurInst)
endif
endif

R_σ è l'update per ASM corrispondente all'istruzione σ per SMM, senza il controllo di CurInst e Mode.

La simulazione lock-step prevede l'individuazione di una funzione di mappatura ϕ con le seguenti caratteristiche:

- se la stringa $w_1, \ldots, w_l$ descrive l'elemento x allora in $\phi(A)$ x è identificato da Center.$w_1.\ldots.w_k$;

- se k è l'indice dell'istruzione corrente in A allora CurInst in $\phi(A)$ ha valore k;

- se A presenta input x, $\phi(A)$ presenta lo stesso input xL;

- se lo stato A produce l'output β, lo stato $\phi(A)$ produce lo stesso output;

- se A è lo stato iniziale, Mode è uguale a Initial. Se A è lo stato finale, in ϕ Mode è uguale a Final. Se non si tratta dello stato iniziale né dello stato finale, Mode = Working.

Inoltre, quando $\mathcal{A}$ crea un nuovo elemento a, $\mathcal{B}$ importa l'elemento a' per rappresentare a.

Volendo provare che per ogni SMM $\mathcal{A}$ con stati $A_0, A_1, \ldots$ esiste una ASM $\mathcal{B}$ con stati $B_0, B_1, \ldots$ tale che per ogni $B_i = \phi(A_i)$, Gurevich utilizza una dimostrazione per induzione. Il caso base $B_0 = \phi(A_0)$ è già stato trattato nei precedenti paragrafi. Serve quindi dimostrare che se $B_{k-1} = \phi(A_{k-1})$ allora $B_k = \phi(A_{k-1})$. Dalla descrizione della simulazione di un programma per SMM risulta palese come ogni tipologia di istruzione prevista da Schönhage sia simulabile da un update per una ASM in ogni suo aspetto. Ciò dimostra non solo che il passaggio da $\phi(A_{k-1})$ a $\phi(A_k)$ è possibile tramite $\mathcal{B}$ ma, poiché gli update sono atomici, anche che $\phi(A_i) = B_i$.

2.3.3 Simulazione di una ASM Tramite SMM

A questo punto è necessario occuparsi del caso speculare a quello appena trattato. In questa sezione viene riportata in breve la procedura utilizzata da Gurevich per dimostrare che per ogni ASM $\mathcal{A}$ esiste una SMM $\mathcal{B}$ in grado di simularla strettamente lock-step. Analogamente a quanto precedentemente fatto, la simulazione verrà compiuta cercando di utilizzare i costrutti disponibili (in questo caso i puntatori della macchina di Schönhage) per riprodurre le caratteristiche della macchina da simulare (cioè i valori delle funzioni della macchina a stati astratti), nonché le istruzioni.

Simulazione della Struttura Dati di una ASM

Per prima cosa, si stabiliscono i valori appartenenti a Δ. È possibile identificare nel modello di ASM che si vuole simulare tre differenti tipologie di funzioni:

- **Funzioni statiche nullarie**: servono per dare un nome agli elementi. Per esprimerle nella macchina simulante si utilizzano puntatori uscenti direttamente dal nodo centrale. Si noti che in questa maniera è necessario che il centro sia fisso durante l'esecuzione. Anche le **funzioni dinamiche nullarie** vengono rappresentate con la stessa modalità.

- **Funzioni unarie**: sono rappresentate come puntatori tra nodi semplici. Il nodo dal quale il puntatore esce rappresenta il valore di partenza, il nodo puntato rappresenta il risultato dell'applicazione della funzione al valore di partenza.

Per ogni funzione f viene utilizzato anche un puntatore f' utilizzato come flag per eventuali controlli. Dal nodo attivo escono anche puntatori con le etichette `True, False` e $New_1, \ldots, New_k$ per simulare l'aggiunta di k elementi tramite Import. Il nodo centrale corrisponde a undef. Nella stessa maniera con cui è stato descritto il caso base di una SMM si cerca a questo punto di simulare lo stato iniziale di una ASM. Esso consiste in:

- un superuniverso ed una interpretazione dei nomi di funzioni presenti nella segnatura. Entrambi sono rappresentati in modo tale che per ogni $f_k(f_0.f_1 \ldots f_{k-1}) = x$ allora $f_0 f_1 \ldots f_k$ e $f_0 f_1 \ldots f_k'$ indichino x nella Δ-struttura;

- l'universo di InputPortions. Esso è rappresentato dalla stringa degli input;

- Output = undef, rappresentato dalla stringa vuota in output;

Simulazione delle Istruzioni per ASM

Le istruzioni delle macchine a stati astratti vengono simulate nelle modalità descritte di seguito.

Updates: poiché il modello simulato prevede funzioni con massima arietà 1, ogni istruzione update si presenta nella forma $f_k(f_0 f_1 \ldots f_{k-1}) = g_0 g_1 \ldots g_l$. L'istruzione simulante è `set` $f_0 f_1 \ldots f_k$ to $g_0 g_1 \ldots g_l$ con f_i e g_i i puntatori corrispondenti alle relative funzioni;

Importazione di elementi: l'istruzione per macchine a stati astratti

```
import v_1 ... v_k
    R
endimport
```

può essere resa con il frammento di codice per macchine a modificazione di memoria

```
new New_1
⋮
new New_k
R'
```

dove R' è il corrispettivo di R e ogni New_i corrisponde a v_i

Costrutti condizionali: volendo simulare l'istruzione if g then R endif, si chiami R' la sequenza di istruzioni per SMM che simula R. A seconda della natura di g, il controllo della macchina di Schönhage viene trasferito a parti diverse del codice. Nel caso più semplice, se y è un termine booleano e F è la parola ad esso corrispondente, il costrutto viene simulato nella seguente maniera: l'istruzione if g then R è simulata dalla porzione di codice

```
    if F = True then goto ℒ
    goto ℒ'
ℒ : R'
ℒ':
```

con $\mathcal{L}$ e $\mathcal{L}'$ etichette.

Come già detto, serve individuare una funzione di mappatura ϕ per realizzare la simulazione in lock-step. Essa è così descritta:

- se in A_i $f_k(f_0 f_1 \dots f_{k-1}) = x$ allora in $\phi(A_i)$ $f_0 f_1 \dots f_k$ indica x;
- se A_i ha x in input allora $\phi(A_i)$ lo stesso input;
- se β è l'output generato tra gli stati A_{i-1} e A_i allora è anche l'unico output emesso tra $\phi(A_{i-1})$ e $\phi(A_i)$;
- se Mode = Initial in A_i allora $\phi(A_i)$ è lo stato iniziale. Se Mode = Final in A_i allora $\phi(A_i)$ è lo stato finale. Altrimenti $\phi(A_i)$ non è iniziale nè finale.

Poiché la simulazione dello stato iniziale è già stata descritta, per poter dimostrare la seconda parte del teorema è sufficiente spiegare che per ogni update $f(t) := t_0$ lanciato nello stato A_{k-1} della ASM allora set t to t_0

è eseguito tra gli stati B_{k-1} e B_k dalla macchina di Schönhage simulante. Nel caso di una regola di update, il caso è banale: la transazione tra t e t_0 è diretta. Se si tratta di una regola di importazione, le modalità con le quali un nuovo elemento viene aggiunto sono rispettate in ogni aspetto dall'esecuzione dell'istruzione `new`. In una regola condizionale, se la regola g_i è valutata come vera in A_{i-1} allora la simulazione della regola viene eseguita tra A_{i-1} e A_i. Infine, i blocchi di regole sono update atomici che, pertanto, riportano all'analogo caso elementare. Dunque gli aggiornamenti eseguiti nello stato A_{k-1} corrispondono alla simulazione che avviene tra gli stati Bk_1 e B_k. Dunque $B_i = \phi A_i$.

Capitolo 3

Evolving Graph Structures

Scopo di questo capitolo è la presentazione del modello introdotto da Leivant e Marion in [11], sul quale gli stessi autori hanno dimostrato alcuni teoremi di complessità computazionale. Il linguaggio imperativo da essi proposto prende inspirazione dalle macchine di Schönhage e dalle macchine a stati astratti di Gurevich, inserendosi quindi all'interno del contesto descritto fino a questo punto. Il fine di questo studio è l'introduzione di un metodo di analisi statica in grado di garantire la fattibilità dal punto di vista di complessità temporale di programmi su strutture dati astratte.

Questo risultato si inserisce all'interno della cosiddetta Complessità Computazionale Implicita (ICC), il cui studio si basa su metodi divisibili in descrittivi (cioè riguardanti la Teoria dei Modelli Finiti, che si occupa della relazione tra un linguaggio formale e la sua semantica) ed applicativi (cioè quelli che definiscono restrizioni su programmi e tecniche di dimostrazione per garantire che un programma rispetti dati upper bound di complessità computazionale). Il metodo applicativo alla base di questa dimostrazione è la *ramificazione*, anche nota come *tiering*. Essa si basa sul rapporto tra il tempo di esecuzione di un programma ed il tipo di flusso di informazione che avviene durante essa. È possibile limitare in maniera effettiva il flusso regolandolo per mezzo di una relazione di precedenza (ad esempio, dal livello più alto al livello più basso). Nel presente caso, la ramificazione è applicata ai comandi del linguaggio imperativo ed alle espressioni che denotano gli elementi della struttura alla base delle evolving graph structures.

Il risultato principale di Leivant e Marion stabilisce che una funzione su una evolving graph structure è computabile in tempo polinomiale se e solo se è computabile da un programma terminante che utilizzi ramificazione nella manipolazione del grafo stesso, premesso che gli archi letti o creati nella stessa iterazione di un costrutto loop abbiano la stessa etichetta. Questa estensione introduce un nuovo utilizzo della ramificazione nello studio della

complessità computazionale implicita.

3.1 Evolving Graph Structures

In questo capitolo la presentazione delle Evolving Graph Structures, abbreviato EGS, segue da vicino la descrizione fornita dagli autori in [11]. Allo stesso tempo però, verranno sottolineate alcune somiglianze strutturali che ricordano le macchine di Schönhage ed i componenti ausiliari per esse definiti. Questa similitudine sarà ancora più evidente nel capitolo seguente, al momento di descrivere la simulazione effettuata da parte di questo ultimo modello.

3.1.1 Sorted Partial Structures

Le strutture grafo utilizzate in questo modello sono **sorted structure**; in particolare, verranno considerate quelle strutture che vedono come sorte distinte quella dei vertici $\mathcal{V}$ e quella dei dati $\mathcal{D}$. In una sorted structure, se $\mathcal{V}$ e $\mathcal{D}$ sono sorte, una funzione f è di tipo $\mathcal{V} \to \mathcal{D}$ se il suo dominio consiste negli elementi della struttura di sorta $\mathcal{V}$ e il suo codominio in elementi di sorta $\mathcal{D}$.

Si considerino grafi i cui archi sono etichettati con nomi chiamati **azioni** ed ogni nodo ha al più un arco uscente per ogni data etichetta. È possibile rappresentare questi grafi facendo corrispondere ad ogni azione una funzione parziale. L'equazione $f(u) = v$ indica che l'arco uscente da u avente etichetta f è diretto al nodo v. Analogamente, $f(u) =$undef indica che nessun arco uscente da u ha etichetta f. Volendo rappresentare nell'ambiente delle strutture il concetto di funzione parziale è possibile utilizzare una costante speciale **nil**, che si assume avere sorta singoletto. Una funzione parziale f di tipo $\mathcal{V} \to \mathcal{D}$ è quindi una funzione $\mathcal{V} \to (\mathcal{D} \cup \{\mathbf{nil}\})$. Si noti la forte somiglianza tra **nil** e la struttura ausiliaria **null** delle Macchine di Schönhage.

3.1.2 Strutture Grafo

Verranno prese in considerazione da questo momento le strutture parziali aventi tre sorte distinte. La sorta $\mathcal{V}$ di vertici e, così come per le macchine a stati astratti, una sorta $\mathcal{R}$ di vertici riservati, necessari per la creazione di nuovi nodi. Infine la sorta $\mathcal{D}$ dei dati. Un **graph vocabulary** Σ è un vocabolario composto da cinque identificatori:

- $\mathbb{V}$, l'insieme dei nomi dei vertici;

- $\mathbb{D}$, l'insieme dei nomi dei dati;

- $\mathbb{F}$, l'insieme dei nomi di funzioni per archi etichettati, di tipo $\mathcal{V} \rightarrow \mathcal{V}$;
- $\mathbb{G}$, l'insieme dei nomi di funzioni per dati, di tipo $\mathcal{V} \rightarrow \mathcal{D}$;
- $\mathbb{R}$, l'insieme dei nomi di relazioni di tipo $\tau \times \ldots \times \tau$ dove ogni τ può appartenere a $\mathcal{V}$ oppure a $\mathcal{D}$.

Gli elementi di questi insiemi verranno identificati con $\mathbf{v} \in \mathbb{V}$, $\mathbf{d} \in \mathbb{D}$, $\mathbf{f} \in \mathbb{F}$, $\mathbf{g} \in \mathbb{G}$ e $\mathbf{R} \in \mathbb{R}$.

Dato un vocabolario Σ così descritto, una **Σ-struttura** S consiste in un universo finito di vertici $\mathcal{V}_S$, un universo riserva potenzialmente infinito $\mathcal{R}_S$, un universo di dati $\mathcal{D}_S$, un oggetto distinto $\perp$ che rappresenta **nil** ed una interpretazione $\mathbf{A}_S$ per ogni Σ-identificatore $\mathbf{A}$: $\mathbf{v}_S \in \mathcal{V}_\mathcal{S}$, $\mathbf{d}_S \in \mathcal{D}_\mathcal{S}$, $\mathbf{f}_S \in [\mathcal{V}_S \rightarrow \mathcal{V}_S]$ (una funzione parziale), $\mathbf{g}_S \in [\mathcal{V}_S \rightarrow \mathcal{D}_S]$ (una funzione dati) e per ogni elemento dell'insieme dei nomi di relazioni $\mathbf{R}$ una relazione $\mathbf{R}_S \subseteq \tau_S \times \ldots \times \tau_S$. Si noti la mancanza di funzioni su dati, così come di funzioni con arietà maggiore di 1. La rappresentazione degli archi per mezzo di funzioni rispetta il determinismo degli archi dei grafi in questione.

Tarjan definisce in [18] una struttura simile, parlando però di *record* ed *item* piuttosto che di vertici ed archi. Si può notare infatti che sussiste una certa similitudine con il costrutto `struct` del linguaggio di programmazione C e con i linguaggi di programmazione object oriented, in cui ogni oggetto può essere identificato da un vertice e lo stato dell'oggetto è dato dai suoi campi (cioè funzioni unarie). La restrizione di una struttura grafo S alla singola sorta $\mathcal{V}$ di vertici può essere rappresentata per mezzo di un multigrafo in cui gli unici archi sono quelli tali che $f(u) = v$ per qualche u e v. Così come per le macchine di Schönhage, il numero massimo di archi uscenti dai nodi di ogni grafo è limitato dal numero di etichette di Σ.

3.1.3 Espressioni

Dato un insieme $\mathbb{X}$ di variabili per i vertici, un insieme $\mathbb{Y}$ di variabili per i dati e gli identificatori prima definiti, è possibile generare le espressioni come segue:

$V \in \text{VExpr} ::= X \mid \mathbf{nil} \mid \mathbf{v} \mid \mathbf{f}(V)$ con $X \in \mathbb{X}$
$D \in \text{DExpr} ::= Y \mid \mathbf{d} \mid \mathbf{g}(V)$ con $Y \in \mathbb{Y}$
$B \in \text{BExpr} ::= V = V \mid D = D \mid \neg(B) \mid \mathbf{R}(E_1 \ldots E_n)$ con $\mathbf{R}{:}\tau^n, E_i : \tau$

3.1.4 Programmi

I programmi che modificano le strutture sopra descritte sono scritti in un linguaggio imperativo che supporta i puntatori.

$$P \in \text{Prg} ::= X := V \mid V := D \mid \mathbf{f}(X) := V \mid \mathbf{g}(X) := D \mid \mathbf{New}(X)$$
$$\mid \mathbf{skip} \mid P; P \mid \mathbf{if}(B)\{P\}\{P\} \mid \mathbf{while}(B)\{P\}$$

dove l'espressione B è chiamata guardia. Insieme ad ogni programma è dato un insieme di variabili in *input* $\mathbb{X}_0 \subset \mathbb{X}$.

Un interessante esempio è il seugente algoritmo su grafi descritto da Tarjan in [18]. Siano q e r due insiemi disgiunti, rappresentati da liste. Il vertice a capo di ogni lista funge anche da nome per identificarla. Per ottenere questa rappresentazione viene utilizzata la funzione parziale `next`, con l'ausilio della funzione `parent` che mappa ogni nodo alla testa del suo insieme-lista. L'algoritmo prende in input q e r e restituisce una lista data dalla loro unione, mantenendo r come nome dell'insieme finale.

```
while (q≠nil) { save := next(q);
                parent(q) := r;
                next(q) := next(r);
                next(r) := q;
                q := save }
```

3.1.5 Evolving Structures

Per ogni programma su una Σ-struttura $\mathcal{S}$ esiste uno **store** (o ambiente, o valutazione), cioè una funzione $\mu = \mu_X \cup \mu_Y$ con $\mu_X : \mathbb{X} \to \mathcal{V}_\mathcal{S} \cup \{\perp\}$ e $\mu_Y : \mathbb{Y} \to \mathcal{D}_\mathcal{S}$. Una **$\Sigma$-configurazione** è una coppia formata da una Σ-struttura $\mathcal{S}$ e da uno store μ. La parola configurazione è indicativa della natura dinamica ed in evoluzione di questi due elementi.
I comandi scritti nel linguaggio imperativo visto in 3.1.4 sono semanticamente interpretati come funzioni parziali da una configurazione all'altra. Ad esempio $\mathbf{New}(X)$ sposta un elemento da $\mathcal{R}_\mathcal{S}$ a $\mathcal{V}_\mathcal{S}$ ed aggiorna lo store facendo puntare X al nuovo vertice. La configurazione risultante è espressa come $(\mathcal{S}, \mu)[\nu X]$. Il comando $\mathbf{f}(X)$:=(V) invece modifica la semantica della funzione parziale $\mathbf{f}_\mathcal{S}$ in maniera tale che se $a = \mu(X)$ e w è il valore di V in $(\mathcal{S}, \mu)$ allora l'arco etichettato $\mathbf{f}$ da v a u venga ridiretto verso w. La configurazione risultante è espressa come $(\mathcal{S}, \mu)[\,\mathbf{f}(X) \leftarrow w]$.

Volendo fare un paragone con le ASM di Gurevich, si può notare che nelle evolving structures gli unici identificatori dinamici sono quelli per descrivere gli archi. Gli stati delle ASM possono essere visti come strutture, mentre Leivant e Marion identificano la progressione della computazione in configurazioni, come appena visto. Verrà ora descritta la semantica di espressioni e programmi.

Semantica di Espressioni

Si introducono ora le regole di valutazione per Σ-espressioni E. Data $(\mathcal{S}, \mu)$ Σ-configurazione, si scrive $\mathcal{S}, \mu \models E \overset{e}{\Rightarrow} a$ per indicare che E valuta l'elemento a di $\mathcal{S}$

$$\frac{\mathbf{b} \in \mathbb{V} \cup \mathbb{D}}{S, \mu \models \mathbf{b} \overset{e}{\Rightarrow} \mathbf{b}_S} \quad \frac{Z \in \mathbb{X} \cup \mathbb{Y}}{S, \mu \models Z \overset{e}{\Rightarrow} \mu(Z)} \quad \frac{S, \mu \models E \overset{e}{\Rightarrow} a \quad \mathbf{h} \in \mathbb{F} \cup \mathbb{G}}{S, \mu \models \mathbf{h}(a) \overset{e}{\Rightarrow} \mathbf{h}_S(a)}$$

$$\frac{S, \mu \models E_i \overset{e}{\Rightarrow} a_i \quad \langle a_1, \ldots, a_n \rangle \in \mathbf{R}_S}{S, \mu \models \mathbf{R}(E_1, \ldots, E_n)} \quad \frac{S, \mu \models E_i \overset{e}{\Rightarrow} a_i \quad \langle a_1, \ldots, a_n \rangle \notin \mathbf{R}_S}{S, \mu \models \neg\mathbf{R}(E_1, \ldots, E_n)}$$

Fig. 3.1: Regole di semantica per espressioni

Semantica dei Programmi

La semantica dei programmi è definita come nella Figura 3.2

$$\frac{S, \mu \models E \overset{e}{\Rightarrow} a}{S, \mu \models Z := E \overset{s}{\Longrightarrow} S, \mu[Z \leftarrow a] \models \mathbf{skip}} \quad \frac{}{S, \mu \models \mathbf{New}(X) \overset{s}{\Longrightarrow} (S, \mu)[\nu X] \models \mathbf{skip}}$$

$$\frac{S, \mu \models X \overset{e}{\Rightarrow} a \quad S, \mu \models V \overset{e}{\Rightarrow} b}{S, \mu \models \mathbf{f}(X) := V \overset{s}{\Longrightarrow} S, \mu[\mathbf{f}(a) := b] \models \mathbf{skip}}$$

$$\frac{S, \mu \models P_1 \overset{s}{\Longrightarrow} S', \mu' \models P_1'}{S, \mu \models P_1; P_2 \overset{s}{\Longrightarrow} S', \mu' \models P_1'; P_2} \quad \frac{S, \mu \models P_1 \overset{s}{\Longrightarrow} S', \mu' \models \mathbf{skip}}{S, \mu \models P_1; P_2 \overset{s}{\Longrightarrow} S', \mu' \models P_2}$$

$$\frac{S, \mu \models B}{S, \mu \models \mathbf{if}(B)\{P_0\}\{P_1\} \overset{s}{\Longrightarrow} S, \mu \models P_0} \quad \frac{S, \mu \models \neg B}{S, \mu \models \mathbf{if}(B)\{P_0\}\{P_1\} \overset{s}{\Longrightarrow} S, \mu \models P_1}$$

$$\frac{S, \mu \models B}{S, \mu \models \mathbf{while}(B)\{P\} \overset{s}{\Longrightarrow} S, \mu \models P; \mathbf{while}(B)\{P\}} \quad \frac{S, \mu \models B}{S, \mu \models \mathbf{while}(B)\{P\} \overset{s}{\Longrightarrow} S, \mu \models \mathbf{skip}}$$

Fig. 3.2: Regole di semantica per programmi

La frase $\mathcal{S}, \mu \models P \overset{s}{\Rightarrow} \mathcal{S}', \mu' \models P'$ sta ad indicare che la valutazione di un programma P partendo della configurazione $(\mathcal{S}, \mu)$ riduce a P' in configurazione $(\mathcal{S}', \mu')$. Una configurazione $(\mathcal{S}, \mu)$ è detta **iniziale** se $\mu(X)$=**nil** per ogni variabile (non di input) X. Un programma P **computa** la funzione parziale $[\![P]\!]$ con configurazione iniziale come input definita da: $[\![P]\!](\mathcal{S}, \mu) = (\mathcal{S}', \xi)$ se e solo se $\mathcal{S}, \mu \models P \overset{s}{\Rightarrow}^* \mathcal{T}, \xi \models \mathbf{skip}$.

Si dice che che un programma P viene eseguito in tempo t su input $(\mathcal{S}, \mu)$, scritto $Time_P(\mathcal{S}, \mu) = t$, quando $\mathcal{S}, \mu \models P(\overset{s}{\Rightarrow})^t \mathcal{T}, \xi \models$ **skip** per qualche $(\mathcal{T}, \xi)$. La **dimensione** $| \mathcal{S}, \mu |$ di una configurazione $(\mathcal{S}, \mu)$ è il numero n di elementi dell'universo dei vertici V. Il numero di archi è al più n^2, e non incide dunque sul risultato finale, così come la dimensione dell'universo di dati poiché essi non vengono modificati dal programma in esecuzione. Un programma P viene eseguito in **tempo polinomiale** se esiste $k > 0$ tale che $Time_P(\mathcal{S}, \mu) \leq k\cdot \mid \mathcal{S}, \mu \mid^k$ per tutte le configurazioni $(\mathcal{S}, \mu)$.

3.2 Programmi Ramificabili

La **ramificazione**, o tiering, fu introdotta da Leivant in [19] (sebbene alcune sue forme ristrette fossero utilizzate già negli anni precedenti [20]) con lo scopo di fornire un metodo per il controllo sintattico dei programmi a runtime. Per prima cosa, si vuole adattare la ramificazione alle EGS, prestando particolare attenzione alla loro evoluzione nel corso della computazione. Si lavorerà in particolare con il reticolo finito $\mathbb{T} = (\{\mathbf{0,1}\}, \leq, \mathbf{0}, \vee, \wedge)$, dove $\{\mathbf{0,1}\}$ sono chiamati **tier** e α, β vengono usati come parametri per indicare i tier nel corso della trattazione.

Dato $\mathbb{T}$, la coppia (Γ, Δ) viene chiamata **$\mathbb{T}$-environment**; Γ assegna un tier ad ogni variabile di $\mathbb{X}$ e Δ assegna ad ogni nome di funzione $\mathbf{f}\colon \mathcal{V} \to \mathcal{V}$ una o più espressioni nella forma $\alpha \to \beta$ tali che valga una delle due seguenti affermazioni:

- tutti i tipi in $\Delta(\mathbf{f})$ sono nella forma $\alpha \to \alpha$. In tal caso $\mathbf{f}$ è detto **stabile** nell'environment;
- tutti i tipi in $\Delta(\mathbf{f})$ sono nella forma $\alpha \to \beta$, con $\beta < \alpha$. In tal caso $\mathbf{f}$ **riduce** nell'environment.

Una **asserzione di tiering** è un giudizio nella forma $\Gamma, \Delta \vdash V : \alpha$, dove V è un'espressione vertice appartenente a VExp e (Γ, Δ) è un $\mathbb{T}$-environment. Le corrette asserzioni di tiering vengono generate dal seguente sistema di tiering per espressioni di vertici, espressioni booleane e programmi.

Dato un reticolo $\mathbb{T}$ un programma P è **$\mathbb{T}$-ramificabile** se esiste un $\mathbb{T}$-environment (Γ, Δ) tale che $\Gamma, \Delta \vdash P : \alpha$ per qualche α e tale che $\Gamma(X) = \mathbf{1}$ per ogni variabile in input $X \in \mathbb{X}_0$ dove $\mathbb{X}_0$ è l'insieme di variabili in input. Quindi un programma ramificabile può essere visto come un programma arricchito con informazioni riguardanti il tiering.

Lemma 1 (Subject Reduction)
Se $\mathcal{S}, \mu \models P \overset{s}{\Rightarrow} \mathcal{S}', \mu' \models P'$ *e* $\Gamma, \Delta \vdash P : \alpha$ *allora* $\Gamma, \Delta \vdash P' : \alpha$

$$\frac{}{\Gamma,\Delta\vdash \mathbf{c}:\alpha}\quad \frac{\Gamma(X)=\alpha}{\Gamma,\Delta\vdash X:\alpha}\quad \frac{\alpha\to\beta\in\Delta(\mathbf{f})\quad \Gamma,\Delta\vdash V:\alpha}{\Gamma,\Delta\vdash \mathbf{f}(V):\beta}$$

$$\frac{\Gamma,\Delta\vdash V_i:\alpha}{\Gamma,\Delta\vdash \Gamma,\Delta\vdash \mathbf{R}(V_1,\ldots,V_n):\alpha}\quad \frac{\Gamma,\Delta\vdash V_i:\alpha}{\Gamma,\Delta\vdash V_0=V_1:\alpha}$$

$$\frac{\Gamma,\Delta\vdash X:\alpha\quad \Gamma,\Delta\vdash V:\alpha}{\Gamma,\Delta\vdash X:=V:\alpha}\quad \frac{\Gamma,\Delta\vdash \mathbf{f}(X):\alpha\quad \Gamma,\Delta\vdash V:\alpha}{\Gamma,\Delta\vdash \mathbf{f}(X):=V:\alpha}$$

$$\frac{\Gamma,\Delta\vdash X:\mathbf{0}}{\Gamma,\Delta\vdash \mathbf{New}(X):\mathbf{0}}\quad \frac{\Gamma,\Delta\vdash B:\alpha\quad \Gamma,\Delta\vdash P:\alpha}{\Gamma,\Delta\vdash \mathbf{while}(B)\{P\}:\alpha}\quad (\mathbf{0}<\alpha)$$

$$\frac{}{\Gamma,\Delta\vdash \mathbf{skip}:\mathbf{0}}\quad \frac{\Gamma,\Delta\vdash P:\alpha\quad \Gamma,\Delta\vdash P':\beta}{\Gamma,\Delta\vdash P';P':\alpha\vee\beta}$$

$$\frac{\Gamma,\Delta\vdash B:\alpha\quad \Gamma,\Delta\vdash P_i:\alpha}{\Gamma,\Delta\vdash \mathbf{if}(B)\{P_0\}\{P_1\}:\alpha}\quad \frac{\Gamma,\Delta\vdash P:\beta}{\Gamma,\Delta\vdash P:\alpha}\quad (\beta\leq\alpha)$$

Fig. 3.3: Regole di tiering

Lemma 2 (Inferenza di Tipo)
Dati P programma e $\mathbb{T}$ reticolo, se P sia $\mathbb{T}$-ramificabile o meno è decidibile in tempo polinomiale

Dimostrazione. Si associ ad ogni variabile vertice X una variabile tier α_X, e ad ogni funzione $\mathbf{f}\in\mathbb{F}$ due variabili α_f e β_f (in maniera tale che $\alpha_f\to\beta_f$ sia un possibile tiering per $\mathbf{f}$). Per mezzo delle regole di tipaggio per tier si viene a formare un insieme di vincoli lineari per queste variabili tier, un problema che è decidibile in tempo polinomiale. □

3.2.1 Loop Stazionari e Strettamente Modificanti

A seconda delle funzioni che vengono modificate o visitate nel corso dell'esecuzione di un programma vengono date alcune definizioni. Una funzione $\mathbf{f}$ è:

- **Ispezionata** in P se, in P, occorre in qualche assegnamento $X:=V$ o nella guardia di un loop o di una istruzione che causa ramificazione (cioè nella guardia di un **while** o di un **if**). Ad esempio $\mathbf{f}$ è ispezionato in $X:=\mathbf{f}(V)$ così come in `if` $(\mathbf{f}(X)\neq$`nil`$)\{P\}\{P'\}$;

- **Modificata** in P se in esso occorre in un assegnamento $\mathbf{f}(X):=V$.

Dato un reticolo $\mathbb{T}$ ed un $\mathbb{T}$-environment (Γ, Δ), se un loop **while**$(B)\{P\}$ è un tier di α allora $\Gamma, \Delta \vdash B : \alpha$. Esso è detto:

- **Stazionario** se al suo interno nessuna funzione **f**$\in \mathbb{F}$ di tipo $\alpha \to \alpha$ viene modificata;
- **Strettamente modificante** se modifica funzioni di tipo $\alpha \to \alpha$ ma al massimo una di esse è ispezionata, cioè gli archi che sono sia creati che letti all'interno del loop hanno la stessa etichetta.

Ad esempio, nell'algoritmo di Tarjan **next** è modificato ed ispezionato, mentre **parent** è modificato ma non ispezionato. Pertanto il loop che li contiene è strettamente modificante. Un esempio di programma ramificato è il seguente. Esso inserisce l'albero T all'interno di un albero binario ordinato la cui radice è puntata da x.

```
if (x^1 = nil)
   {x^1 :=T:1;}
   {while ((x^1 ≠ nil) and (key(T^1) ≠ key(x^1)))
            {if (key(T^1))<(key(x^1)) {p^1:=x^1; x^1:=left(x^1)^1}
                                      {p^1:=x^1; x^1:=right(x^1)^1}} :1;
if (key(T^1) < key(p^1)) {left(p^1):= T^1:1}
            {right(p^1):= T^1:1}
```

Le variabili in input sono x e T. Se l'albero di partenza è vuoto, allora si fa puntare x direttamente a T. Dopodiché, fintanto che l'indice di T è inferiore a quello della testa di x, il controllo si sposta seguendo il ramo sinistro dell'albero binario utilizzando in p come contatore. Lo stesso accade verso il ramo destro se l'indice di T è superiore. Una volta raggiunto il punto opportuno, T viene inserito per mezzo di puntatori uscenti da P.

3.3 Ramificazione Stretta e Polinomialità

Dati un reticolo $\mathbb{T}$ e $\Gamma, \Delta \vdash P : \alpha$ si dice che che (Γ, Δ) è una ramificazione **stretta** di P se Γ è un tiering iniziale ed ogni loop di P è stazionario o strettamente modificante. Inoltre, P è **strettamente-ramificabile** se ha una $\mathbb{T}$-ramificazione stretta (Γ, Δ) con Γ iniziale per qualche $\mathbb{T}$ non banale.

Teorema 3.3.1. *Una funzione su una struttura grafo è computabile in tempo polinomiale se e solo se è computabile da un programma terminante e strettamente ramificabile.*

La dimostrazione si divide in due parti. Si comincia dimostrando che ogni programma strettamente ramificato computa in tempo polinomiale una funzione su configurazioni. L'idea dietro alla dimostrazione è che un tale

programma preserva la proprietà di non avere vertici assegnati a variabili di tier differenti. Ciò garantisce che i vertici possano essere ramificati in maniera non ambigua. Grazie alle regole di tiering è possibile dire che un programma P di tier $\mathbf{0}$ non ha loop, e verrà quindi valutato in un numero di passi $\leq\mid P\mid$. Allo stesso modo, il valore di una variabile di tier $\mathbf{1}$ dipende solo dai vertici con lo stesso tier. Una implicazione di questo fatto è che il numero di iterazioni di un loop è limitato dal numero di possibili configurazioni generate dal suo corpo. La restrizione alle ramificazioni strettamente modificanti garantisce la polinomialità di questo numero.

3.3.1 Non-Interferenza

Lemma 3 (Restrizione)
Sia (Γ,Δ) un environment. Se $\Gamma,\Delta\vdash P$: $\mathbf{0}$ allora $\Gamma(X)=\mathbf{0}$ per ogni variabile X assegnata in P.

Dimostrazione. Ovvia per induzione strutturale. □

Si noti che il programma P di tier $\mathbf{0}$ non può avere loop ed è pertanto valutato in $\mid P\mid$ passi. Un tiering di vertici Γ è **compatibile** con uno store μ se $\Gamma(X)\neq\Gamma(X')$ implica $\mu(X)\neq\mu(X')$ per ogni $X,X'\in\mathbb{X}$. Si definisce Γ **tiering iniziale** se $\Gamma(X)$ vale $\mathbf{1}$ per ogni $X\in\mathbb{X}_0$, $\mathbf{0}$ altrimenti. Quindi un tiering iniziale è sempre compatibile con uno store iniziale.

Lemma 4 (Compatibilità)
Siano $\Gamma,\Delta\vdash P:\alpha$ e $\mathcal{S},\mu\models P\overset{s}{\Rightarrow}\mathcal{S}',\mu'\models P'$. Allora se μ è compatibile con Γ lo è anche μ'.

Dimostrazione. Ovvia per induzione strutturale su P. □

Il passo seguente è dimostrare che il tiering, se compatibile con la configurazione iniziale, garantisce non interferenza con i valori di tier inferiore durante l'esecuzione di programmi con tier maggiore. Simili risultati, sebbene più semplici o applicati ad altri campi, furono precedentemente osservati in [17] e [19].

Il (Γ,Δ)-collapse di una configurazione $(\mathcal{S},\mu)$ è la configurazione $(\mathcal{S}_\Delta,\mu_\Gamma)$ dove $\mu_\Gamma(X)$ vale $\mu(X)$ se $\Gamma(X)=\mathbf{1}$, altrimenti è indefinita. Allo stesso tempo $\mathcal{S}_\Delta$ è la struttura identica a $\mathcal{S}$ fatta eccezione per gli $\mathbf{f}$ tali che $(\mathbf{1}\to\mathbf{1})\notin\Delta(\mathbf{f})$, che vengono interpretati come $\emptyset$. Il risultato è che $(\mathcal{S}_\Delta,\mu_\Gamma)$ scarta i vertici non raggiungibili da qualche variabile di tier $\mathbf{1}$ usando archi di tipo $(\mathbf{1}\to\mathbf{1})$.

Lemma 5 (Collapsing)
Siano $\Gamma,\Delta\vdash P:\alpha$ e $\mathcal{S},\mu\models P\overset{s}{\Rightarrow}\mathcal{S}',\mu'\models P'$. Esiste una configurazione $((\mathcal{S}'',\mu''))$ tale che $\mathcal{S}_\Delta,\mu_\Gamma\models p\overset{s}{\Rightarrow}\mathcal{S}'',\mu''\models P'$ e $(\mathcal{S}''_\Delta,\mu''_\Gamma)=(\mathcal{S}'_\Delta,\mu'_\Gamma)$.

Dimostrazione. Ovvia per induzione strutturale su P. □

In altre parole, se un programma restituisce in output vertici di tier **1** essi non dipendono da vertici di tipo **0**, né da archi che non hanno tier **1**→**1**.

3.3.2 Limiti Polinomiali

Si vuole a questo punto dimostrare che il numero di passi della computazione di un programma P strettamente modificante avente un dato tier è al più polinomiale in termini della dimensione della struttura grafo.

Lemma 6 (Soundness)
Sia $\Gamma, \Delta \vdash P : \alpha$ *con* P *strettamente modificante. Esiste* $k > 0$ *tale che per ogni struttura grafo* $\mathcal{S}$ *ed ogni store* μ *compatibile con* Γ*, se* $\mathcal{S}, \mu \models P \overset{s}{\Rightarrow} \mathcal{S}'', \mu'' \models P'$ *allora* $\mathcal{S}, \mu \models P\ (\overset{s}{\Rightarrow})^t\ \mathcal{S}'', \mu'' \models P'$ *per qualche* $t < k + \mid \mathcal{S} \mid^k$.

La prova di questo lemma verrà discussa al termine di questa sezione. È necessario prima introdurre un'ulteriore dimostrazione. Sia $\mathcal{G}$ un digrafo con out-degree 1. Diciamo che un insieme di vertici C **genera** $\mathcal{G}$ se ogni vertice in $\mathcal{G}$ è raggiungibile da un cammino che inizia in C. Il seguente lemma, dimostrato in [11], fornisce un limite superiore polinomiale sul numero di digrafi con k generatori.

Lemma 7 (Digrafi)
Il numero di digrafi modulo isomorfismo con n *vertici ed un generatore di dimensione* k *è* $\leq n^{2k^2}$.

È possibile a questo punto dimostrare il Lemma 6.

Dimostrazione. Si procede per induzione strutturale su P. Il caso più cruciale è quello in cui $P = \textbf{while}(B)Q$ per qualche B, Q. Date $X_1, \ldots, X_m$ le variabili vertici di B, esse sono tutte di tier **1** (come conseguenza delle regole di tiering). Se Q aggiorna solo gli archi che non sono ispezionati in P (inclusa la guardia B) allora né l'esecuzione di Q né la valutazione di B subiscono modifiche, cioè nel corso della computazione tutte le configurazioni presenteranno gli stessi vertici di tier **1**, senza modifiche sugli archi che influiscano sull'esecuzione di P (Lemma 5). Pertanto il valore di verità di B ad ogni chiamata di Q è determinato dalla combinazione di valori assegnati alle variabili X_i, mentre i cambiamenti strutturali causati da Q non hanno conseguenze sulle chiamate seguenti a Q stesso. Assumendo che P termini, le combinazioni di valori per $X_1, \ldots, X_m$ devono essere tutte diverse. Dato n numero di vertici di tier **1** vale che $n \leq \mid \mathcal{S} \mid$ e dunque esistono n^m possibili combinazioni di valori delle variabili. Per ipotesi induttiva Q termina in tempo polinomiale, e dunque anche P.

Si supponga a questo punto che Q aggiorni gli archi che vengono ispezionati in P. Poiché è stato premesso che P è strettamente modificante, si può affermare che update riguardano la stessa funzione $\mathbf{f} \in \mathbb{F}$. Sia C l'insieme dei valori iniziali delle variabili che occorrono in P (cioè $X_1, \ldots, X_m$ e possibili altre) e U l'insieme di vertici raggiungibili da C percorrendo un cammino di tier **1**. Per il Lemma 5, l'esecuzione di P incluse le chiamate iterative di B e Q, presenta solo vertici di U come valore di tier **1**. Inoltre U è generato da C, la cui dimensione è fissa per sintassi di P. Ne segue, per il Lemma 7, che il numero di configurazioni è polinomiale rispetto alla grandezza di U, che è limitata dalla dimensione $\mid \mathcal{S} \mid$ dell'universo dei vertici della struttura S. □

Proposizione (Completezza)
Ogni funzione su strutture grafo a tempo polinomiale è computabile da un programma terminante e strettamente ramificabile.

Dimostrazione. Sia M una macchina di Turing su un alfabeto Σ che computa una funzione unaria su strutture grafo attraverso la loro descrizione per mezzo di stringhe. Si assuma che essa operi in tempo $\mid k \cdot n^k$ utilizzando un nastro per la lettura ed un nastro per la computazione. È possibile simulare M per mezzo di un programma strettamente ramificabile su strutture grafo $\Gamma, \Delta \vdash P : \mathbf{1}$ come segue. Entrambi i nastri sono simulati per mezzo di una lista puntata di record con campi **left**, **right** e **val**. I primi due ritornano un puntatore, il terzo una lettera dell'alfabeto che è rappresentata come dato. Il nastro per l'input ha tier **1**, mentre il nastro per la computazione ha tier **0** ed è inizialmente vuoto. Esso viene riempito progressivamente per mezzo di istruzioni **New**. All'interno della configurazione viene aggiunto un clock formato da k loop annidati, così come mostra il seguente esempio a due loop. Il puntatore all'input dei nastri è chiamato **head**.

u$^{\mathbf{1}}$:= head$^{\mathbf{1}}$:**1**
while (u$^{\mathbf{1}}$ $\neq$ **nil**
{ v$^{\mathbf{1}}$:=head**1**:**1**;
while (v$^{\mathbf{1}}$ $\neq$ **nil**)
{ v$^{\mathbf{1}}$:= **succ**(v$^{\mathbf{1}}$):**1**;
{*funzione di transizione*} : **0**}: **1**
u$^{\mathbf{1}}$:= **suc**(u$^{\mathbf{1}}$):**1**}:**1**

□

3.3.3 Risultati Aggiuntivi

La pubblicazione [11] di Leivant e Marion si conclude con una sezione che descrive la possibilità di aggiungere ricorsione al linguaggio di programmazione presentato, con particolare attenzione nei confronti della ricorsione

lineare (che ammette al massimo una chiamata ricorsiva nel corpo della procedura ricorsiva). Nello stesso articolo è possibile trovare una dimostrazione del seguente teorema:

Teorema 3.3.2. *Nel suo dominio di computazione, un programma strettamente ramificabile con chiamate ricorsive lineari è computabile in tempo polinomiale.*

Dunque, l'aggiunta della ricorsione non inficia il risultato polinomiale precedentemente dimostrato.

Capitolo 4

EGS e SMM: simulazione

In questo capitolo si vuole dimostrare che è possibile costruire una macchina di Schönhage che simuli una data evolving graph structure. L'approccio scelto per questa dimostrazione passa per mezzo della costruzione di una Δ-struttura (la struttura di una SMM) che abbia le stesse caratteristiche della Σ-struttura di una EGS. Similarmente, verrà mostrato come simulare i programmi delle EGS utilizzando le istruzioni delle SMM. Nonostante sia stata necessaria l'aggiunta di nuove strutture ausiliarie, la maggior parte della dimostrazione risulta lineare ed è a tratti evidente come Leviant e Marion si siano in parte ispirati al modello di Schönhage.

La dimostrazione procederà riproponendo alcune definizioni delle componenti di una EGS, descrivendo di volta in volta le modalità di simulazione ed introducendo le strutture ausiliarie ad esse necessarie. Infine, si dimostrerà come la simulazione preservi l'esecuzione in tempo polinomiale del programma della struttura di partenza.

4.1 Insiemi ed Elementi Base

Come descritto nel capitolo precedente, la struttura di una EGS è caratterizzata da tre insiemi: l'insieme dei vertici $\mathcal{V}$, l'insieme di dati $\mathcal{D}$ e l'insieme dei vertici riservati $\mathcal{R}$. L'idea alla base della simulazione di queste componenti, che sono disgiunte, è la possibilità di rappresentare i loro elementi per mezzo di funzioni nullarie, aggiungendo cioè apposite etichette all'alfabeto Δ della SMM simulante.

Si comincia descrivendo il vocabolario Σ della EGS che verrà simulata. Esso è suddiviso in cinque insiemi di identificatori: l'insieme dei nomi dei vertici $\mathbb{V} = \{\mathbf{v}, \mathbf{v}_1, \ldots\}$, l'insieme dei nomi di dati $\mathbb{D} = \{\mathbf{d}, \mathbf{d}_1, \ldots\}$, l'insieme $\mathbb{F}$ di nomi di funzioni di tipo $\mathcal{V} \to \mathcal{V}$ per archi, l'insieme $\mathbb{G}$ di nomi di funzioni di tipo $\mathcal{V} \to \mathcal{D}$ per dati e l'insieme $\mathbb{R}$ di nomi di relazioni di tipo $\tau \times \ldots \times \tau$

dove τ è $\mathcal{V}$ o $\mathcal{D}$.

Dato un vocabolario Σ, una Σ-struttura S consiste in un universo finito di vertici $\mathcal{V}_S$, un universo di riserve potenzialmente infinito $\mathcal{R}_S$, un universo di dati $\mathcal{D}_S$, un elemento **nil** ed una interpretazione sort-correct per ogni elemento appartenente ai cinque insiemi che compongono Σ (dove $\mathbf{v}_S$ è la interpretazione di $\mathbf{v}$).

Tutti i componenti descritti fino a questo punto sono rappresentabili in una Δ-struttura con il solo uso di archi coerentemente etichettati. Per cominciare, vengono creati cinque sottoinsiemi di Δ, chiamati $\Delta_{\mathbb{V}}$, $\Delta_{\mathbb{D}}$, $\Delta_{\mathbb{F}}$, $\Delta_{\mathbb{G}}$ e $\Delta_{\mathbb{R}}$ e contenenti almeno una etichetta per ognuno degli elementi dell'insieme originario corrispondente. Si distingue ora tra i diversi tipi di elementi per stabilire quali e quante etichette utilizzare.

Costanti

Le costanti appartengono agli insiemi di nomi di vertici e di nomi di dati. Per ognuno degli elementi appartenenti a $\mathbb{V}$ e $\mathbb{D}$ si crea un'etichetta aggiungendola a $\Delta_{\mathbb{V}}$ o $\Delta_{\mathbb{D}}$. Essa verrà utilizzata su di un apposito arco uscente dal nodo centrale della Δ struttura, indirizzato verso il nodo che rappresenta il relativo elemento della Σ-struttura. Nel caso siano presenti eventuali elementi aggiuntivi come `root`, gli archi rappresentanti le costanti usciranno dal nodo d'ingresso alla parte di grafo non ausiliaria. La Figura 4.1 illustra una Δ-struttura con tre costanti, tale che $\Delta_{\mathbb{V}} = \{a, b, c\}$. Dal nodo centrale attivo, quello puntato dalla struttura ausiliaria `root`, parte un arco per ognuna delle costanti. I tre nodi così raggiunti rappresentano gli elementi costanti della EGS di partenza.

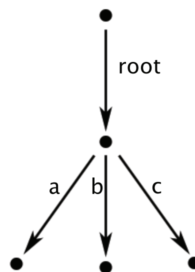

Fig. 4.1: Rappresentazione di costanti

Funzioni

Le funzioni unarie sono gli elementi di $\mathbb{F}$ e $\mathbb{G}$ e per ognuna di esse verrà introdotta un'omonima etichetta contenuta in $\Delta_{\mathbb{F}}$ o $\Delta_{\mathbb{G}}$. La notazione utilizzata

fino a questo momento vede ogni funzione rappresentata per mezzo di archi etichettati con il suo nome. Essi collegano ogni elemento del dominio alla relativa immagine (o a `null`, laddove la funzione non sia definita). Sebbene sia possibile riutilizzare questa stessa modalità applicandola agli elementi di $\Delta_{\mathbb{F}}$ e $\Delta_{\mathbb{G}}$, si vuole proporre un'alternativa che tenga conto dell'esistenza di funzioni nullarie (utilizzate per rappresentare le costanti).
Per ogni funzione viene quindi creato un arco appositamente etichettato, uscente dal nodo centrale effettivo così come spiegato nel caso precedente. Esso porta al nodo rappresentante la funzione stessa, in maniera del tutto analoga al caso delle costanti. Da questo nodo parte un arco etichettato con il nome di ognuno degli elementi appartenenti al dominio, ciascuno diretto verso la relativa immagine. Se $f(a) = b$ allora dal nodo centrale parte un arco etichettato f diretto verso un nodo rappresentante la funzione stessa. L'arco etichettato a uscente da questo stesso nodo conduce all'elemento b, rappresentato con un arco omonimo uscente dal nodo centrale. Ad esempio, la Figura 4.2 rappresenta una Δ-struttura con due etichette di costanti **a** e **b** ed una etichetta di funzione **f**. Gli archi uscenti da **f** indicano che $f(a) = b$ e $f(b) = a$. Si noti che le funzioni delle EGS non hanno arietà superiore ad uno.

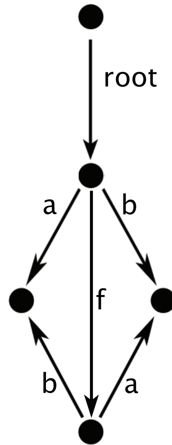

Fig. 4.2: Rappresentazione di funzioni

L'Insieme delle Riserve e nil

La struttura ausiliaria `null` funge da naturale corrispettivo di **nil**. Si noti ad esempio come entrambi gli elementi siano utilizzati, nel corrispettivo modello, per indicare il risultato dell'applicazione di una funzione ad un valore non appartenente al suo dominio.

L'insieme potenzialmente infinito di riserve $\mathcal{R}$ viene utilizzato nelle EGS per introdurre nella Σ-configurazione nuovi elementi, che vengono da esso prelevati. Si noti come nelle SMM la creazione di un nodo coincida con la definizione di un suo identificatore univoco: la stringa (potenzialmente infinita) che porta ad esso, cioè la sequenza di archi che lo collegano al nodo centrale. Si possono dunque immaginare le riserve come nodi non raggiungibili, la cui esistenza è completamente ignorata per definizione: *"Saranno dunque ammesse solo strutture che garantiscono la piena accessibilità ad ognuno dei nodi (cioè tali che $p^*(\Delta^*) = X$)".* Come dimostrato nel resto della dimostrazione e nel risultato di complessità, la scelta di non rappresentare l'esistenza dell'insieme $\mathcal{R}$ non inficia in alcun modo la validità della simulazione.

Relazioni

A questo punto serve capire come rappresentare le relazioni, un concetto più complesso di quelli visti fino a questo momento. Nonostante i numerosi approcci differenti, nessuna delle soluzioni individuate nel corso di questo studio sembra imporsi come più adatta in termini assoluti. Si è quindi deciso di presentare due rappresentazioni alternative, rimandando alle relative sezioni il loro confronto in termini di manipolazione della struttura e complessità.

Il primo metodo, che verrà chiamato **metodo degli indici**, consiste nella creazione una catena di archi tra gli elementi di ogni tupla, etichettandola in maniera univoca per ognuna delle tuple della struttura. Infatti, una relazione $\mathbf{R}$ appartenente a $\mathbb{R}$ può essere descritta nella forma:

$$\mathbf{R}=\{(x_1^1, x_1^2, \ldots, x_1^n), \ldots, (x_k^1, x_k^2, \ldots, x_k^n)\}.$$

I vari elementi x_i^j sono rappresentati da rispettivi nodi all'interno della Δ-struttura. Infatti, per definizione, ogni x_i^j è un elemento appartenente a $\mathcal{V}$ o $\mathcal{D}$ ed è dunque rappresentato da un nodo. Per ogni relazione $\mathbf{R}^i \in \mathbb{R}$ viene calcolata la sua dimensione k, cioè il numero delle sue tuple. Si aggiungono quindi all'insieme $\Delta_{\mathbb{R}}$ le etichette $\mathbf{R}_1^i, \ldots, \mathbf{R}_k^i$. Ogni insieme di elementi in relazione tra loro è rappresentato da un indice, un numero intero $j \leq k$. Se $(f, g, h) \in \mathbf{R}^i$ ed il loro indice è j allora vengono posizionati i seguenti archi:

- `set` $\mathbf{f}\mathbf{R}_j^i$ `to` $\mathbf{g}$
- `set` $\mathbf{g}\mathbf{R}_j^i$ `to` $\mathbf{h}$

Viene quindi creata una catena di archi etichettati R_j^i che unisce gli elementi in relazione, così come mostrato nella Figura 4.3 Dunque, per scoprire se un insieme di elementi appartiene alla relazione R^i, si indaga sull'esistenza di una catena R_j^i per qualche j che unisca tutti gli elementi appartenenti a

questo insieme.
Caso particolare è quello delle relazioni di **R** di arietà pari ad 1, per le quali viene a mancare la necessità di indicizzare le tuple. È sufficiente infatti utilizzare archi, etichettati con lo stesso **R**, uscenti da ogni elemento del grafo in direzione di `null` o di `exist` a seconda che il nodo in questione soddisfi o meno la relazione. Questo arco finale è stato omesso per brevità nel caso di relazioni di arietà maggiore, dal momento che il controllo di esistenza può essere effettuato direttamente sui nodi stessi.

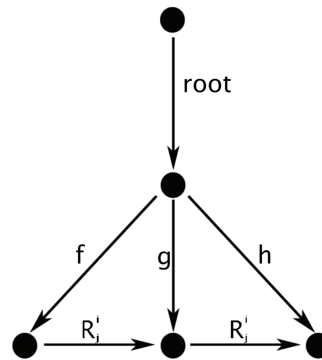

Fig. 4.3: Relazioni: metodo degli indici

Il secondo approccio, chiamato **metodo delle tuple**, consiste invece nel rappresentare nella stessa struttura tutte le possibili tuple appartenenti alle relazioni sotto forma di cammini. Per cominciare, viene creato a partire dal nodo centrale attivo un arco etichettato con il nome di ognuna delle relazioni. Serve quindi aggiungere a $\Delta_{\mathbb{R}}$ una etichetta per ognuna delle relazioni in $\mathbb{R}$. Dal nodo rappresentante ogni relazione vengono inizializzati gli archi etichettati con le etichette di tutte le costanti del grafo (cioè appartenenti a $\Delta_{\mathbb{V}}$ e $\Delta_{\mathbb{D}}$. Si noti che fino a questo punto la definizione coincide con quella data per descrivere la simulazione di funzioni unarie. Per queste, infatti, gli archi uscenti sono diretti verso l'immagine associata all'elemento rappresentato dall'etichetta.

Il comportamento delle relazioni unarie è del tutto analogo: gli archi etichettati con nomi di elementi appartenenti alla relazione conducono a `exist`, che ricopre il ruolo di valore *true*, mentre gli altri archi conducono a `null`, cioè *false*. Per rappresentare una relazione R n-aria l'idea è quella di allungare la catena in modo da farla diventare lunga n, costruendo percorsi uscenti dal nodo **R** diretti verso `exist` o `null`. In questo modo, partendo dal nodo rappresentante una relazione, si potranno seguire cammini etichettati con gli elementi delle sue tuple diretti verso `exist`. Al contrario, cercando di accedere ad una tupla non valida si giungerà a `null`. Si noti che è necessario creare esplicitamente soltanto i cammini associati alle tuple appartenenti

alle relazioni. Infatti, per definizione, ogni cammino non dichiarato conduce automaticamente a `null`. La Figura 4.4 mostra come esempio la parte una di struttura ausiliaria contenente informazioni sulle tuple della relazone binaria $\mathbf{R}$=$\{(a,a),(b,a),(b,b)\}$. Le stringhe **R a a**, **R b a** e **R b b** conducono a `exist`. La tupla (b,a) non appartenente alla relazione **R** è rappresentata per mezzo di archi che conducono a `null`. Si noti come queste informazioni siano memorizzate in una parte di Δ-struttura separata da quella contenente la vera a propria simulazione della Σ-struttura, a cui si accede per mezzo dell'arco `root`.

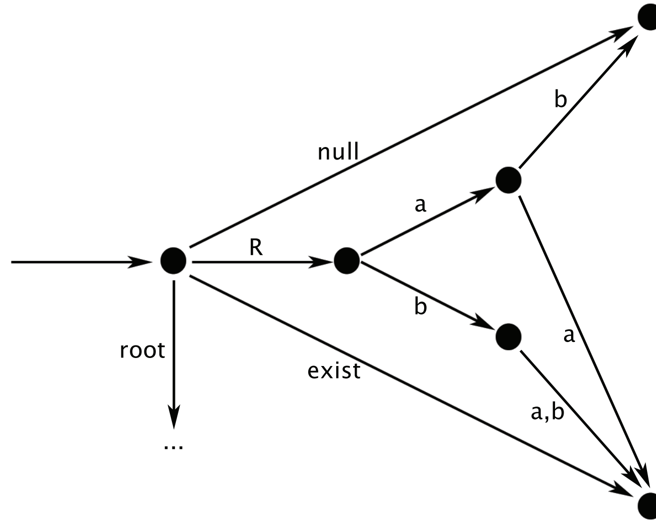

Fig. 4.4: Relazioni: metodo delle tuple

4.2 Espressioni

Si introducono a questo punto l'insieme delle variabili per i vertici $\mathbb{X}$ e l'insieme delle variabili per i dati $\mathbb{Y}$. Essi sono facilmente rappresentabili per mezzo di due nuovi sottoinsiemi di Δ contenenti nomi di funzioni nullarie, $\Delta_{\mathbb{X}}$ e $\Delta_{\mathbb{Y}}$.

Ogni EGS ammette le seguenti espressioni:
$V \in$ VExpr ::= $X \mid \mathbf{nil} \mid \mathbf{v} \mid \mathbf{f}(V)$ con $X \in \mathbb{X}$
$D \in$ DExpr ::= $Y \mid \mathbf{d} \mid \mathbf{g}(V)$ con $Y \in \mathbb{Y}$
$B \in$ BExpr ::= $V = V \mid D = D \mid \neg(B) \mid \mathbf{R}(E_1 \ldots E_n)$ con $\mathbf{R}$:$\tau^n, E_i : \tau$

Le varie espressioni sono facilmente rappresentabili per mezzo dei costrutti introdotti nella sezione precedente. Le variabili X ed Y, così come **v** e **d**, sono elementi puntati direttamente dal centro della struttura tramite archi

etichettati con il loro stesso nome. L'elemento **nil** corrisponde a `null`. Le funzioni **f** e **g** e la relazione **R** corrispondono alle strutture a cui si accede per mezzo degli archi `f` e `g`, così come descritto precedentemente. L'identità $=$ altro non è che la relazione binaria di equivalenza, così come la negazione $\neg$ può essere rappresentata per mezzo di una relazione di arietà 1. Si noti però che negazione ed identità possono anche essere omesse dalla rappresentazione fisica nella struttura e dedotte per mezzo dei costrutti ausiliari (alcuni già descritti, come ad esempio `exist`, altri introdotti nella sezione seguente).

4.2.1 Semantica delle Espressioni

Una Σ-configurazione è una coppia (S, μ) dove S è una Σ-struttura. La funzione μ è chiamata *store* ed è definita come $\mu = \mu_X \cup \mu_Y$ dove $\mu_X : \mathbb{X} \to \mathcal{V} \cup \{\mathbf{nil}\}$ e $\mu_Y : \mathbb{Y} \to \mathcal{D}$. Essa evolve nel corso della computazione, così come S, modificata durante l'esecuzione dei programmi che saranno descritti in seguito. Un apposito elemento μ viene inserito nell'insieme ausiliario Δ_a: esso verrà utilizzato per etichettare gli archi uscenti da una variabile e diretti verso il nodo rappresentante il suo valore. Un analogo elemento `eval` viene aggiunto per indicare il risultato della valutazione di una espressione, ottenuto per mezzo delle regole che seguono. Si ricorda che, data la Σ-configurazione (S, μ), la dicitura $S, \mu \models E \overset{e}{\Rightarrow} a$ indica che l'espressione E valuta all'elemento $a \in S$. Sia l'espressione E che l'elemento a sono rappresentabili nella Δ-struttura, ed è dunque possibile utilizzare gli archi `eval` per associarli.

$$\frac{\mathbf{b} \in \mathbb{V} \cup \mathbb{D}}{S, \mu \models \mathbf{b} \overset{e}{\Rightarrow} \mathbf{b}_S} \quad \frac{Z \in \mathbb{X} \cup \mathbb{Y}}{S, \mu \models Z \overset{e}{\Rightarrow} \mu(Z)} \quad \frac{S, \mu \models E \overset{e}{\Rightarrow} a \quad \mathbf{h} \in \mathbb{F} \cup \mathbb{G}}{S, \mu \models \mathbf{h}(a) \overset{e}{\Rightarrow} \mathbf{h}_S(a)}$$

$$\frac{S, \mu \models E_i \overset{e}{\Rightarrow} a_i \quad \langle a_1, \dots, a_n \rangle \in \mathbf{R}_S}{S, \mu \models \mathbf{R}(E_1, \dots, E_n)} \quad \frac{S, \mu \models E_i \overset{e}{\Rightarrow} a_i \quad \langle a_1, \dots, a_n \rangle \notin \mathbf{R}_S}{S, \mu \models \neg\mathbf{R}(E_1, \dots, E_n)}$$

Fig. 4.5: Regole di semantica per espressioni

Le regole illustrate nella Figura 4.5 permettono di stabilire verso quali nodi rivolgere gli archi etichettati `eval`, azione che può essere compiuta al momento dell'inserimento del nodo così come in quello della sua valutazione. Per stabilire se un nodo rappresenti una variabile od una costante vengono aggiunti a Δ_a i seguenti elementi ausiliari:

- Le costanti $\mathbb{V}$, $\mathbb{D}$, $\mathbb{X}$, $\mathbb{Y}$, $\mathbb{R}$, $\mathbb{F}$, $\mathbb{G}$;
- L'arco `in` che unisce ogni elemento della struttura al suo insieme di appartenenza.

Ad esempio, l'espressione booleana `w in` $= \mathcal{V}$ stabilisce se `w` sia un elemento di $\mathcal{V}$ o meno, ed utilizzato come guardia di un'istruzione `if` permette di

comportarsi di conseguenza. È possibile descrivere ora l'effetto delle regole di valutazione sulla Δ-struttura della SMM simulante. Per direzionare gli archi, ed in particolare l'arco `eval`, basta utilizzare una semplice istruzione `set`.

- Se **b** è una costante appartenente a $\mathbb{V}$ o $\mathbb{D}$, cioè se `b in` $= \mathbb{V}$ oppure `b in` $= \mathbb{D}$, allora `b eval` $=$ `b`, cioè **b** valuta a sé stessa;

- Se Z è una variabile appartenente a $\mathbb{X}$ o $\mathbb{Y}$, cioè se `Z in` $= \mathbb{X}$ oppure `Z in` $= \mathbb{Y}$, allora `Z eval` $=$ `Z` μ;

- Se **h** è una funzione di $\mathbb{F}$ o $\mathbb{G}$, cioè se `h in` $= \mathbb{F}$ oppure `h in` $= \mathbb{G}$, allora se `E eval` $=$ `a` si ha che `E h eval` $=$ `a h`;

- Per sapere se le espressioni $(E_1, \ldots, E_n)$ siano in relazione **R** o meno è necessario valutare le singole espressioni e controllare se l'insieme dei loro valori uno degli elementi indicizzati di **R** stesso. È la stessa semantica a stabilire che le espressioni valutabili sono quelle appartenenti a VExpr e DExpr, ed è dunque possibile utilizzare l'arco `eval` appena introdotto. Si procede in modalità diversa a seconda della strategia di base scelta al momento di rappresentare le relazioni.

  L'idea da adottare nel caso del metodo degli indici è di effettuare la valutazione delle espressioni $(E_1, \ldots, E_n)$ sfruttando l'arco `eval`. La modalità con cui questo arco viene definito ed utilizzato nei vari tipi di nodi è descritta nei punti precedenti. L'unico caso non affrontato è ovvio per definizione: l'arco `eval` uscente da `null` punta a `null` stesso. Per mezzo di istruzioni `if` si analizzano i nodi a cui conduce la valutazione delle n espressioni, cercando una catena di archi etichettati $\mathbf{R}_j$ per qualche indice j. Se questa catena esiste allora si può dedurre che $\mathbf{R}(E_1, \ldots, E_n)$, in caso contrario che $\neg\mathbf{R}(E_1, \ldots, E_n)$. Questa deduzione verrà richiesta al momento di valutare la guardia di istruzioni **if** ed istruzioni **while**.

  Per quanto riguarda invece il metodo delle tuple, il primo passo da compiere è quello di valutare le espressioni $(E_1, \ldots, E_n)$. Una volta noti i loro valori, sarà possibile indicarli come etichette del cammino da percorrere partendo dal nodo della relazione **R**. Questo cammino terminerà in `exist` se la tupla appartiene alla relazione **R**, a `null` altrimenti. Sebbene sia possibile raggiungere il nodo a cui porta la valutazione di una espressione E_i semplicemente seguendo l'arco `eval` associato, non ci fornisce in maniera esplicita il nome della costante associata al valore stesso. In altre parole, se E_i è espressa per mezzo della stringa σ allora il percorso σ `eval` condurrà ad un nodo raggiungibile dal centro percorrendo un singolo arco etichettato con il nome

di una costante. Per stabilire quale sia questa costante non si può fare altro che procedere per tentativi. Per mezzo di istruzioni `if` la stringa σ `eval` verrà quindi confrontata con tutte le etichette appartenenti a $\mathbb{V}$ e $\mathbb{D}$ fino a stabilire quale di esse corrisponde al valore di E_i. Una volta stabiliti i valori delle n espressioni, identificabili con $(\sigma_1, \ldots, \sigma_n)$ sarà sufficiente una singola istruzione `if` per stabilire se la tupla appartenga o meno ad $\mathbf{R}$. Come guardia dell'istruzione verrà utilizzata l'espressione $\mathbf{R}\sigma_1 \ldots \sigma_n =$ `exist`.

Dunque, è possibile realizzare per mezzo di una SMM una rappresentazione semanticamente corretta delle espressioni di una EGS di partenza.

4.3 Programmi

Il passaggio rimanente per poter completare la descrizione della SMM simulante una EGS riguarda il programma che controlla la macchina. Si vuole cioè dimostrare che ogni programma computabile sulla struttura di partenza è simulabile per mezzo delle istruzioni per macchine di Schönhage. Si ricordi la sintassi dei programmi per EGS:

$$\begin{aligned} P \in \text{Prg} ::= X := V \mid V := D \mid \mathbf{f}(X) := V \mid \mathbf{g}(X) := D \mid \mathbf{New}(X) \\ \mid \mathbf{skip} \mid P; P \mid \mathbf{if}(B)\{P\}\{P\} \mid \mathbf{while}(B)\{P\} \end{aligned}$$

Le regole semantiche per i programmi sono illustrate nella figura 4.6.

$$\frac{S, \mu \models E \overset{e}{\Rightarrow} a}{S, \mu \models Z := E \overset{s}{\Longrightarrow} S, \mu[Z \leftarrow a] \models \mathbf{skip}} \qquad \frac{}{S, \mu \models \mathbf{New}(X) \overset{s}{\Longrightarrow} (S, \mu)[\nu X] \models \mathbf{skip}}$$

$$\frac{S, \mu \models X \overset{e}{\Rightarrow} a \quad S, \mu \models V \overset{e}{\Rightarrow} b}{S, \mu \models \mathbf{f}(X) := V \overset{s}{\Longrightarrow} S, \mu[\mathbf{f}(a) := b] \models \mathbf{skip}}$$

$$\frac{S, \mu \models P_1 \overset{s}{\Longrightarrow} S', \mu' \models P_1'}{S, \mu \models P_1; P_2 \overset{s}{\Longrightarrow} S', \mu' \models P_1'; P_2} \qquad \frac{S, \mu \models P_1 \overset{s}{\Longrightarrow} S', \mu' \models \mathbf{skip}}{S, \mu \models P_1; P_2 \overset{s}{\Longrightarrow} S', \mu' \models P_2}$$

$$\frac{S, \mu \models B}{S, \mu \models \mathbf{if}(B)\{P_0\}\{P_1\} \overset{s}{\Longrightarrow} S, \mu \models P_0} \qquad \frac{S, \mu \models \neg B}{S, \mu \models \mathbf{if}(B)\{P_0\}\{P_1\} \overset{s}{\Longrightarrow} S, \mu \models P_1}$$

$$\frac{S, \mu \models B}{S, \mu \models \mathbf{while}(B)\{P\} \overset{s}{\Longrightarrow} S, \mu \models P; \mathbf{while}(B)\{P\}} \qquad \frac{S, \mu \models B}{S, \mu \models \mathbf{while}(B)\{P\} \overset{s}{\Longrightarrow} S, \mu \models \mathbf{skip}}$$

Fig. 4.6: Regole di semantica per i programmi

Si ricordi che $\mathcal{S}, \mu \models P \overset{s}{\Rightarrow} \mathcal{S}', \mu' \models P'$ indica che, partendo dalla Σ-configurazione (S, μ), l'esecuzione del programma P riduce alla valutazione del programma P' nella Σ-configurazione (S', μ'). Si vuole ora dimostrare che è possibile simulare il programma di una EGS per mezzo di istruzioni per SMM semanticamente equivalenti. Si procede per casi:

- $X := V$ Questa istruzione causa una modifica dello store μ ed è dunque facilmente simulabile per mezzo di una istruzione `set`. In particolare, all'istruzione $Z := E$ corrisponde `set` Z μ `to` E `eval`;

- $Y := D$ Analogo al caso precedente;

- $\mathbf{f}(X){:=}V$ Questa istruzione fa sì che dal vertice a cui valuta X parta un arco **f** diretto verso il vertice a cui valuta V. Dunque, anche in questo caso, si tratta dello spostamento di un arco. Esso è ottenibile per mezzo dell'istruzione `set` X `eval` **f** `to` V `eval`. Si noti che X `eval` **f** $= X\ \mu$ **f**, e che dunque entrambe le stringe portano adesso ad un nuovo nodo. Lo store non subisce modifiche;

- $\mathbf{g}(X){:=}D$ Analogo al caso precedente;

- **New**(X) Causa lo spostamento di un elemento da $\mathcal{R}$ a $\mathcal{V}$ ed aggiorna lo store in modo che X valuti al nuovo elemento. L'istruzione corrispondente è `New` X. Serve però definire anche le strutture ausiliarie uscenti da X, in particolare precisando l'istruzione `set` X `in to` $\mathbb{X}$;

- **skip** Indica la fine di esecuzione di un programma ed è necessaria per far funzionare la sequenzialità come definita nel punto successivo. Solitamente può essere omessa, o descritta come un `goto` verso l'istruzione successiva, ma può anche essere simulata per mezzo di un `halt` quando si tratta dell'ultima istruzione del programma;

- $P; P$ È l'istruzione che indica sequenzialità. $P_1; P_2$ è il programma che esegue il P_1 e, una volta terminato, esegue P_2 sulla Σ-configurazione da esso raggiunta. La sequenzialità sulle SMM è implicita e può comunque essere forzata per mezzo di un'istruzione `goto`. Inoltre si ricordi che l'output di una SMM, che descrive la Δ-struttura al termine dell'esecuzione, può essere visto come l'input di un'altra macchina con un altro programma;

- **if**$(B)\{P_0\}\{P_1\}$ Mentre i due programmi P_0 e P_1 corrispondono intuitivamente alla loro stessa simulazione, la valutazione della guardia non è un procedimento immediato. Per stabilire infatti quale dei due programmi eseguire (scelta che sarà portata a termine per mezzo di una istruzione `goto`) è necessario capire se l'espressione B sia accettata o meno dalla Σ-configurazione (S, μ). Come già accennato, l'identità

= può essere rappresentata per mezzo di una relazione di equivalenza. Pertanto, B sarà o un'espressione di tipo $\mathbf{R}(E_1, \ldots, E_n)$ o di tipo $\neg\mathbf{R}(E_1, \ldots, E_n)$. Come visto nella sezione precedente, indipendentemente dalla rappresentazione scelta è possibile dedurre una di queste due espressioni per mezzo di una sequenza di istruzioni `if` che controllano se le n espressioni compongano o meno una delle tuple della relazione $\mathbf{R}$. Se per almeno una di queste istruzioni la guardia viene soddisfatta allora il controllo del programma passa alla prima delle istruzioni che simulano P_0. In caso contrario, il controllo passerà alla prima delle istruzioni che simulano P_1;

- **while**$(B)\{P_0\}$ Questo caso è simile a quello precedente, con l'unica aggiunta di istruzioni `goto` da eseguire una volta valutata l'espressione B. Un ciclo **while** si può infatti definire per mezzo di istruzioni `if` e salti.

Una configurazione iniziale è una configurazione in cui $\mu(X)$=**nil** per ogni variabile X. Serve dunque garantire che prima dell'esecuzione del programma simulante tutti gli archi etichettati μ portino a `null`. Ciò è facilmente ottenibile per mezzo di istruzioni `set`.

4.4 Run time: Confronto tra Strutture

Come già visto nel capitolo precedente, programma P per EGS viene eseguito in tempo t su input $(\mathcal{S}, \mu)$, scritto $Time_P(\mathcal{S}, \mu) = t$, quando $\mathcal{S}, \mu \models P(\stackrel{s}{\Rightarrow})^t \mathcal{T}, \xi \models \mathbf{skip}$ per qualche $(\mathcal{T}, \xi)$. La dimensione $\mid \mathcal{S}, \mu \mid$ di una configrazione $(\mathcal{S}, \mu)$ è il numero n di elementi dell'universo dei vertici V. Un programma P viene eseguito in tempo polinomiale se esiste $k > 0$ tale che $Time_P(\mathcal{S}, \mu) \leq k \cdot \mid \mathcal{S}, \mu \mid^k$ per tutte le configurazioni $(\mathcal{S}, \mu)$.

Le caratteristiche principali analizzate per stabilire se un programma venga eseguito in tempo polinomiale sono due: il numero di elementi vertici, cioè $\mid V \mid$, e il numero di istruzioni eseguite. Si vuole capire come queste due entità si comportino una volta rappresentate all'interno di una SMM, per capire se la simulazione di un programma eseguito in tempo polinomiale sia anch'essa eseguibile in tempo polinomiale. Sebbene sia ovvio che il numero di istruzioni della macchina simulante deve avere uno stretto legame polinomiale con il numero di istruzioni del programma di partenza, ci si chiede cosa considerare come corrispettivo dell'insieme dei vertici. Intuitivamente, il corrispettivo naturale di V potrebbe essere l'unione di $\Delta_{\mathbb{V}}$ e $\Delta_{\mathbb{X}}$. Includendo anche i dati, poiché rappresentati come nodi, bisognerebbe aggiungere a questi insiemi anche $\Delta_{\mathbb{D}}$ e $\Delta_{\mathbb{Y}}$. Tenendo conto della presenza di strutture ausiliarie, si potrebbe voler espandere l'insieme scelto fino a considerare tutto Δ. Volendo però considerare la dimensione della Δ-struttura vera e

propria, bisognerebbe contare il numero di nodi in essa inizializzati. Anche in questo caso, però, bisognerebbe decidere se considerare solamente i nodi appartenenti alla struttura che rappresenta esplicitamente elementi presenti nella EGS simulata, oppure includere nel conto anche le strutture ausiliarie. Queste alternative sono elencate in ordine crescente di dimensione, cioè in maniera tale che pongano limiti sempre meno restrittivi. Si comincia dunque esaminando l'ultima opzione, cercando di capire quali scelte possano portare al miglior risultato di simulazione.

Definizione 4.1 (Esecuzione in tempo polinomiale)**.** Un programma P_Δ per SMM viene eseguito in tempo t sulla sua Δ-struttura S_Δ, scritto $Time_{P_\Delta}(S_\Delta) = t$, quando il numero di istruzioni di P_Δ computate prima dell'esecuzione dell'istruzione `halt` (essa compresa) è uguale a t. La dimensione di una Δ-struttura $\mid S_\Delta \mid$ è uguale al numero di nodi in essa contenuti. Un programma P_Δ viene eseguito in tempo polinomiale se esiste $k > 0$ tale che $Time_{P_\Delta}(S_\Delta) \leq k \cdot \mid S_\Delta \mid^k$ per ogni configurazione di S_Δ nel corso della computazione.

Si vuole quindi ora cercare di capire se la computazione del programma simulante una EGS da parte di una SMM è polinomiale, cioè se soddisfa la definizione appena fornita. Più formalmente:

Teorema 4.4.1 (Polinomialità della simulazione)**.** *Sia P_Σ un programma per EGS eseguito in tempo polinomiale. Allora anche il programma P_Δ per SMM che lo simula viene eseguito in tempo polinomiale.*

La dimostrazione di questo teorema si divide in due fasi. Nella prima parte, si dimostrerà per induzione strutturale che la dimensione del programma simulante le istruzioni di partenza rispetta il vincolo polinomiale dato. Nella sezione seguente, si dimostrerà che anche considerando il costo della costruzione della Δ-struttura in termini di numero di istruzioni il teorema di complessità continua a valere. Si comincia analizzando per casi le istruzioni da simulare:

- $X := V$ È sufficiente una singola istruzione `Set`, necessaria per la modifica dello store;
- $Y := D$ Analogo al caso precedente;
- $\mathbf{f}(X){:=}V$ Anche in questo caso si tratta dello spostamento di un singolo arco. Dunque, è nuovamente sufficiente una singola istruzione `Set`;
- $\mathbf{g}(X){:=}D$ Analogo al caso precedente;
- **New**(X) Oltre all'omonima istruzione `New` X è necessario definire le strutture ausiliarie uscenti da X e ridirezionare eventuali archi non utilizzati verso `null`. Per farlo, occorre al più una istruzione `Set` per ognuna delle etichette disponibili, pari a $\mid \Delta \mid$;

- **skip** Come precedentemente discusso, la simulazione di questa istruzione può essere effettuata in diversi modi: affidandosi al passaggio di controllo sequenziale delle SMM o per mezzo di una istruzione `goto` o `halt`, a seconda della posizione all'interno del programma. Serve dunque al massimo un passo di computazione;

- $P_1; P_2$ Anche in questo caso si tratta di esprimere sequenzialità, e dunque al più di una singola istruzione. Ad essa si vanno a sommare le istruzioni per simulare P_1 e P_2;

Fino a questo punto l'istruzione più onerosa è la creazione di un nuovo, che richiede un numero di istruzioni pari a $\mid \Delta \mid +1$. Diventa a questo punto necessario affrontare la problematica legata alla simulazione delle relazioni, il concetto più problematico di questa simulazione. Per cominciare, si vuole dare una spiegazione del perché non esista per esse una rappresentazione intuitiva che non pecchi a livello di complessità di valutazione o di costo nella costruzione delle strutture ausiliarie.

Uno principali punti di forza del modello di Schönhage è la modalità di accesso ai dati, che avviene per mezzo di puntatori. Si supponga ad esempio che una Δ-struttura rappresenti, come precedentemente descritto, la funzione $f(a) = b$. Partendo dal centro della struttura, è possibile raggiungere un particolare nodo sia seguendo l'arco etichettato `b` sia seguendo la sequenza di archi etichettati `f` ed `a`. Ciò significa che, volendo manipolare il nodo in questione, si può accedere ad esso direttamente per mezzo della stringa che rappresenta $f(a)$, senza che sia necessario calcolare il suo valore (cioè b). Se si considera il valore di $f(a)$ come il particolare nodo a cui la stringa `f a` porta, allora è sufficiente seguirla per accedere ad esso per poter manipolare espressioni che contengono questo valore. È quindi possibile definire $g(f(a)) = c$, e cioè $g(b) = c$, senza che sia necessario sapere che $f(a) = b$.

Il problema sorge quando si vuole definire il valore di un'espressione per mezzo di costanti (cioè quando si vuole ottenere la stringa `b` partendo dalla stringa `f a`). Come spiegato nel primo capitolo, l'unico modo per stabilire se due sequenze di archi portano allo stesso nodo è visitarle. Bisogna dunque procedere per casi, controllando se l'equivalenza tra `f a` ed ognuna delle costanti appartenenti a Δ sia soddisfatta o meno. Ciò è esattamente quello che avviene rappresentando le relazioni per mezzo del metodo delle tuple: prima di poter controllare se una tupla appartenga o meno alla relazione è necessario indagare il suo valore confrontandolo con tutte le costanti appartenenti alla struttura. Una volta stabilito questo valore, lo si può utilizzare per effettuare il controllo sulla struttura ausiliaria che rappresenta la relazione per mezzo di archi etichettati con i nomi di costanti. Il metodo degli indici non prevede questo tipo di valutazione, poiché lavora direttamente sul nodo

rappresentate il valore dell'espressione e non necessita di sapere quale esso sia. Si vedrà adesso come questi due metodi, all'apparenza estremamente diversi, siano strettamente interconnessi.

- **if**$(B)\{P_0\}\{P_1\}$ Alle istruzioni per la simulazione di P_0 e P_1 si aggiungono quelle necessarie per valutazione dell'espressione B. Queste dipendono dal metodo di simulazione scelto per rappresentare B, che come precedentemente detto è una relazione.

  - **Metodo degli indici**. Questo metodo è estremamente conveniente dal punto di vista strutturale, ma è anche molto costoso a livello di programma. Quello che ci si chiede è: *"Esiste una catena di archi etichettati* $\boldsymbol{R}_j$ *collega gli elementi della tupla, per qualche indice* j*?"*. L'unico modo per saperlo è procedere per casi, partendo dal primo elemento della tupla. Si controlla per mezzo di istruzioni `if` l'esistenza di R_i per $1 \leq i \leq d$, dove d è il numero di tuple appartenenti a $\mathbf{R}$. Ogni volta che un arco $\mathbf{R}_i$ viene trovato il controllo si sposta sul nodo a cui esso porta, cercando di procedere con la catena per un numero di volte pari all'arietà k della relazione. Nel caso peggiore, la relazione conterrà un numero di tuple pari a $\mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid^k$. Il numero di istruzioni massime necessarie per capire se un insieme di elementi appartenga o meno a $\mathbf{R}$ è dunque pari a $k\cdot \mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid^k$;
  - **Metodo delle tuple**. Si lavora nuovamente sulla stessa relazione $\mathbf{R}$ di arietà k. Come precedentemente accennato, alla base di questo metodo sta la necessità di associare ad ogni espressione dell'insieme fornito in B il nome della costante a cui essa valuta. Per ognuna delle k espressioni è necessario eseguire al più un numero di istruzioni `if` pari a $\mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid$, cioè il numero di costanti della struttura. Una volta compiuta questa operazione, conoscendo il valore delle costanti è sufficiente una singola istruzione per stabilire se esse siano o meno in relazione R tra loro. Il costo totale è quindi $k\cdot \mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid +1$.

  Sebbene il metodo delle tuple sembri estremamente conveniente se confrontato a quello degli indici, la situazione cambierà una volta fatte alcune considerazioni.

- **while**$(B)\{P_0\}$ Caso analogo al precedente.

I casi più onerosi in termini di istruzioni simulanti sono dunque **if** e **while**, cioè quelli che richiedono la valutazione di una tupla rispetto ad una relazione. Con il metodo delle tuple, il costo di questa valutazione equivale al più a $k\cdot \mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid +1$, dove k è l'arietà della relazione più grande appartenente a $\mathbb{R}$. Dunque, se $Time_P(\mathcal{S}, \mu) = t$ allora $Time_{P_\Delta}(S_\Delta)$ sarà al più uguale a

$t{\cdot}k{\cdot} \mid \Delta_{\mathbb{V}}{\cup}\Delta_{\mathbb{D}} \mid +1$. Ma allora esiste d tale che $t{\cdot}k{\cdot} \mid \Delta_{\mathbb{V}}{\cup}\Delta_{\mathbb{D}} \mid +1 \leq d{\cdot} \mid S_\Delta \mid^d$ e dunque P_Δ viene eseguito in tempo polinomiale.

Volendo utilizzare il metodo degli indici, $Time_{P_\Delta}(S_\Delta)$ sarà al più uguale a $t \cdot k{\cdot} \mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid^k$. Anche in questo caso esiste D tale che $Time_{P_\Delta}(S_\Delta) \leq D{\cdot} \mid S_\Delta \mid^D$, sebbene il valore di D sia di molto maggiore a quello di d.

4.4.1 Costruzione della Struttura

Bisogna però tenere in considerazione il fatto che per lavorare su una Δ-struttura è necessario prima costruirla, servendosi di altre istruzioni. Per poter realizzare una SMM simulante, è dunque necessario costruire una serie di nodi. L'aggiunta di ogni nodo richiede un numero di istruzioni pari a $\mid \Delta \mid +1$, cioè un'istruzione `New` più un'istruzione `Set` per ognuno degli archi uscenti dal nodo. Serve capire quanti nodi vengono creati in quella che si potrebbe chiamare **fase di inizializzazione**. Si procede per casi

- *Strutture ausiliarie.* I nodi ausiliari di una SMM sono presenti in quantità fissa e finita, che può variare in base alle necessità di calcolo. Fino a questo punto, sono stati presentati al più una dozzina di nodi ausiliari, che sono più che sufficienti per svolgere le operazioni di costruzione e simulazione;

- *Nodi di costanti.* La Δ-struttura contiene un nodo per ognuna delle costanti in $\mathbb{V}$ e $\mathbb{D}$, per un totale di nodi pari a $n_v =\mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid$;

- *Nodi di funzioni.* Caso analogo al precedente. La Δ-struttura contiene un nodo per ognuna delle funzioni in $\mathbb{F}$ e $\mathbb{G}$, per un totale di nodi pari a $\mid \Delta_{\mathbb{F}} \cup \Delta_{\mathbb{G}} \mid$;

- *Relazioni.* Ancora una volta, si distingue tra i due casi. Utilizzando il metodo degli indici, non è necessario aggiungere alcun nodo. Infatti, le relazioni vengono rappresentate unicamente per mezzo di archi tra nodi di costanti e le strutture ausiliarie `null` e `exist`. Utilizzando invece il metodo delle tuple, la situazione cambia drasticamente. Per ogni relazione $\mathbf{R}$ di arietà k appartenente a $\mathbb{R}$ serve costruire una struttura ausiliaria composta da:

  - Un nodo rappresentante la relazione $\mathbf{R}$;
  - Per rappresentare le tuple è necessario un numero di nodi ausiliari pari a $\sum_{i=0}^{k-1} n_v^i$.

  Per brevità, si approssima la somma di questi nodi a n_v^k. Dunque, per ogni elemento $\mathbf{R}_i$ di arietà k_i appartenente a $\mathbb{R}$ è necessario costruire

un numero di nodi pari a n^{k_i} dove $n_v = \mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid$. La somma dei nodi per rappresentare tutte le funzioni avrà come limite superiore n_v^w per qualche w.

Ciò che salta all'occhio osservando questi valori sono i risultati all'apparenza opposti dei due metodi di rappresentazione delle funzioni. Il metodo degli indici, che vedeva una crescita esponenziale di istruzioni richieste per la valutazione di relazioni, non richiede l'aggiunta di nodi in fase di inizializzazione. Al contrario il metodo delle tuple, dal costo costante per quanto riguardava la valutazione, assume un peso esponenziale durante la costruzione della struttura. Questo potrebbe indurre a pensare che, volendo considerare sia il costo di costruzione che quello di simulazione del programma per EGS, i due metodi finiscano per avere complessità molto simile. In realtà, è bene sottolineare che la scelta del metodo delle tuple incide significativamente sul valore di $\mid \Delta \mid$. Per ogni tupla di ogni relazione $\mathbf{R}$ appartenente a $\mathbb{R}$ è infatti necessario creare un'etichetta apposita. Ognuna di queste etichette è associata ad un arco che dovrà essere inizializzato per tutti i nodi appartenenti alla struttura, aggiungendo al costo della costruzione un quantitativo di istruzioni pari al numero di nodi della struttura moltiplicato per il numero totale di tuple u.

Ricapitolando, utilizzando il metodo delle tuple, il costo in istruzioni della fase di inizializzazione è pari a $\mid \Delta \mid \cdot (\mid \Delta \mid + (n_v)^w)$ per qualche w, dove $n_v = \mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid$. Dunque $Time_{P_\Delta}(S_\Delta)$ sarà al più uguale a $t \cdot k \cdot n_v + 1 + \mid \Delta \mid \cdot (\mid \Delta \mid + n_v^w)$. Per $n = \mid \Delta \mid$ di ha che

$$Time_{P_\Delta}(S_\Delta) = t \cdot k \cdot n_v + 1 + n^2 + n \cdot (n_v)^w$$

Approssimando, $n^2 + n \cdot (n_v)^w \leq c \cdot n^{w+1}$ per qualche c e $t \cdot k \cdot n_v \leq t \cdot k \cdot n$. Pertanto,

$$Time_{P_\Delta}(S_\Delta) \leq t \cdot k \cdot n + c \cdot n^{w+1}$$

dove il valore di w non dipende da S_Δ ma solamente dall'alfabeto Δ. Ciò significa che, anche valutando il costo della costruzione della struttura, l'esecuzione in tempo polinomiale è assicurata.

Si noti che la descrizione delle EGS e dell'esecuzione dei loro programmi non sembra tenere in considerazione la costruzione della Σ-struttura.

Capitolo 5

Tiering e SMM

Nel capitolo precedente si è dimostrato che per ogni EGS il cui programma P viene eseguito in tempo polinomiale esiste una macchina di Schönhage in grado di simularla, tale che la simulazione preserva la polinomialità dell'esecuzione. L'enunciato principale dimostrato nell'articolo di Leivant e Marion riguarda il rapporto tra funzioni calcolabili in tempo polinomiale e programmi che possiedono una certa proprietà di stratificazione, ricordata nella sezione 3.2.

Teorema 5.0.2 (Caratterizzazione)**.** *Una funzione su una struttura grafo è computabile in tempo polinomiale se e solo se è computabile da un programma terminante e strettamente ramificabile.*

Applicare questo teorema alla simulazione ottenuta è estremamente intuitivo alla luce del risultato ottenuto nel Capitolo 4. Si considerino separatamente entrambi i versi d'implicazione del teorema originale.

Teorema 5.0.3 (Correttezza)**.** *Sia P_Σ un programma per EGS strettamente ramificabile. Esso computa una funzione su strutture grafo in tempo polinomiale. Allora il suo programma simulante P_Δ per SMM computa la stessa funzione, anch'esso in tempo polinomiale.*

Dimostrazione. Per il Teorema di caratterizzazione, P_Σ è calcolabile tempo polinomiale. Ma anche la sua simulazione P_Δ ha costo al più polinomiale. Dunque P_Δ computa una funzione su strutture grafo in tempo polinomiale. □

Teorema 5.0.4 (Completezza estensionale)**.** *Sia $\boldsymbol{f}$ una funzione polinomiale su strutture grafo. Allora esiste un programma P_Δ per SMM terminante e strettamente ramificabile che calcola questa funzione.*

L'idea dietro la dimostrazione di questo teorema è che P_Δ sia la simulazione del programma terminante e strettamente ramificabile P_Σ, la cui esistenza è garantita dal teorema di caratterizzazione originale. Tuttavia non è

ancora stata definita una nozione di ramificazione per SMM: questo sarà lo scopo della prima sezione di questo capitolo, che introdurrà tale nozione per le SMM e mostrerà che la simulazione preserva le proprietà di tiering.

La situazione ottimale sarebbe quella di poter definire la nozione di tiering direttamente per generiche SMM, mostrando poi che una SMM ramificabile ha tempo d'esecuzione polinomiale. Purtroppo questo non sembra possibile: le SMM sono un modello troppo astratto (o forse bisognerebbe dire rudimentale) per poter definire un tiering significativo. L'unico costrutto delle SMM sono le etichette, che non consentono di distinguere tra valori, funzioni e relazioni a meno che non si introducano particolari restrizioni, che non fanno però parte del modello di SMM. È per questo motivo che verranno qui considerate solo un sottoinsieme di macchine di Schönhage: come vedremo nell'ultima sezione, questo sottoinsieme corrisponde alle SMM simulanti una qualche EGS astraendo dai dettagli implementativi.

5.1 Simulazione e Ramificazione

Si vuole descrivere come adattare il concetto di tiering alle SMM che simulano una EGS, mostrando come la simulazione descritta nel capitolo precedente preservi il tier dei programmi. Poiché Δ viene già utilizzato per indicare l'alfabeto della struttura grafo, i $\mathbb{T}$-environment verranno indicati con la coppia (Γ, Ω). Nuovamente, si concentrerà lo studio sul reticolo booleano $\mathbb{T} = \{T = \{\mathbf{0},\mathbf{1}\}, \leq, \mathbf{0}, \wedge, \vee\}$, senza perdita di generalità. Le lettere greche α e β fungono da parametri per i tier $\{\mathbf{0},\mathbf{1}\}$. Nelle EGS le regole di tiering permettono di assegnare un tier ad espressioni e programmi. Nella SMM simulante, questi elementi corrispondono a nodi ed archi del grafo ed a programmi composti da istruzioni per macchine di Schönhage.

5.1.1 Tiering di Espressioni

Dato un $\mathbb{T}$-environment (Γ, Ω) si definisce $\mathbb{T}_\Delta$-environment della macchina simulante la coppia $(\Gamma_\Delta, \Omega_\Delta)$. La funzione Γ_Δ assegna ad ogni etichetta $X \in \Delta_\mathbb{X}$ il tier dato da $\Gamma(X)$. La funzione Ω_Δ ad ogni etichetta $\mathbf{f} \in \mathbb{F}$ le espressioni nella forma $\alpha \to \beta$ date da $\Omega(\mathbf{f})$. Esattamente come per le EGS, la funzione $\mathbf{f}$:

- È **stabile** se tutte le espressioni in $\Omega_\Delta(\mathbf{f})$ sono nella forma $\alpha \to \alpha$;
- **Riduce** nell'environment se tutte le espressioni in $\Omega_\Delta(\mathbf{f})$ sono nella forma $\alpha \to \beta$ con $\beta < \alpha$;

Una asserzione di tiering è una espressione nella forma $\Gamma_\Delta, \Omega_\Delta \vdash V_\Delta : \alpha$ dove $(\Gamma_\Delta, \Omega_\Delta)$ è un $\mathbb{T}$-environment. V_Δ è l'insieme di nodi della struttura ed espressioni necessarie per esprimere la simulazione dell'espressione V per

EGS. Questo insieme dipende dalla forma di V: si descrivono ora caso per caso i suoi elementi e le modalità per inferire il suo tier. Alla base dell'assegnazione di tier stanno le regole per EGS descritte nella Figura 5.1.

$$\frac{}{\Gamma, \Omega \vdash \mathbf{c} : \alpha} \quad \frac{\Gamma(X) = \alpha}{\Gamma, \Omega \vdash X : \alpha} \quad \frac{\alpha \to \beta \in \Omega(\mathbf{f}) \quad \Gamma, \Omega \vdash V : \alpha}{\Gamma, \Omega \vdash \mathbf{f}(V) : \beta}$$

$$\frac{\Gamma, \Omega \vdash V_i : \alpha}{\Gamma, \Omega \vdash \Gamma, \Omega \vdash \mathbf{R}(V_1, \ldots, V_n) : \alpha} \quad \frac{\Gamma, \Omega \vdash V_i : \alpha}{\Gamma, \Omega \vdash V_0 = V_1 : \alpha}$$

Fig. 5.1: Regole di tiering per espressioni

Si noti che il tiering non coinvolge i dati, la cui esistenza all'interno della struttura viene temporaneamente ignorata. Dove l'aspetto dell'oggetto nella Δ-struttura può dipendere dal metodo scelto per la simulazione si adotta la notazione $[\![V]\!]$ per indicare la simulazione di V. Si procede descrivendo come assegnare il tier nelle strutture simulanti delle SMM, distinguendo i casi in base alla istruzione simulata.

- $\Gamma, \Omega \vdash \mathbf{c} : \alpha$. Ogni costante è espressa tramite un nodo raggiungibile dal centro attivo della struttura tramite un arco etichettato con il suo nome. Gli elementi costanti puntano per mezzo dell'arco `in` a $\mathbb{V}$. È sempre possibile assegnare un tier ai nodi rappresentanti una costante;

- $\Gamma, \Omega \vdash X : \alpha$. Le variabili sono simulate in maniera analoga alle costanti. L'arco etichettato `in` uscente dai loro nodi punta a $\mathbb{X}$. Il tier del nodo che rappresenta la variabile X è uguale a $\Omega_\Delta(X)$;

- $\Gamma, \Omega \vdash \mathbf{f}(V) : \beta$. Il nodo che rappresenta una funzione $\mathbf{f}$ è connesso tramite l'arco `in` a $\mathbb{F}$ o $\mathbb{G}$. Se $\alpha \to \beta \in \Omega_\Delta(\mathbf{f})$ e $\Gamma_\Delta, \Omega_\Delta \vdash V_\Delta : \alpha$ allora V $\mathbf{f}$ ha tier β;

- $\Gamma, \Omega \vdash \mathbf{R}(V_1, \ldots, V_n) : \alpha$. In questo caso $\mathbf{R}$ `in` $= \mathbb{R}$. Sia $[\![V_i]\!]$ la simulazione di V_i, che dipende dal metodo utilizzato per rappresentare le relazioni. Allora l'espressione $\mathbf{R}\ [\![V_1]\!] \ldots, [\![V_n]\!]$ ha tier α se e solo se $\Gamma_\Delta, \Omega_\Delta \vdash V_i : \alpha$ per ogni i;

- $\Gamma, \Omega \vdash V_0 = V_1 : \alpha$. Se $\Gamma_\Delta, \Omega_\Delta \vdash V_i : \alpha$ per $i = \{0, 1\}$ allora anche $[\![V_0]\!] = [\![V_1]\!]$ ha tier α.

5.1.2 Tiering di Programmi

Le regole per inferire il tier di un programma per EGS sono descritte su singole istruzioni. Come si evince dal capitolo precedente, ognuna di queste istruzioni viene simulata per mezzo di un insieme più vasto di istruzioni per

SMM. Si può rendere più chiaro questo concetto immaginando di assegnare tutte le istruzioni simulanti ad una serie di etichette che differiscano solo per un indice finale. Ad esempio, per simulare un'istruzione per EGS **New** tramite il programma di una SMM tale che $\Delta = n$ sono necessarie $n + 1$ istruzioni: una istruzione `New` e n istruzioni per stabilire dove far puntare gli archi del nodo. L'insieme di queste istruzioni verrà chiamato **blocco simulante**. Ognuna di esse verrà identificata con una etichetta indicizzata tale che $\mathcal{L}\text{New}_i$ per $1 \leq i \leq (n+1)$. Le regole di ramificazione che assegnano un tier a **New** indicano che tier assegnare all'intero blocco etichettato $\mathcal{L}\text{New}_i$. Dunque, se P_Σ è un programma per EGS e $[\![P_\Sigma]\!] = P_\Delta$ è il programma che lo simula allora se $\Gamma, \Omega \vdash P_\Sigma : \alpha$ allora $\Gamma_\Delta, \Omega_\Delta \vdash P_\Delta : \alpha$.

5.1.3 Proprietà

Si vogliono ora adattare alle SMM che simulano le EGS alcune definizioni inerenti al tiering. Dato un programma P_Δ, una etichetta $\mathbf{f}$ è detta:

- **Ispezionata** se compare nel blocco simulante una istruzione di tipo $X =: V$ o all'interno di una guardia nel blocco simulante una istruzione **if** o una istruzione **while**;

- **Modificata** se durante l'esecuzione di P_Δ viene spostato l'arco etichettato con $\mathbf{f}\ X$ uscente dal nodo che rappresenta una qualsiasi variabile X.

Dato un $\mathbb{T}$-environment $(\Gamma_\Delta, \Omega_\Delta)$, il blocco simulante un'istruzione di tipo **while**$(B)\{P\}$ di tier α è detto:

- **Stazionario** se nessun arco uscente dai nodi rappresentanti le funzioni $\mathbf{f}$ di tier $\alpha \to \alpha$ viene modificato;

- **Strettamente modificante** se vengono modificati archi uscenti dai nodi rappresentanti funzioni $\mathbf{f}$ di tier $\alpha \to \alpha$ ma solo una di esse è ispezionata. Questo significa che al massimo un nodo rappresentante una funzione viene raggiunto sia per 'leggere' archi da esso uscenti sia per modificarli.

Si noti come tutte queste proprietà, una volta riformulate per le SMM, vengano preservate durante la simulazione. Infatti, l'accesso per lettura o modifica dei vari elementi da parte del programma simulante avviene se e solo se esistente nel programma di origine. Inoltre, si dice che Γ_Δ è un *tiering iniziale* se $\Gamma_\Delta(X) = \mathbf{1}$ per ogni etichetta di variabile in input. Anche in questo caso, Γ_Δ è tiering iniziale se e solo se lo è Γ. Dato un reticolo $\mathbb{T}$ e $\Gamma_\Delta, \Omega_\Delta \vdash P_\Delta : \alpha$ si dice che $(\Gamma_\Delta, \Omega_\Delta)$ è una *ramificazione stretta* di P_Δ se Γ_Δ è un tiering iniziale e ogni blocco simulante un'istruzione **while** è stazionario o strettamente modificante. Si dice che P_Δ è **strettamente**

ramificabile se ha una $\mathbb{T}$-ramificazione stretta $(\Gamma_\Delta, \Omega_\Delta)$ con Γ_Δ iniziale per qualche $\mathbb{T}$ non banale.

L'ultimo elemento mancante riguarda la terminazione. Se P_Σ è un programma per EGS terminante allora anche la sua simulazione P_Δ termina. Infatti, la terminazione espressa per mezzo dell'istruzione **skip** è simulata dall'istruzione `halt` che causa l'arresto della macchina. Allo stesso tempo non sussistono elementi che possono portare ad un mancato arresto della macchina simulante in presenza dell'istruzione `halt`. Eventuali salti e loop infiniti indicherebbero la presenza di analoghi elementi nel programma simulato: il numero di cicli di ogni loop ed il valore di verità della guardia ad ogni iterazione vengono preservati. A questo punto è possibile dimostrare il teorema di completezza estensionale.

Dimostrazione. Sia P_Σ programma terminante e strettamente ramificabile che calcola **f**. Allora esiste una sua simulazione P_Δ per SMM, a sua volta terminante. Inoltre, poiché P_Σ è strettamente ramificabile esiste $\mathbb{T}$-ramificazione stretta (Γ, Ω) con Γ iniziale per qualche $\mathbb{T}$ non banale. Sia $\Gamma_\Delta, \Omega_\Delta$ la relativa $\mathbb{T}$-ramificazione di P_Δ. Poiché Γ è tiering iniziale anche Γ_Δ lo è, e poiché ogni loop in P_Σ è stazionario o strettamente modificante allora anche il blocco simulante ognuno di essi possiede le stesse proprietà. Ma allora anche P_Δ è strettamente ramificabile. □

5.2 Teorema di Caratterizzazione per SMM

Come accennato nell'introduzione a questo capitolo, sarebbe interessante individuare un sottoinsieme di macchine di Schönhage che abbiano una struttura ed una descrizione formale intuitivamente comprensibili da un osservatore umano e su cui sia possibile dimostrare il teorema di caratterizzazione senza dover fare riferimento esplicito al passaggio di simulazione. In altre parole, si vuole definire un insieme di macchine di Schönhage SMM_Σ che soddisfi il seguente teorema.

Teorema 5.2.1 (Teorema di caratterizzazione per SMM_Σ)**.** *Una funzione su una struttura grafo è computabile in tempo polinomiale se e solo se è computabile da un programma per SMM_Σ terminante e strettamente ramificabile.*

Sebbene esistano anche macchine non appartenenti a SMM_Σ che soddisfino questo teorema, ci si concentrerà su di questo per poter effettuare una dimostrazione del teorema di caratterizzazione che proceda in parallelo con quella per EGS. L'idea è infatti quella di identificare l'insieme delle cosiddette *macchine simulanti*, cioè tali da poter essere considerate la simulazione di una evolving graph structure nelle modalità viste nel capitolo precedente. Cioè, fissata una tecnica di di simulazione, SMM_Σ è l'insieme di macchine di

Schönhage M_Δ tali che per ognuna di esse esiste una EGS M_Σ da essa simulata (cioè $M_\Delta = [\![M_\Sigma]\!]$). Si descrivono ora le caratteristiche di queste macchine, considerando il metodo di simulazione visto nel capitolo precedente.

Δ-strutture di SMM$_\Sigma$

Sia M_Δ un elemento di SMM$_\Sigma$. Allora la sua Δ-struttura presenta i seguenti componenti:

- **Strutture ausiliarie**. L'alfabeto Δ della macchina contiene tutte le etichette ausiliarie descritte nei Capitoli 1 e 4, il cui uso corrisponde a quello già visto. Ad esempio, sono presenti gli elementi `exist` e `null`, così come μ, `in` e `eval`;

- **Costanti**. Etichette particolari, appartenenti a $\Delta_\mathbb{V}$ e $\Delta_\mathbb{V}$. Vengono utilizzate dal centro attivo per identificare nodi che rappresentano costanti. Questi sono uniti per mezzo dell'arco `in` ai nodi ausiliari $\mathbb{V}$ e $\mathbb{D}$;

- **Funzioni**. Etichette appartenenti a $\Delta_\mathbb{F}$ e $\Delta_\mathbb{G}$. Vengono utilizzate dal centro attivo per identificare nodi che rappresentano funzioni. Questi sono uniti per mezzo dell'arco `in` ai nodi ausiliari $\mathbb{F}$ e $\mathbb{G}$. Gli archi uscenti dai nodi funzioni etichettati con il nome di una costante portano ad una costante oppure a `null`;

- **Variabili**. Descritte come le costanti, ma connesse per mezzo di `in` a $\mathbb{X}$ e $\mathbb{Y}$. Un arco **f** uscente dal nodo di una variabile conduce al valore della funzione **f** applicata alla variabile che rappresenta;

- **Relazioni**. A seconda del metodo di simulazione scelto, la Δ-struttura può contenere speciali etichette che uniscono le tuple in relazione tra loro oppure una struttura ausiliaria composta da cammini/tuple che conducono a `exist` o `null`.

Per tutti questi elementi, gli archi ausiliari vengono gestiti esattamente come descritto nell'introduzione alle SMM e nella descrizione della simulazione. Data la Δ-struttura di M_Δ è possibile ricondursi alla Σ-configurazione (S, μ) di una certa M_Σ compiendo il percorso inverso della simulazione, e aggiungendo agli elementi descritti un insieme di nodi di riserva.

Programmi di SMM$_\Sigma$

Poiché si desidera che ogni M_Δ corrisponda ad una qualche M_Σ è necessario che anche il suo programma sia espresso in una forma riproducibile su una EGS. Per questa ragione, si considerano validi solo i programmi realizzati come sequenze di blocchi simulanti. Per ognuno dei nove tipi di istruzione accettati da M_Σ, la simulazione introdotta nel Capitolo 4 ha descritto un

insieme di istruzioni corrispondenti per M_Δ. Ciò significa che esistono diversi tipi di blocchi simulanti, a cui è possibile associare una istruzione per EGS che abbia lo stesso effetto:

- **Modifica di una variabile**. Questo blocco simulante è composto da una singola istruzione con la forma `set` X `eval` **f** `to` E `eval`. I due elementi X ed E devono appartenere allo stesso insieme di variabili/valori vertici o dati. Corrisponde ad un'istruzione per EGS di forma $X := E$;
- **Modifica di una funzione**. Anche questo blocco simulante è composto da una singola istruzione `set` che sposta un arco uscente da un nodo funzione. Corrisponde ad un'istruzione per EGS di forma $\mathbf{f}(X) := V$ (anche in questo caso per vertici o dati);
- **Creazione di un nodo**. Se la dimensione dell'alfabeto $\mid \Delta \mid$ è n allora questo blocco simulante è composto da una istruzione `New` e n istruzioni `set` che importano gli archi uscenti dal nuovo nodo. Corrisponde ad una istruzione per EGS **New**;
- **Arresto**. Espresso per mezzo della sola istruzione `halt`, che corrisponde ad un'istruzione **skip** al termine di un programma per EGS;
- **Sequenzialità**. Il passaggio tra un blocco simulante e un altro può essere espresso per mezzo di una istruzione `goto`, o semplicemente omesso grazie al meccanismo di controllo sequenziale delle SMM. Corrisponde alla istruzione sequenziale per EGS P**;**P';
- **Blocco if**. Insieme di istruzioni `if goto` necessarie per simulare il controllo della guardia in una istruzione **if** per EGS;
- **Blocco while**. Caso simile al precedente, ma in cui l'insieme di istruzioni `if` e `goto` permette di simulare una istruzione **while** per EGS.

Un programma per SMM scritto servendosi unicamente di questi blocchi simulanti corrisponde ad un programma per EGS composto da una opportuna istruzione per ogni blocco. Si noti che una eventuale parte di programma dedicata alla costruzione della Δ-struttura può essere ignorata perché non rilevante ai fini di questa dimostrazione, se realizzata in maniera consona alla descrizione della struttura data.

5.2.1 Ramificazione Stretta Implica Polinomialità

Ora che è stata data una definizione delle macchine incluse in SMM_Σ è possibile dimostrare il teorema di caratterizzazione definito per questo insieme. Si procede dimostrando separatamente i due versi dell'implicazione. Per

cominciare, si vuole dimostrare che ogni programma su SMM_Σ strettamente ramificabile computa una funzione su Δ-strutture in tempo polinomiale. La dimostrazione procederà parallelamente a quella delle EGS. Infatti, per definizione, ogni macchina M_Δ appartenente a questo insieme può essere vista come simulazione di una evolving graph structure M_Σ. Il tier di M_Δ preserva il tier di M_Σ.

Per prima cosa, si dimostrerà che se tutte le variabili della Δ-struttura hanno lo stesso tier allora anche nel corso dell'esecuzione questa proprietà rimarrà preservata. Se P_Δ ha tier **0** allora non ha cicli ed è quindi eseguito in un numero di passi $\leq\mid P\mid$. Infatti, nei programmi per SMM_Σ, tutte le istruzioni `goto` portano ad etichette posizionate in un punto successivo del programma, fatta eccezione per i blocchi simulanti le istruzioni **while** che hanno però tier maggiore di **1**. Inoltre, il valore di una variabile di tier **1** dipende solo dai vertici di tier **1**. Dunque il numero di iterazioni di un dato loop è limitato dal numero di possibili valori modificati durante l'esecuzione del suo corpo. Poiché per ipotesi P_Δ è strettamente ramificabile questo numero è polinomiale.

Non-Interferenza

Lemma 1 (Restrizione)
Sia $(\Gamma_\Delta, \Omega_\Delta)$ *un* $\mathbb{T}$*-environment. Se* $\Gamma_\Delta, \Omega_\Delta \vdash P_\Delta$*:* **0** *allora* $\Gamma_\Delta(X) = \mathbf{0}$ *per ogni etichetta che rappresenta una variabile* X *a cui è stato assegnato un valore in* P*.*

Dimostrazione. P_Δ può essere visto come la simulazione di un programma P_Σ per EGS. Ma allora, poiché la simulazione preserva il tiering, vale che $\Gamma, \Omega \vdash P_\Sigma$: **0**, dove Γ e Ω sono ricostruibili invertendo il processo descritto nella prima sezione di questo capitolo. Per lo stesso lemma di restrizione, per ogni variabile X in P_Σ vale che $\Gamma(X) = \mathbf{0}$. Ma allora, per definizione di Γ_Δ, anche $\Gamma_\Delta(X) : \mathbf{0}$. □

Si dice che Γ_Δ è **compatibile** con una Δ-struttura avente etichetta μ utilizzata come descritto precedentemente, se per ogni X, X' tali che X `in` $=$ $\mathbb{X}$ e X' `in` $= \mathbb{X}$ vale che se $\Gamma(X) \neq \Gamma(X')$ allora anche $X\ \mu \neq X'\ \mu$.

Lemma 2 (Compatibilità)
Sia $\Gamma_\Delta, \Omega_\Delta \vdash P_\Delta : \alpha$ *e sia la* Δ*-struttura di* P_Δ *compatibile con* Γ*. Allora la* Δ*-struttura rimane compatibile con* Γ *anche nel corso dell'esecuzione di* P_Δ*.*

Dimostrazione. L'esecuzione di P_Δ segue passo a passo l'esecuzione del programma per EGS P_Σ che simula. La proprietà di compatibilità dipende dagli archi μ e dal valore di Γ_Δ, che rispecchiano il valore di μ e di Γ della

macchina simulata. Dunque la compatibilità è preservata dalla simulazione e rimane valida nel corso dell'esecuzione di P_Σ. Ad ogni istruzione eseguita da P_Σ corrisponde l'esecuzione di un blocco simulante in P_Δ. In ogni stato raggiunto dopo una istruzione in P_Σ vale la compatibilità, e dunque lo stesso si può dire della Δ-struttura raggiunta al termine dell'esecuzione di un blocco simulante. □

Si vuole ora dimostrare che se il tiering è compatibile con la configurazione inziale allora i valori di tier inferiore a quello del programma non vengono modificati nel corso della sua computazione. Anche in questo caso sfruttiamo l'esistenza di una macchina M_Σ che viene simulata da M_Δ con le modalità stabilite. In [11] la dimostrazione di questa proprietà per EGS avviene definiendo il (Γ, Ω)-collapse di una Σ-configurazione. Se (S, μ) è la configurazione di M_Σ, allora il suo (Γ, Ω)-collapse è una configurazione (S_Γ, μ_Ω) tale che S_Γ si ottiene da S eliminando le funzioni $\mathbf{f}$ tali che $(\mathbf{1} \rightarrow \mathbf{1}) \notin \Delta(\mathbf{f})$ e $\mu_\Omega(X) = \mu(X)$ se $\Gamma(X) = \mathbf{1}$, altrimenti è indefinito. Si vuole dimostrare che applicando le rispettive restrizioni a M_Δ il risultato è la SMM che simula (S_Γ, μ_Ω). Si procede per casi:

- L'eliminazione di una funzione $\mathbf{f}$ dalla Σ-struttura (S, μ) è rappresentabile per mezzo della rimozione dell'omonima etichetta dall'alfabeto Δ di M_Δ. Di conseguenza, dalla Δ-struttura viene rimosso il nodo che rappresenta la funzione, così come tutti gli archi etichettati $\mathbf{f}$. La rimozione del nodo comporta anche l'eliminazione dei suoi archi. Il risultato è una Δ-struttura del tutto analoga a quella di partenza, fatta eccezione per l'assenza degli archi etichettati f e dei nodi che non sono raggiungibili per mezzo di un cammino che non includa archi di questo genere (ad esempio il nodo che rappresenta la funzione stessa). Questa Δ-struttura è l'esatto corrispondente di S_Γ nella sua simulazione come SMM_Σ;

- Si potrebbe simulare l'introduzione di μ_Ω tramite l'aggiunta di un'omonima etichetta ausiliaria. Essa verrebbe utilizzata dai nodi della Δ-struttura in maniera identica a μ, fatta eccezione per i nodi X per i quali $\Gamma(X) \neq \mathbf{1}$. L'arco μ_Ω uscente da essi condurrebbe a `null`. In realtà, è possibile e più semplice reindirizzare gli archi di μ stesso verso `null` quando richiesto. La Δ-struttura così modificata è esattamente la struttura della SMM simulante (S_Γ, μ_Ω).

Seguendo queste istruzioni è possibile ottenere il Δ-collapse di una Δ-struttura. Come già detto, data M_Δ esiste una EGS M_Σ da essa simulata. Ma per definizione di Δ-collapse, il (Γ, Ω)-collapse di M_Σ è simulato dal Δ-collapse di M_Δ. Dunque, poiché il tier è preservato dalla simulazione, anche per M_Δ l'esecuzione di un programma di un dato tier non influisce sui valori di tier inferiore.

Si vuole ora dimostrare che i vertici di tier **1** nell'output di un programma non dipendono da vertici di tier **0** né da archi che non hanno tier $\mathbf{1} \to \mathbf{1}$.

Lemma 3 (Collapsing)
Se un programma P_Δ viene eseguito su una Δ-struttura e sul suo $(\Gamma_\Delta, \Omega_\Delta)$-collapse allora le Δ-strutture risultanti dalle due computazioni sono identiche modulo $(\Gamma_\Delta, \Omega_\Delta)$-collapse.

Dimostrazione. Lo stesso teorema è dimostrato, con le opportune diciture, su EGS. Come già visto, esiste un programma P_Σ che è simulato dal programma P_Δ e per cui vale il lemma di collapsing. Ciò significa che esistono una Σ-struttura (S, μ) ed il suo Γ, Ω-collapse (S_Γ, μ_Ω) da cui si ottiene la stessa Σ-struttura modulo Γ, Ω-collapse eseguendo P_Σ. Ma per definizione, la Δ-struttura su cui viene eseguito P_Δ è il risultato della simulazione di (S, μ) e la sua modifica per $(\Gamma_\Delta, \Omega_\Delta)$-collapse rispecchia (S_Γ, μ_Ω). Il risultato dell'esecuzione di P_Δ su di esse è la stessa Δ-struttura modulo $(\Gamma_\Delta, \Omega_\Delta)$-collapse, e dunque il lemma è valido. □

Limiti Polinomiali

Grazie alle dimostrazioni viste fino a questo punto è adesso possibile dimostrare che la restrizione a programmi strettamente ramificabili garantisce la polinomialità della sua esecuzione. Il lemma originale, presentato per le EGS, è il seguente:

Lemma 3 (Soundness)
$\Gamma, \Omega \vdash P : \alpha$ *con P strettamente modificante. Esiste $k > 0$ tale che per ogni struttura grafo $\mathcal{S}$ ed ogni store μ compatibile con Γ, se $\mathcal{S}, \mu \models P \overset{s}{\Rightarrow} \mathcal{S}'', \mu''$ $\models P'$ allora $\mathcal{S}, \mu \models P\ (\overset{s}{\Rightarrow})^t\ \mathcal{S}'', \mu'' \models P'$ per qualche $t < k + \mid \mathcal{S} \mid^k$.*

Si vuole adattare lo stesso lemma alle SMM simulanti. Per l'esattezza, si vuole dimostrare che se $\Gamma_\Delta, \Omega_\Delta \vdash P_\Delta : \alpha$ con P_Δ strettamente modificante allora esiste $k > 0$ tale che per ogni Δ-struttura compatibile con Γ_Δ se P_Δ riduce a P'_Δ in t passi allora $t < k + \mid \Delta \mid^k$.

Dimostrazione. Si procede per induzione strutturale su P_Δ. Infatti, il numero massimo di passi di esecuzione di un programma è dato dalla somma dei passi per l'esecuzione dei suoi sottoblocchi simulanti. Per il Lemma di compatibilità, P_Δ rimane compatibile con Γ_Δ per tutta la durata dell'esecuzione. Se P_Δ ha tier **0** allora il numero di passi della sua esecuzione sarà $< \mid P_\Delta \mid$ come precedentemente dimostrato. Se P_Δ ha invece tier **1** si procede con l'analisi dei singoli tipi di sottoblocchi simulanti. Quelli potenzialmente più costosi in termine di numero di passi per l'esecuzione sono quelli che simulano istruzioni **while** con guardia booleana **B**. La guardia contiene un

numero n di variabili, che come B sono di tier **1**. Poiché P_Δ è strettamente modificante, per ogni blocco di istruzioni P simulante un'istruzione **while**$(B)Q$ si possono distinguere due casi:

- Il blocco simulante Q modifica solo archi di funzioni che non sono ispezionate in P e soprattutto non nella guardia B. Come visto precedentemente, i vertici di tier **1** in P dipendono solo dalle funzioni i cui archi hanno tier $\mathbf{1} \to \mathbf{1}$, quindi essi rimangono invariati durante tutte le iterazioni del blocco simulante il loop. Dunque il valore di verità di B dipende solo dalle combinazioni delle variabili $X_1, \ldots, X_m$ che contiene. Poiché l'esecuzione termina, esse sono tutte differenti e sono al più x^m dove x è il numero di variabili di tier **1**. Utilizzando il metodo delle tuple, il costo per la valutazione di B è pari a $k \cdot v + 1$, dove $v = \mid \Delta_{\mathbb{V}} \cup \Delta_{\mathbb{D}} \mid$ e k è l'arietà della relazione in B. Il numero massimo di passi necessari è dunque $(k \cdot v + 1) \cdot x^m$ che è inferiore a $k \cdot \mid \Delta \mid^{m+1}$. Per ipotesi induttiva il blocco simulante Q termina in tempo polinomiale, e dunque lo stesso vale per l'intero blocco P;

- Il blocco simulante Q modifica archi che sono ispezionati in P, e sono tutti uscenti dal nodo che rappresenta una singola funzione **f**. Sia C l'insieme di variabili presenti in P, incluse $X_1, \ldots, X_m$. U l'insieme di valori raggiungibili da queste variabili tramite archi di tier **1** etichettati con nomi di funzioni. Poiché P ha tier **1** allora durante la sua esecuzione tutti i vertici aventi medesimo tier sono contenuti in U. Inoltre U è generato da C la cui dimensione è fissa e determinata dalla sintassi di P. Come dimostrato anche per le EGS, il numero di configurazioni tra variabili e vertici è dunque un valore polinomiale che è limitato dal numero di vertici della dimensione di Δ. Esattamente come nel caso precedente, queste possibili configurazioni vanno moltiplicate per il costo della valutazione di B che è limitato da $k \cdot \mid \Delta \mid$.

□

5.2.2 Polinomialità Implica Ramificazione Stretta

In questa seconda fase si vuole dimostrare che ogni funzione polinomiale su una funzione grafo è computabile da un programma per SMM_Σ terminante e strettamente ramificabile. Come descritto nel capitolo dedicato alle EGS, Leivant e Marion dimostrano in [11] come una Macchina di Turing che opera in tempo $k \cdot n^k$ sia simulabile per mezzo di una EGS M_Σ il cui programma è strettamente ramificabile. Come dimostrato, esiste la simulazione $M_\Delta = [\![M_\Sigma]\!]$ che termina ed è strettamente ramificabie. Ma per definizione $M_\Delta \in \text{SMM}_\Sigma$, poiché la sua Δ-struttura ed il programma da essa simulata soddisfano tutti i vincoli richiesti.

In conclusione, una funzione polinomiale su una struttura grafo è computabile in tempo polinomiale se e solo se è computabile per mezzo di un programma per SMM_{Σ} terminante e strettamente ramificabile.

Bibliografia

[01] Arnold Schönhage (1970), Universelle Turing Speicherung, Automatentheorie und Formale Sprachen, Dörr, Hotz, eds. Bibliogr. Institut, Mannheim, pp. 69-383

[02] Arnold Schönhage (1980), Storage Modification Machines, Society for Industrial and Applied Mathematics, SIAM J. Comput. Vol. 9, No. 3, August 1980

[03] Domiziana Suprani (2010), Macchine di Schönhage e riduzione su grafi, Dipartimento di Scienze dellInformazione, Università di Bologna, domiziana.suprani@studio.unibo.it

[04] Arnold Schönhage (1973), Real-time simulation of multidimensional Turing machines by storage modification machines, Technical Memorandum 37, M.I.T. Project MAC, Cambridge, MA

[05] Yuri Gurevich (1988), On Kolmogorov Machines and Related Issues, the column on Logic in Computer Science, Bulletin of European Association for Theoretical Computer Science, Number 35, 71-82

[06] Yuri Gurevich (1988), Algorithms in the World of Bounded Resources, The Universal Turing Machine - a Half-Century Story, ed. R. Herken, Oxford University Press

[07] Amir Ben-Amram (1995), What is a Pointer machine?, SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory), volume 26, also: DIKU, Department of Computer Science, University of Copenhagen, amirben@diku.dk

[08] Scott Dexter, Patrick Doyle and Yuri Gurevich (1997), Gurevich Abstract State Machines and Schöonhage Storage Modification Machines, Journal of Universal Computer Science, vol. 3, no. 4 (1997), 279303

[09] D. Plump, Term Graph Rewriting, Universitt Bremen, Fachbereich Mathematik und Informatik, det@informatik.uni-bremen.de

[10] Ugo Dal Lago and Simone Martini, Derivational Complexity is an Invariant Cost Model, Dipartimento di Scienze dell'Informazione, Università di Bologna, dallago,martini@cs.unibo.it

[11] Daniel Leivant and Jean-Yves Marion (2013), Evolving Graph-Structures and Their Implicit Computational Complexity, ICALP 2013, Part II, LNCS 7966, pp.349-360, Springer-Verlag Berling Heidelberg, 2013

[12] Yuri Gurevich (1991), Evolving Algebras: An Attempt to Discover Semantics, Bull. EATCS 43, pp. 264-284; a slightly revised version in Current Trends in Theoretical Computer Science, Eds. G. Rozenberg and A. Salomaa, World Scientific, 1993, pp. 266-292

[13] Yuri Gurevich (1995), Evolving Algebras 1993: A Lipari guide, Specification and Validation Methods, Ed. E. Boerger, Oxford University Press, pp. 9-36

[14] PROLOG. Part 1, General Core, Committee Draft 1.0, ISO/IEC JTCI SC22 WG17 No. 92, 1992

[15] Georg Gottlob, Gerti Kappel and Michael Schrefl, Semantics of Object Oriented Data Models - The Evolving Algebra Approach, Next Generation Information Technology, Springer LNCS 504, pp. 144-160

[16] D.E. Knuth, The Art Of Computer Programming, vol. 2, Seminumerical Algorithms, Addison-Wesley, Reading, MA, 2nd ed. 1971, Chapter 4.3.3 p. 275

[17] Volpano, Irvine, Smith (1996), A sound type system for secure flow anlysis. Journal of Computer Security 4(2/3), pp. 176-188

[18] R.E. Tarjan (1977), Reference machines require non-linear time to maintain disjoint sets, STOC 1977, pp.18-29, ACM

[19] Daniel Leivant (1994), Predicative recurrence and computational complexity I: Word recurrence and poly-time, Feasible Mathematics II, Birkhauser-Boston

[20] S. Bellantoni, S.A Cook (1992), A new recursion-theoretic characterization of the poly-time functions, Computational Complexity 2, pp. 97-110

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Martini, relatore di questa mia tesi. In pochi, dopo aver sperimentato una prima volta le mie bozze confuse e le mie innumerevoli email ansiose, mi avrebbero nuovamente accettata come tesista. A lui vanno i miei ringraziamenti anche per avermi dato esempio di un approccio coinvolto ma rigoroso alle scienze informatiche, insegnandomi ad impegnarmi con più metodo e passione.

Ringrazio i miei genitori, che mi hanno sempre sostenuta e spronata, rispettando le mie scelte e sopportando le conseguenze di quelle sbagliate con la pazienza che solo una madre ed un padre possono avere. A loro va il mio affetto e la gratitudine per avermi dato gli strumenti per costruire la persona che sono.

Grazie a mia sorella Caterina per essere sempre stata il mio punto di riferimento e primo esempio da seguire, in ogni luogo e istante. Ogni parte di questa tesi che è stata scritta mentre mi sostituivi in qualche faccenda è anche un po' tua.

Un ringraziamento anche ai miei colleghi dell'Università di Bologna e della École Normale Supérieure de Lyon, compagni di lezioni accademiche e insegnanti di lezioni di vita. Grazie in particolare a Matteo, Balta e Michele per aver condiviso con me quei piccoli eventi che segnano la fine di un percorso e l'inizio di uno nuovo. Grazie di cuore anche a Marco, che nominato mio senpai il primo giorno di università non è mai venuto a meno al suo dovere.

Grazie a tutti gli amici, colleghi e parenti che di questa tesi hanno visto solo le lunghe assenze improvvise e gli effetti collaterali sul mio umore. Citarvi solo per nome non renderebbe giustizia al supporto che mi avete dato. Posso solo impegnarmi a ringraziarvi nei prossimi mesi ed anni facendo frutto delle esperienze vissute insieme.

Infine grazie ad Amedeo, per avermi insegnato quello che non avrei mai potuto imparare sui libri e che non riesco a descrivere a parole. Confido che tu possa capire cosa provo, come hai sempre fatto. Qualsiasi lezione seguirà questo momento la condividerò con te.