Alma Mater Studiorum · Università di Bologna

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

Macchine di Schönhage e riduzione su grafi

Tesi di Laurea in Paradigmi di Programmazione

Relatore:
Chiar.mo Prof.
Simone Martini

Presentata da:
Domiziana Suprani

Sessione II
Anno Accademico 2009/2010

"A man provided with paper, pencil
and rubber, and subject to strict discipline
is, in effect, a universal machine."
- *ALAN TURING*

Indice

Introduzione

Questa tesi nasce con la volontà di mettere a confronto due modelli di calcolo che presentano forti analogie e interessanti punti di contatto: la classe di macchine a puntatori (con particolare attenzione nei confronti del modello introdotto da Schönhage) e il sistema di riscrittura su term graph. È possibile suddividere il lavoro svolto per la realizzazione di questo progetto essenzialmente in due fasi distinte. In un primo momento l'attenzione è stata focalizzata sull'individuazione dei principali modelli di macchine a puntatori.
Come verrà spiegato in seguito, la letteratura dedicata a questo argomento è frammentaria e scarsamente approfondita. Un'ulteriore difficoltà incontrata è stata l'impossibilità di accedere ad alcune delle prime pubblicazioni contenenti le definizioni complete di certi modelli: esse infatti, oltre a non essere disponibili in forma digitalizzata o su pubblicazioni internazionali, non sono mai state tradotte in inglese dalla lingua madre degli autori.
Nonostante ciò, è stato possibile ottenere una visione di queste macchine sufficientemente completa da permettere di scegliere un particolare caso che è stato considerato il più idoneo per far fronte alle specifiche necessità della fase successiva di lavoro: si tratta delle *macchine a modificazione della memoria*, o macchine di Schönhage. Dopo aver rielaborato gli elementi a disposizione, prestando particolare attenzione ai precedenti risultati di simulazione e teoria della complessità computazionale che hanno visto coinvolto questo modello, si è passati alla fase successiva.

La seconda parte del lavoro ha concentrato i propri sforzi nel tentativo di raggiungere il principale obiettivo: dimostrare che è possibile utilizzare una macchina a puntatori per memorizzare la struttura di un term graph e automatizzare per mezzo del programma interno della macchina stessa l'applicazione delle regole di un sistema di riscrittura su term graph. È stato necessario definire, per mezzo dei costrutti propri della macchina scelta, una serie di strutture ausiliarie che permettessero di adattare il sistema a disposizione alle esigenze di simulazione. Una volta definita e descritta una modalità per memorizzare e manipolare term graph per mezzo delle macchine a modificazione della memoria è stato possibile astrarre dai dettagli implementativi per occuparsi della dimostrazone dell'ipotesi iniziale.

Nel capitolo relativo verrà mostrato che, scelta una particolare regola di riscrittura, esiste una macchina di Schönhage capace di applicare questa regola ad ogni term graph che venga fornito in input o che si trovi già memorizzato nella struttura interna.

La suddivisione dei capitoli di questa tesi ricalca parzialmente lo schema sequenziale adottato nei corso del lavoro: il Capitolo 1 tratta le macchine di Schönhage, fornendo la definizione di questo modello e mostrando alcuni casi particolarmente interessanti di simulazione che lo coinvolgono. Nel Capitolo 2 vengono descritte in breve altre due macchine a puntatori non utilizzate direttamente nella parte sperimentale della tesi ma che hanno comunque contribuito a chiarire alcuni dubbi riguardanti questa categoria di automi. Il Capitolo 3 introduce il concetto di riscrittura su term graph, partendo dalle definizioni di term graph e di sistema di riduzione. Il Capitolo 4 mostra infine la parte sperimentale del lavoro svolto, fornendo la definizione delle strutture ausiliarie implementate e dimostrando l'ipotesi formulata inizialmente. Nei capitoli in appendice sono consultabili le definizioni delle macchine coinvolte nelle dimostrazioni di simulazione ed equivalenza svolte nelle prime sezioni.

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 [11] come modello generale di computazione. Dieci anni dopo, lo stesso autore dedicò un articolo intero allo studio più completo di queste macchine [05] 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 [12]. I seguenti paragrafi illustreranno la struttura delle macchine a modificazione di memoria e i principali risultati ottenuti in merito a questo modello.

1.1 Strutture dati e istruzioni

Una macchina a modificazione di 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 assumono il valore di direzione e sono i corrispondenti delle istruzioni di spostamento delle testine delle 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 dai nodi della struttura e gli elementi dell'alfabeto Δ, dal momento che ogni elemento viene usato una e una sola volta da ogni nodo per

etichettare il relativo arco; per questa ragione il numero di archi uscenti da ogni nodo è fissato per ogni particolare macchina ed è uguale a $\sharp\Delta$. Inoltre è presente in ogni grafo un particolare nodo scelto, detto *centro* della struttura. Esso permette di raggiungere ogni altro nodo della stessa struttura e, poiché rappresenta il punto di accesso per il mondo esterno, è analogo alle testine delle macchine di Turing. A differenza delle celle di queste, 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 da X in X indicizzato dagli elementi di Δ. In particolare $p_\alpha(x) = (y)$ indica che il puntatore con etichetta α che parte dal nodo x va in y. Si indichi con Δ^* l'insieme delle parole sull'alfabeto Δ e con $\Box$ la parola vuota. Per ogni Δ-struttura S si definisce la mappatura p^* in maniera ricorsiva:

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

Basandosi sulla definizione appena data, risulta immediato associare parole di Δ^* a elementi di 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 $U, V, W \in \Delta^*$.

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 tramite 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, l'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 è 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. Il linguaggio utilizzato per realizzare il controllo delle SMM prevede 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 il carattere in 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:

  * $W = \square$; $a' = y$ e $p'_\delta(y) = a$ per ogni $\delta \in \Delta$. Si tratta del caso base. Se la parola da creare è vuota il nuovo nodo corrisponde al nodo centrale e pertanto tutti i suoi puntatori portano al nodo centrale stesso;
  * $W = U\alpha$ con $U \in \Delta^*$ e $\alpha \in \Delta$; $a' = a$, $p'_\alpha(p^*(U)) = y$ e $p'_\delta(y) = p^*(W)$ per ogni $\delta \in \Delta$. 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:

  * $W = \square$; $a' = p^*(V)$, cioè a diventa il nodo indicato da V, 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.

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

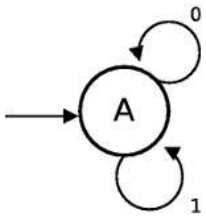


Fig. 1.1: Condizione iniziale

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
```

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).
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.

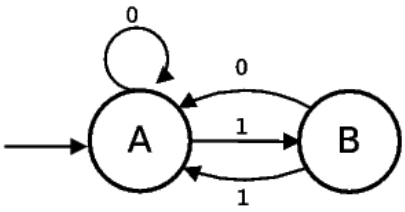


Fig. 1.2: Creazione di un nodo

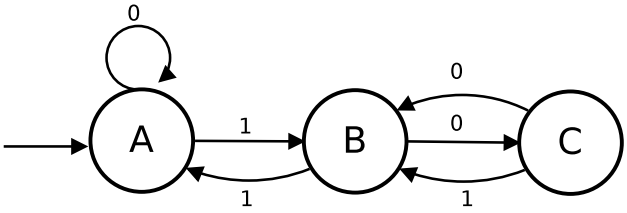


Fig. 1.3: Creazione di un nodo

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.

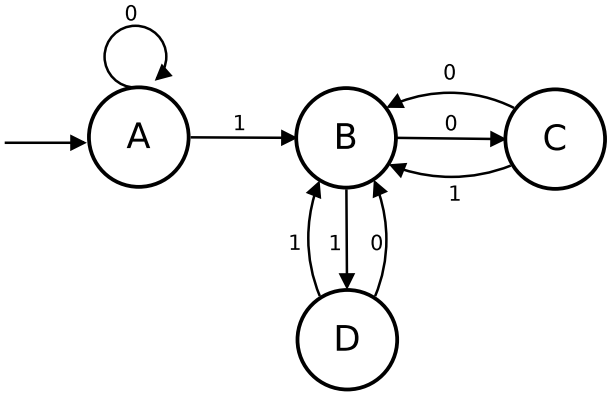


Fig. 1.4: Creazione di un nodo

Si può definire il tempo di esecuzione di un programma come il numero di istruzioni eseguite includendo l'halt. Lo stesso Shönhage 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.

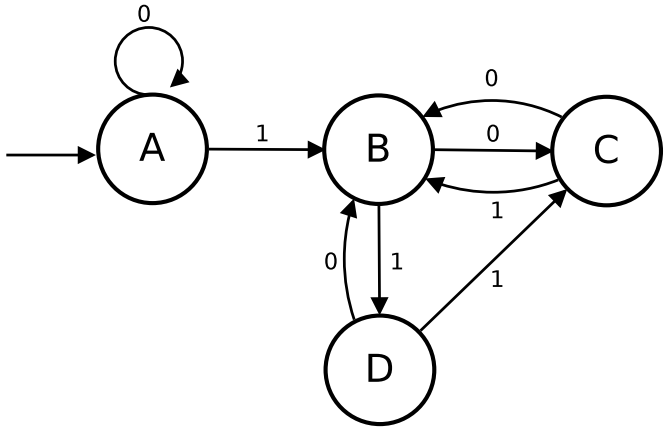


Fig. 1.5: Spostamento di un puntatore

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" [01]. 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 fornita dall'autore una definizione di flessibilità, Yuri Gurevich suppose che si riferisse alla capacità delle SMM di simulare in tempo reale ogni macchina sequenziale [13]. 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 [05] 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 sequenza 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) $\mathcal{M} \overset{\tau}{\leftrightarrow} \mathcal{M}'$ significa che che $\mathcal{M} \overset{\tau}{\rightarrow} \mathcal{M}'$ e $\mathcal{M}' \overset{\tau}{\rightarrow} \mathcal{M}$.

Date queste definizioni è possibile formulare la principale tesi di Schönhage nella seguente maniera: $\mathcal{M} \overset{\tau}{\rightarrow}$ SMM è valido per ogni modello di macchine atomistiche $\mathcal{M}$. Amir M. Ben Amram fornì una chiara descrizione di cosa Schönhage indendesse per macchina atomistica [02]: 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.

Tutte le macchine a puntatori, categoria che verrà trattata e descritta in capitoli successivi, fanno parte dell'insieme delle macchine atomistiche.

Un ulteriore interessante risultato per quanto riguarda la complessità di computazioni eseguite dal modello di macchine a modificazione di memoria è espresso dal seguente teorema formulato da Schönhage [05]:

Teorema 1.2.1. *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.*

Bisogna sottolineare che nonostante certi modelli di macchine, compresi tutti i tipi di RAM, abbiano come miglior upper bound per la moltiplicazione tra due numeri interi di N bit $\mathcal{O}(NlogN)$, il valore analogo relativo alle macchine di Turing multidimensionali per la stessa operazione è $\mathcal{O}(NlogNloglogN)$. Questo sembrerebbe 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.

1.3 Equivalenza in tempo reale tra SMM e alcuni modelli RAM

Per descrivere alcune caratteristiche del suo modello, Schönhage si servì di due versioni delle macchine ad accesso casuale, dette RAM (la cui descrizione viene fornita brevemente nei capitoli seguenti). La ragione di questo

suo interesse era la volontà di provare l'equivalenza in tempo reale tra le SMM e due particolari versioni di RAM dette *successor RAM*: esse verranno chiamate in seguito RAM0 e RAM1. Sebbene la struttura di esse sia fondamentalmente simile a quella del modello generale, l'autore precisa alcune differenze atte a renderle più idonee alla gestione di grafi. Questa sezione si occupa di spiegare i passi principali della sua dimostrazione [05]. Per prima cosa, gli interi memorizzati nei loro registri altro non sono che puntatori. Ogni registro n, chiamato *locazione*, rappresenta un singolo nodo da cui partono due puntatori: il primo verso il registro $n+1$, in modo da realizzare una lista concatenata, il secondo verso un qualsiasi nodo $\langle n \rangle$ della struttura chiamato *contenuto* della locazione n. Inoltre le RAM1 utilizzano un ulteriore registro accumulatore il cui contenuto è indicato con la lettera z. Questo registro non è indirizzabile ed è inizializzato a 0 così come tutti gli $\langle n \rangle$. Per quanto riguarda le RAM0, esse utilizzano l'accumulatore già presente nella normale struttura. Anche in questo caso, il valore memorizzato in esso è indicato con z. Oltre a questo, un ulteriore registro contiene l'istruzione corrente n ed è inizializzato a 0.

Istruzioni	**Effetto**	**Spiegazione**
LDA n	$z := n$	carica indirizzo
LDD n	$z := \langle n \rangle$	carica direttamente
LDI n	$z := \langle\langle n \rangle\rangle$	carica indirettamente
STD n	$\langle n \rangle := z$	memorizza direttamente
STI n	$\langle\langle n \rangle\rangle := z$	memorizza indirettamente
COM n	$z := \langle n \rangle ?$	se il confronto è soddisfatto esegue l'istruzione succ.
SUC	$z := z + 1$	computa la funzione successore

Tabella 1.1: Istruzioni interne di RAM1

Istruzioni	**Effetto**	**Spiegazione**
Z	$z := 0$	imposta a zero
A	$z := z + 1$	aggiunge 1
N	$z := n$	imposta l'indirizzo n
L	$n := z$	carica
S	$\langle n \rangle := z$	memorizza
C	$z := 0?$	se il confronto è soddisfatto esegue l'istruzione succ.

Tabella 1.2: Istruzioni interne di RAM0

Le due tabelle Tabella 1.1 e Tabella 1.2 mostrano gli insiemi di istruzioni interne utilizzabili in questi due modelli. Esse sono simili a quelle delle SMM, sebbene presentino alcune leggere differenze. Ogni istruzione è indicata da un codice e da un indirizzo n. Una volta fatta questa descrizione, si può affrontare la dimostrazione della tesi inizialmente esposta

Teorema 1.3.1. *I modelli di macchine SMM, RAM1 e RAM0 sono equivalenti in tempo reale.*

In base alla definizione di equivalenza in tempo reale, ciò significa voler dimostrare

$$RAM1 \xrightarrow{\tau} RAM0 \xrightarrow{\tau} SMM \xrightarrow{\tau} RAM1$$

Provare che RAM0 e RAM1 hanno lo stesso potere computazionale può essere macchinoso ma concettualmente semplice. Poiché le due classi condividono le stesse istruzioni comuni, è sufficiente provare che ogni istruzione interna di RAM1 può essere rimpiazzata con un insieme di istruzioni equivalenti in RAM0. Schönhage ne fornì una dimostrazione nella pubblicazione dedicata al suo modello [05], mostrando il corrispettivo di simulazione per ogni tipologia di istruzione (si veda la Tabella 1.3).

Istruzione in RAM1	**Simulazione**	**Effetto sulla RAM0**
LDA n	$ZANZA^{2n}S$	$\langle 1\rangle := 2n$
LDD n	$ZANZA^{2n}LS$	$\langle 1\rangle := \langle 2n\rangle$
LDI n	$ZANZA^{2n}LLS$	$\langle 1\rangle := \langle\langle 2n\rangle\rangle$
STD n	$ZA^{2n}NZALS$	$\langle 2n\rangle := \langle 1\rangle$
STI n	$ZA^{2n}LNZALS$	$\langle\langle 2n\rangle\rangle := \langle 1\rangle$
COM n; σ^*	$ZALA^3NZS$	$\langle\langle 1\rangle + 3\rangle := 0$
	$ZA^{2n}LA^3NS$	$\langle\langle 2n\rangle + 3\rangle := x\ (\geq 3)$
	$ZALA^3L$	$z := \langle\langle 1\rangle + 3\rangle$
	C; goto λ, σ, λ;	
SUC	$ZANLAAS$	$\langle 1\rangle := \langle 1\rangle + 2$

Tabella 1.3: Simulazione delle istruzioni interne di RAM1

Allo stesso modo, per compiere il passo di dimostrazione successivo si fornisce una simulazione dell'insieme di operazioni di RAM0 realizzata utilizzando istruzioni di SMM (Tabella 1.4). Nell'ultimo passaggio, invece, basta supporre che ogni elemento appartenente a $\Delta = \{0, 1, ..., k-1\}$ sia memorizzato in k locazioni consecutive della RAM1. Ad esempio, l'elemento x viene

Istruzione in RAM0	**Simulazione**
Z	`set` BA `to` A
A	`if` BAA `=` A `then new` BAA
	`set` BA `to` BAA
N	`set` BB `to` BA
L	`set` BA `to` BAB
S	`set` BBB `to` BA
C	`if` $BA = A$ `then` ...

Tabella 1.4: Simulazione delle istruzioni interne di RAM0

memorizzato nelle k locazioni che vanno da $xk + 4$ a $xk + 4 + k$. Inoltre, per ogni puntatore δ uscente dal nodo x, la sua destinazione è memorizzata nella locazione $xk+4+\delta$. Nella locazione 0 viene memorizzato il centro della struttura S, mentre nella locazione 1 è salvato l'indirizzo di memoria contenente il valore da incrementare ogni qualvolta si esegue un'istruzione `new`. Le locazioni 2 e 3 servono per calcolare gli indirizzi utilizzati nelle istruzioni che utilizzano U e V come input. Per realizzare la simulazione in tempo reale, le istruzioni della SMM vanno sostituite con istruzioni equivalenti di RAM1, utilizzando come configurazione iniziale

$$\langle 0 \rangle = 4, \langle 1 \rangle = 4 + k, \langle 4 + \delta \rangle = 4$$

per $0 \leq \delta < d$.

1.4 Simulazione di una macchina di Turing

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 detto *Self Crossing problem*.

Per prima cosa, è necessario descrivere la classe di MdT che si vuole simulare. In particolare, ci si vuole riferire alle macchine che possono consistere 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 di memoria. 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 seguenti istruzioni:

- **head** v; con $v \in \mathbb{N}$ e $v \geq 1$. Le istruzioni successive si riferiscono alla testina contrassegnata dal numero v fino all'istruzione **head** successiva. Dopo l'esecuzione di questo codice v è la testina attiva;

- **write** α; $\alpha \in \{0, 1, \beta\}$, dove β sta per blank. Nella cella sotto la testina attiva viene scritto il carattere α;

- **read** λ_0, λ_1; se la cella sotto la testina attiva contiene 0 il controllo è trasferito all'istruzione etichettata con λ_0. Se la cella sotto la testina

contiene 1 il controllo è trasferito all'istruzione etichettata con λ_1. Se la cella è blank il controllo passa all'istruzione successiva;

- **move** δ; la testina attiva viene spostata di una posizione in direzione δ, dove δ appartiene all'alfabeto delle direzioni.

Il teorema dimostrato da Schönhage [12] è il seguente:

Teorema 1.4.1. *Ogni macchina di Turing appartenente alla classe appena descritta è simulabile da una idonea SMM in tempo reale.*

Si osservi che una MdT con n testine e componenti di memoria diversi di dimensione $K_1, \ldots, K_d$ può apparentemente essere simulata in tempo reale da un'apposita MdT con un solo componente di memoria isomorfo a Z^d.

Si stabilisce inoltre che la SMM simulante una data MdT userà l'alfabeto $\Delta = \{$N, S, W, E, U, D$\}$ dove U e D stanno per up e down. Durante la simulazione i dati e la configurazione della macchina vengono memorizzati in una particolare struttura così composta:

- 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.

1.4.1 Struttura Piramidale

In ogni momento della simulazione, la struttura piramidale P è una parte finita della struttura infinita $\mathcal{P}$ con nodi $M_k = (x, y)\ \forall k \in \mathbb{N}$ con $(x, y) \in \mathbb{Z}^2$. K indica il livello della struttura che si vuole considerare e assume per questa ragione il significato di coordinata verticale. La struttura P si estende quindi su più livelli:

$$\mathcal{L}_k = \{M_k(x, y) \mid (x, y) \in \mathbb{Z}^2\}$$

ognuno dei quali consiste in una griglia rettangolare realizzata dai puntatori N, S, W, E. Da ogni $M_k(x, y)$ partono quattro puntatori, rispettivamente diretti verso $M_k(x, y+1), M_k(x-1, y), M_k(x, y-1), M_k(x+1, y)$. Il sottoinsieme $\{M_k(\zeta, \eta) \mid |\zeta - x| \leq |\eta - y| \leq 1\}$ insieme ai 24 puntatori che connettono due a due questi nodi è detto *vicinanza* di $M_k = (x, y)$. La

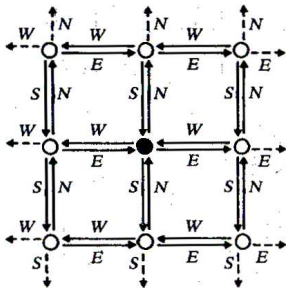


Fig. 1.6: Vicinanza di un nodo

Figura 1.6, tratta (così come le seguenti) dall'articolo di Schönhage che descrive questa struttura [05], rappresenta la vicinanza di un nodo.

$\mathcal{L}_0$ viene utilizzato per rappresentare il piano della MdT. Essendo questo piano potenzialmente infinito la struttura piramidale contiene solo un sottoinsieme finito L_0 di $\mathcal{L}_0$ che rappresenta le aree che sono state visitate almeno una volta da una delle testine. I livelli superiori $\mathcal{L}_1$, $\mathcal{L}_2$,... vengono utilizzati per rappresentare il piano su scale ridotte con rapporto lineare 1:3:9:... In particolare $M_{k+1} = (x, y)$ corrisponde alla vicinanza del nodo $M_k = (3x, 3y)$. La connessione tra i vari livelli viene fornita tramite i puntatori U e D

- Puntatori U: vanno dal nodo $M_k = (x, y)$ al nodo $M_{k+1} = (\lfloor (x + 1)/3 \rfloor, \lfloor (y + 1)/3 \rfloor)$;
- Puntatori D: vanno dal nodo $M_{k+1} = (x, y)$ al nodo $M_k = (3x, 3y)$, $k \geq 1$, mentre sul livello $\mathcal{L}_\prime$ vengono utilizzati per codificare l'informazione contenuta nelle celle della MdT

A questo punto è possibile definire la struttura piramidale come

$$P = L_0 \cup L_1 \cup \ldots \cup L_m$$

dove $\forall k \ 0 \leq k \leq m$ allora L_k è un sottoinseme finito di $\mathcal{L}_k$. Inoltre L_m contiene uno e un solo nodo $M_k(0, 0)$ detto nodo vertice. Le principali proprietà di P sono le seguenti:

(i) Ogni puntatore che parte da un nodo $k \in P$ è diretto o esattamente come lo è in $\mathcal{P}$, o verso il centro A. Nel primo caso il puntatore deve essere correttamente installato e anche il nodo a cui punta deve appartenere a P (questo non vale per i puntatori D del livello L_0);

(ii) Per ogni nodo $K \in P$ (ad eccezione del nodo vertice) sono correttamente installati rispettivamente il puntatore U, la sua destinazione KU, il puntatore D e tutta la vicinanza di KUD.

Dunque P risulta essere un insieme di singoli elementi piramidali composti da 10 nodi tra loro connessi.

1.4.2 Contatori

I contatori vengono utilizzati per tenere traccia della posizione delle varie testine sul piano. Per tracciare una posizione di coordinate (x, y) sarebbe sufficiente utilizzare un puntatore al nodo $M_0(x, y)$ detto H_0. Poichè però si vogliono mantenere i requisiti necessari alla simulazione in tempo reale è talvolta necessario l'accesso immediato alle rappresentazioni di livello superiore di H_0 già appartenenti a P. Per questa ragione esistono dei puntatori ai nodi H_1, H_2, ..., H_t appartenenti ai rispettivi livelli L_1, L_2, ..., L_t che rappresentano la posizione della testina in ogni momento in maniera approssimata. I contatori si occupano dunque di aggiornare dinamicamente questi puntatori basandosi sui movimenti delle testine. Essi controllano una sequenza di stadi numerati in maniera sequenziale. I primi due stadi vengono eseguiti all'inizio della simulazione. Quando viene eseguito l'n-esimo spostamento di una testina, si eseguono i rispettivi stadi $2n + 1$ e $2n + 2$. Nel primo di essi viene aggiornato H_0, nel secondo gli elementi dei livelli superiori. Si può ora introdurre una nuova proprietà di P:

(iii) Se K appartiene alla vicinanza di H_j, allora KU appartiene alla vicinanza di $H_{j+1} (0 \leq j \leq l)$.

1.4.3 Aggiornamento

A questo punto è possibile descrivere la simulazione di una istruzione del tipo `move` δ con $\delta \in \{$N, W, S, E$\}$. Supponiamo si tratti della mossa t-esima della testina numerata con v. Si tratta di realizzare i passi $2t + 1$ e $2t + 2$ del contatore C_v, aggiornando ogni livello della struttura. Per modificare i puntatori dei contatori è sufficiente utilizzare un'istruzione idonea, come ad esempio `set` W `to` $W\delta$. L'aggiornamento di un livello j si divide in preparazione e adattamento. In ogni fase si richiede di tener conto della seguente proprietà

(iv) In ogni momento della simulazione, $\forall j$ relativo a una testina, le vicinanze di H_j sono contenute nella struttura piramidale.

Nella fase di **preparazione** si comincia controllando che la vicinanza di H_j faccia effettivamente parte di P. In caso contrario, si costruiscono gli elementi mancanti utilizzando il seguente algoritmo:

- se $j = l$ allora $H_l = M_l(0, 0)$ e per (ii) esso ha la vicinanza in P, tranne nel caso in cui H_l sia il nodo vertice di P. Questo avviene se l è appena stato creato come nuovo livello, e cioè se il puntatore U da H_l conduce ad A. In questo caso viene creato $M_{l+1}(0, 0)$ come nuovo nodo vetta insieme alla vicinanza di H_l;

- se $j < l$ allora (iii) ci permette di sapere che tutti i nove nodi sono stati posizionati nella vicinanza di H_{j+1}, che per (iv) è definita nella struttura. A questo punto è sufficiente aggiungere i puntatori mancanti.

La fase di **adattamento** sul livello j consiste nello spostare H_j in maniera corretta a seconda dei casi:

- se $j = 0$ allora si rimpiazza H_0 con $H_0\delta$;
- se $j > 0$ allora si rimpiazza H_j con H_{j-1}U.

Quando il livello j è utilizzato per la prima volta, e cioè al passo 2^j, allora l è appena stato incrementato e $j = l$.

1.4.4 Simulazione

Si può specificare adesso il significato dei nodi A_0 e A_1 contenuti nella struttura centrale. Ogni elemento $M_0(x, y)$ serve ad accedere all'informazione β contenuta nella cella di coordinate (x, y) della MdT da simulare. Il puntatore D di ognuno di questi elementi punta al nodo A_β. Se la cella non è inizializzata, il puntatore punta semplicemente al nodo centrale A. L'immagine 1.7 rappresenta questa parte della struttura.

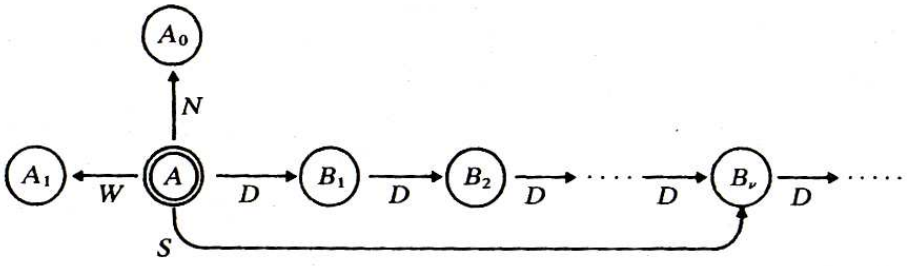


Fig. 1.7: Nodi ausiliari adiacenti al nodo centrale

Per inizializzare la simulazione bisogna quindi creare la struttura centrale, i nodi utilizzati dai contatori e la struttura piramidale contente $M_0(0, 0)$, $M_1(0, 0)$ e le loro vicinanze.

1.5 Lock-step equivalenza tra SMM e Macchine a Stati Astratti

Un altro interessante risultato che concerne le macchine di Shönhage venne fornito da Andreas Blass e Yuri Gurevich nella seconda metà degli anni '90 [06]. Si tratta della dimostrazione della teoria secondo la quale le SMM sono

lock-step equivalenti alle Macchine a Stati Astratti unarie di Gurevich senza funzioni esterne (abbreviato con ASM). Un'ulteriore risultato che è possibile apprendere dall'articolo di Gurevich è la dimostrazione della possibilit à di estendere questo teorema ad ogni ASM generica aggiungendo la funzione *coppia* al modello di Shönhage.

1.5.1 Simulazione lock-step e lock-step equivalenza

Il concetto di lock-step venne definito da Gurevich stesso nella seguente maniera:

Definizione 1.4 (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. 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. A questo punto è possibile dare le seguente definizioni:

Definizione 1.5. $\mathcal{A}$ simula in lock-step $\mathcal{B}$ strettamente se $c = 1$.

Definizione 1.6 (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.

Per la descrizione delle macchine a stati astratti data da Gurevich si veda il capitolo relativo.

1.5.2 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 già è 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 numerare, 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
                δ1(y) := y
                ⋮
                δm(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.

`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.

`output` β

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

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.

`halt`

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

Molto semplicemente indica il termine dell'esecuzione corrispondente allo stato di halt.

`new` $\square$

```
if CurInst = i and Mode = Working
      then
      import y
            Nodes(y) := True
            Center := y
            δ_1(y) := Center
                  ⋮
            δ_m(y) := Center
      endimport
      CurInst := Succ(CurInst)
endif
```

`new` $w_1 \ldots w_k$ $(k \geq 1)$

```
if CurInst = i and Mode = Working
      then
      import y
            Nodes(y) := True
            β(PARENT) := y
            δ_1(y) := δ_1(PARENT)
                  ⋮
            δ_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.

`set` $\square$ `to` $\mathtt{v}_1\mathtt{v}_2\ldots\mathtt{v}_j$	`set` $\mathtt{w}_1\mathtt{w}_2\ldots\mathtt{w}_k$ `to` $\mathtt{v}_1\mathtt{v}_2\ldots\mathtt{v}_j$ $(k \geq 1)$
if CurInst $= i$ and Mode $=$ Working	if CurInst $= i$ and Mode $=$ Working
then	then
Center $:=$ Center.$\mathtt{v}_1$.$\mathtt{v}_2\ldots\mathtt{v}_j$	$\mathtt{w}_k$(Center.$\mathtt{w}_1$.$\mathtt{w}_2\ldots\mathtt{w}_{k-1}$) $:=$
CurInst $:=$ Succ(CurInst)	Center.$\mathtt{v}_1$.$\mathtt{v}_2\ldots\mathtt{v}_j$
endif	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` $\mathtt{u}_1\mathtt{u}_2\ldots\mathtt{u}_k = \mathtt{v}_1\mathtt{v}_2\ldots\mathtt{v}_j$ `then` σ

if CurInst $= i$ and Mode $=$ Working then
  if Center.$\mathtt{u}_1\mathtt{u}_2\ldots\mathtt{u}_k$ $=$ Center.$\mathtt{v}_1\mathtt{v}_2\ldots\mathtt{v}_j$ 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.....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 nè 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 utilizzò 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 $\phi(A_i) = B_i$.

1.5.3 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 stata 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 e una interpretazione dei nomi di funzioni presenti nel vocabolario. 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 $v_1 \ldots v_k$
  R
endimport

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

`new` New_1
$\vdots$
`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 2

Altre macchine a puntatori

Le macchine a modificazione di memoria fanno parte di un più generico insieme di modelli astratti detti *macchine a puntatori*. Nel suo articolo "What is a Pointer Machine?" [02] Amir M. Ben-Amram tentando di dare una definizione più precisa di questa categoria ne individua tre principali esempi in letteratura i quali, in aggiunta a circa tre macchine minori, realizzano l'insieme delle macchine a puntatori. Si tratta delle già citate SMM, delle macchine di Kolmogorov-Uspensky e dei Link Automata di Knuth. Questi tre modelli vennero inizialmente sviluppati dai loro ideatori in maniera del tutto indipendente. Soltanto dopo le prime pubblicazioni a riguardo gli autori vennero a conoscenza della presenza di un'analogia tra il proprio lavoro e quello degli altri due studiosi. Ciò nonostante, sia Schönhage che Kolmogorov proseguirono con i propri studi separatamente, limitandosi a constatare l'esistenza di questa somiglianza e a valutare brevemente le differenze tra i due modelli. Mentre Kolmogorov sosteneva di poter considerare le SSM come un caso particolare di KU dirette, sottolineandone le differenze [08], il suo collega specificò che gli algoritmi di Kolmogorov erano diversi dalle SMM e "sicuramente non più forti, ma forse più deboli" [05]. Schönhage aggiunse anche che mentre le KU sono simulabili in tempo reale dalle SMM, l'inverso non è dimostrato e non è probabilmente vero. Inoltre, affiancando questa osservazione alla dimostrazione di Gurevich [17] secondo il cui risultato le macchine di Kolmogorov non possono essere simulate in tempo reale dalle macchine di Turing, questo porta a supporre che neanche le SMM siano simulabili in tempo reale dalle MdT stesse. Per quanto riguarda Knuth, egli si interessò alle successive stesure degli articoli di Schönhage, al quale suggerì per primo di chiamare il gruppo di automi contenente SMM e Linking Automata con il nome di Pointer Machines. Nei seguenti paragrafi vengono brevemente descritti i modelli di macchine a puntatori di Kolmogorov e Knuth.

2.1 Macchine di Kolmogorov-Uspenksy

Nella letteratura sulle macchine a puntatori il secondo nome che viene più frequentemente citato insieme a Scönhage è quello di Andrei Nikolaevich Kolmogorov. Nel 1958, dodici anni prima che l'idea delle SMM prendesse forma, Kolmogorov pubblicò insieme al suo allievo Vladimir A. Uspenksy un articolo nel quale venivano nominate per la prima volta le cosiddette macchine di Kolmogorov-Uspensky (alle quali ci si riferirà da questo momento con la sigla KU)[14]. L'interesse iniziale di Kolmogorov era quello di sviluppare una nozione formale di *algoritmo*. Come lo stesso Uspenksy spiegò in seguito [07], già nel 1936 videro la luce numerose pubblicazioni di studiosi del calibro di Post, Church, Turing e Klenee che suggerivano nozioni generiche di questo concetto; eppure quello di Kolmogorov fu il primo tentativo completo di formalizzare un'idea per la quale non era ancora stata offerta una chiara definizione matematica. Può essere interessante notare come l'approccio scelto dall'autore facesse precedere al vero e proprio modello matematico la stesura di uno "schema filosofico", una lista di proprietà e caratteristiche relative ad un generico processo algoritmico. Il seguente elenco riporta alcuni dei tratti dello schema filosofico descritto da Uspenksy [07]. Dato un algoritmo Γ:

- In ogni istante è possibile individuare lo stato in cui si trova il processo algoritmico;
- Per alcuni stati S è definito lo stato successore S^*;
- Il processo algoritmico consiste nei vari passaggi tra uno stato S e il successivo S^*. Lo stato iniziale è l'input;
- Quando il processo giunge a uno stato S il cui successore S^* non è definito si parla di *terminazione senza risultato*. Se invece giunge allo stato S che lancia l'istruzione di stop si parla di *terminazione con risultato*. S è detto *soluzione* e Γ è definito solo per gli stati che conducono a una terminazione con risultato;
- Esiste una mappatura parziale Ω_Γ tale che $\Omega_\Gamma(S) = S^*$. Questa funzione esiste per qualsiasi algoritmo Γ.

Sfruttando la definizione di Ω_Γ è possibile individuare il carattere iterativo dei processi (dove Ω è ciò che viene iterato). All'interno dello schema filosofico viene descritta anche l'idea di *zona attiva*, un sottoinsieme ristretto ed "essenziale" di S, la cui dimensione è fissata per ogni singolo algoritmo Γ. Il valore di $\Omega(S)$ dipende non da S nella sua totalità, ma solo dalla sua zona attiva. Inoltre essa è l'unica a subire variazioni in seguito a una iterazione di Ω. Si noti che poichè la zona attiva è di dimensioni limitate per ogni Γ esiste solo un insieme finito di zone non isomorfe; pertanto Ω_Γ può essere

descritto in termini finiti e di conseguenza anche Γ ha carattere finito.
Fissate queste definizioni, per passare alla formulazione matematica è necessario stabilire cosa si vuole intendere con S. La risposta è ciò che viene chiamato *complesso di Kolmogorov*. Sarebbe a dire, un grafo finito connesso da archi unidirezionali che risponde alle seguenti proprietà:

- Per ogni algoritmo esiste un alfabeto, analogo al Δ delle SMM. Ogni arco uscente da un nodo è etichettato da un elemento di questo alfabeto;

- Non esistono due archi uscenti dallo stesso nodo con la stessa etichetta. Questo significa che il numero di puntatori uscenti per nodo è fissato ed è uguale alla cardinalità dell'alfabeto;

- In ogni grafo esiste un nodo distinto detto *nodo attivo*;

- Non solo il numero di archi uscenti da un nodo è limitato da un numero massimo, ma anche il numero degli archi entranti. Questa particolare proprietà può essere realizzata stabilendo che per ogni arco da a in b con etichetta α esiste un arco da b in a con etichetta uguale;

- Per ogni singolo algoritmo Γ esiste una quantità stabilita k. La *zona attiva* è l'insieme dei nodi raggiungibili in massimo k passi.

Date queste definizioni, si può descrivere l'idea di macchina di Kolmogorov: un computer device la cui memoria può cambiare topologia. In altre parole una macchina analoga a quella di Turing, il cui nastro viene però sostituito con un complesso di Kolmogorov. A seconda dell'isomorfismo della zona attiva, il programma di controllo specifica una sequenza di istruzioni, le quali possono eseguire alcuni determinati tipi di azione:

- aggiungere un nuovo nodo e una coppia di archi. Entrambi gli archi devono avere la stessa etichetta;

- rimuovere un nodo e i suoi archi, entranti e uscenti;

- aggiungere o rimuovere una coppia di archi;

- halt.

Nonostante si possa suggerire l'idea di associare l'halt a una particolare configurazione della zona attiva, invece che utilizzare una istruzione apposita, questa soluzione non è considerabile valida. Si considerino lo stato iniziale e lo stato finale di una KU: altro non sono che l'input e l'output della macchina stessa. Sfruttando questa caratteristica, è possibile utilizzare l'output di una macchina come input di un'altra. È chiaro quindi che halt non può essere espresso come stato, a meno di non voler penalizzare la capacità espressiva della macchina stessa. Una volta fissato il tipo di dato D su cui lavorare,

per esprimere ogni funzione da D in D sono necessarie tutte le possibili configurazione del grafo, che non possono quindi essere utilizzate per segnalare l'interruzione del programma.

Una maniera più semplice per descrivere la struttura dati della KU è quella di considerarla come un grafo indiretto. Un arco senza direzione può servire quindi a sostituire una coppia speculare di archi diretti. In questa formulazione, indicata con il nome di *macchina di Kolmogorov indiretta*, bisogna aggiungere alla lista delle proprietà dei grafi una regola che esprima la necessità di mantenere il numero di puntatori entranti in un nodo sotto una certa quantità. Con il nome di *macchina di Kolmogorov diretta*, invece, l'autore indica una macchina con grafo diretto avente la seguente proprietà

Proprietà 1. Per ogni nodo A tutti i nodi che sono al termine degli archi diretti uscenti da A devono avere etichette diverse.

In questo modo il numero di archi uscenti da un nodo continua ad avere un limite massimo, mentre il numero dei nodi entranti può crescere in maniera indefinita. Questa variazione di KU possiede particolare importanza poiché la sua compilazione non può essere vista come la compilazione di una macchina di Kolmogorov indiretta.

2.1.1 KU e tesi di Turing

Secondo Gurevich, Kolmogorov e Uspensky hanno definito e descritto i loro modelli per provare che non c'è modo di estendere il concetto di funzione calcolabile. Si può notare come Kolmogorov stesso, servendosi della sua macchina per dare una definizione di algoritmo, suggerisca che ogni computazione, opportunamente considerata come una successione di istruzioni, possa essere vista come la computazione di un'appropriata KU. In un certo senso, questo è più forte della tesi di Turing. Grigoryev fornì una prova parziale della superiorità delle KU sulle macchine di Turing in termini di potenza [17]. Egli mostrò una funzione computabile in tempo reale da una KU e allo stesso tempo non computabile in tempo reale da nessuna MdT. Nonostante non ci siano prove analoghe relative a computazioni in tempo lineare, si può comunque aggiungere che alcuni teoremi hanno forma concreta nel modello di Kolmogorov, mentre la loro validità non è stata dimostrata nel modello di Turing (e probabilmente non è possibile farlo).

2.2 Macchine di Knuth

All'interno dell'opera "The Art of Computing Program" [04], Donald Knuth ipotizzò una nuova tipologia di macchina astratta in maniera indipendente e quasi contemporanea a Schönhage e Kolmogorov. Ciò nonostante, la struttura dei cosiddetti *Linking Automata*, così l'autore chiamò le proprie macchine, è analoga a quelle dei due precedenti modelli qui descritti. In

particolare, Knuth teorizzò l'implementazione di grafi con nodi contenenti m campi per i puntatori, p campi per l'informazione, n registri per i puntatori e q registri per l'informazione. Essa viene espressa utilizzando dei simboli di un alfabeto stabilito, simboli opportunamente memorizzati nei relativi campi e registri. Per quanto riguarda i puntatori, invece, i campi e i registri ad essi dedicati possono contenere un valore nullo o puntare a uno dei nodi della struttura. I nodi che vengono puntati dai registri di puntatori sono gli unici ai quali è possibile accedere immediatamente. Un'altra sostanziale differenza tra i linking automata e le altre macchine a puntatori sta nell'insieme di azioni che essi possono compiere. In particolare per essi è possibile:

- creare nuovi nodi. Ciò avviene aggiungendo il nodo che si vuole inserire alla lista di quelli immediatamente accessibili. Per rispettare questa richiesta è sufficiente far puntare uno dei registri appositi al nuovo nodo;

- confrontare stringhe di informazione e valori di puntatori. L'unico confronto possibile è quello di equivalenza;

- trasferire stringhe di informazione o valori di puntatori tra registri e nodi.

La ragione per la quale Knuth teorizzò lo studio di queste macchine astratte è analoga alla spiegazione che Schönhage diede ai suoi studi: il modello di Turing, per via delle sue caratteristiche, viene spesso considerato irrealistico. Lo stesso autore ammette che per un numero di nodi molto alto anche i linking automata sarebbero impossibili da implementare nella realtà, rendendo quindi le macchine di Turing più idonee per uno studio di algoritmi con un carico di informazioni tendente a infinito. Ciò nonostante, il fatto che nella realtà i puntatori debbano essere memorizzati con dimensioni fissate e finite ci permette di stabilire un limite massimo di nodi disponibili. Questo rende le macchine di Knuth idonee come modello computazionale realistico in presenza di un carico di informazioni con utilizzi pratici.

Capitolo 3

Riscrittura su term graph

"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." Questa frase, con cui Plump introduce la sua pubblicazione dedicata alla riscrittura su term graph [10], elenca i principali argomenti che verranno trattati in questo capitolo: si fornirà una breve descrizione di term graph, per poi passare alla definizione di riscrittura su grafi. Per quanto riguarda le ragioni che rendono interessante lo studio del modello di term graph, la principale ha a che fare con l'efficienza di questo sistema. Al momento di applicare regole su term graph è infatti possibile ridurre il numero di occorrenze di un termine: per farlo è sufficiente utilizzare dei puntatori che, invece di effettuare una ripetizione, conducano alla prima occorrenza dell'elemento già incontrato. In questo modo l'unica occorrenza risulta *condivisa* e viene valutata solo una volta risparmiando quindi non solo tempo ma anche spazio. Per ulteriori approfondimenti, si rimanda al già citato articolo di Plump.

3.1 Term graph

Esistono varie modalità per rappresentare i term graph; una di queste è l'utilizzo di *ipergrafi*. Dati un insieme Σ di funzioni f con arietà ≥ 0 e su un insieme infinito X di variabili x con arietà $= 0$, un ipergrafo G su essi è un sistema (V_G, E_G, lab_G, att_G) dove

- V_G è un insieme di nodi;
- E_G è un insieme di archi;
- lab_G è una funzione $E_G \rightarrow \Sigma \cup X$, cioè una funzione che etichetta gli archi associandoli a un elemento preso dall'insieme di funzioni e variabili;

- att_G è una funzione $E_G \to V_G^*$ che assegna ad ogni arco e una stringa composta da nomi di nodi in numero uguale alla arietà dell'etichetta di e più 1. Per ogni arco e, se $att_G(e) = v_0 v_1 \dots v_n$ allora v_0 è il nodo risultato e $v_1 \dots v_n$ i nodi argomento. Il nodo risultato è chiamato $res(e)$ mentre i nodi argomenti vengono identificati con $arg(e)$.

Un *percorso* è una sequenza alternata di nodi e interi positivi $\langle v_0, i_1, v_1, \dots, i_n, v_n \rangle$ tale che per ogni intero $j \leq n$ esiste un arco e tale che $res(e) = v_{j-1}$ e v_j sia il nodo i_j-esimo di $arg(e)$. Un nodo v è raggiungibile da un nodo v_0 se esiste un percorso da v_0 a v. Un grafo è detto *aciclico* se non esistono percorsi nei quali lo stesso nodo compare due volte. A questo punto è possibile definire cosa sia un term graph.

Definizione 3.1 (Term Graph)**.** Un ipergrafo G è un term graph se:

- 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 $f(t_1, \dots, t_n)$ dove $f \in \Sigma$ e $(t_1, \dots, t_n)$ sono a loro volta term. Il nodo v in un term graph G rappresenta il seguente valore:

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

dove e è l'unico arco tale che $res(e) = v$ e $arg(e) = v_1, \dots, v_n$. Se $arg(e)$ è vuoto allora $term_G(v) = lab_G(e)$.

Un morfismo tra due ipergrafi G e H è una funzione φ che mappa l'insieme dei nodi di G con l'insieme dei nodi di H in maniera tale da mantenere intatta la struttura. Essa è composta da due sottofunzioni $\varphi_F : V_G \to V_H$ e $\varphi_E : E_G \to E_H$ tali che per ogni arco e:

- $\varphi_E(lab_G(e)) = lab_H$;
- $\varphi(att_H(e)) = \varphi_F^*(att_H(e))$.

Se φ_E e φ_F sono iniettive e/o suriettive allora lo è anche φ. Se φ è biettiva allora G e H sono *isomorfici* ($G \simeq H$). Dato un nodo $v \in V_G$, il *sottografo* di G avente v come radice è il term graph denotato con $G \downarrow v$ tale che:

- $V_{G\downarrow v}$ è il sottoinsieme di V_G comprendente i nodi raggiungibili da v;
- $E_{G\downarrow v}, lab_{G\downarrow v}$ e $att_{G\downarrow v}$ sono le appropriate restrizioni di E_G, lab_G e att_G.

Si definisce inoltre per ogni term t il corrispondente term graph t'. Esiste un grafo morfismo $f : E_t \to t'$ tale che per ogni coppia di archi e_1 e e_2 allora $f_E(e_1) = f_E(e_1)$ se e solo se $lab_{t'}(e_1) = lab_{t'}(e_2)$ ed entrambi $\in X$. Inoltre, è detto $\overline{t'}$ il grafo risultante dall'eliminazione degli archi etichettati con nomi di variabili appartenenti a t'.

3.2 Sistemi di riduzione

I sistemi di riduzione eseguono computazioni per mezzo di una trasformazione a passi di oggetti. Essi permettono di studiare le proprietà astratte delle relazioni tra i vari oggetti, che possono essere separate dalle proprietà che dipendono dalla struttura dell'oggetto stesso.
Un sistema di riduzione astratto è una coppia $\langle A, \to \rangle$ dove A è un insieme e $\to$ è una relazione binaria su A. Dati due elementi $a, b \in A$ se $\langle a, b \rangle \in \to$ allora si può scrivere $a \to b$. Di seguito sono elencati alcuni particolari esempi di relazioni:

- la **funzione di identità** è la relazione $\to^0 = \{\langle a, a \rangle | a \in A\}$;
- la **chiusura riflessiva** di $\to$ è la relazione $\to^= = \to \cup \to^0$;
- la **chiusura transitiva** di $\to$ è la relazione $\to^+ = \cup_{n \geq 0} \to^n$;
- la **chiusura transitiva riflessiva** di $\to$ è la relazione $\to^* = \to^+ \cup \to^0$;
- la **chiusura simmetrica** di $\to$ è la relazione $\leftrightarrow = \to \cup \leftarrow$;
- la **chiusura di equivalenza** di $\to$ denotata da $\leftrightarrow^*$ è la chiusura transitiva riflessiva di $\leftrightarrow$.

Un elemento a è detto *in forma normale* se non esiste alcun b tale che $a \to b$ e ha forma normale se $a \to^* b$ per qualche b in forma normale.
Inoltre si dice che una relazione $\to$ è:

- **terminante** se non esiste una sequenza infinita di relazioni del tipo $a_1 \to a_2 \to \ldots$;
- **normalizzante** se ogni elemento di A ha forma normale;
- **di Church-Rosser** se per ogni a e b per i quali vale che $a \leftrightarrow^* b$ esiste un c tale che $a \to^* c$ e $b \to^* c$;
- **confluente** se per ogni a, b, c tali che $a \to^* b$ e $a \to^* c$ esiste qualche d tale che $b \to^* d$ e $c \to^* d$;
- **localmente confluente** se per ogni a, b, c tali che $a \to b$ e $a \to c$ esiste qualche d tale che $b \to^* d$ e $c \to^* d$;

- **subcommutativa** se per ogni a, b, c tali che $a \to b$ e $a \to c$ esiste qualche d tale che $b \to^{=} d$ e $c \to^{=} d$;

- **convergente** se è terminante e confluente.

Si noti che una funzione terminante è anche normalizzante, la confluenza implica la confluenza locale e una relazione terminante è confluente se e solo se è localmente confluente.

3.2.1 Sistema di riduzione su term

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 $t \to_E u$ 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 $\sigma : T_{\Sigma,X} \to T_{\Sigma,X}$ 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 posso esprimere le singole regole nella forma $l \to r$. Una *relazione di riscrittura* $\to$ su $T_{\Sigma,X}$ indotta da R è definita come $t \to u$ se esistono una regola $l \to_E r$ in R e una sostituzione σ tali che $\sigma(l)$ è un sottotermine di t e u è ottenuto da t sostituendo ogni occorrenza di $\sigma(l)$ con $\sigma(r)$.

3.3 Riscrittura su term graph

La definizione di riscrittura su term graph fornita da Plump [10] si basa sui concetti introdotti nelle sezioni precedenti. Verranno riportati di seguito i passaggi più importanti della sua pubblicazione, compreso un breve esempio di applicazione di una regola su term graph.

Definizione 3.2 (Istanza)**.** Un term graph L è detto istanza del term l se esiste un grafo morfismo $f : \overline{l'} \to L$ tale che $f(root_{l'}) = root_L$.

Definizione 3.3 (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 3.4 (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.

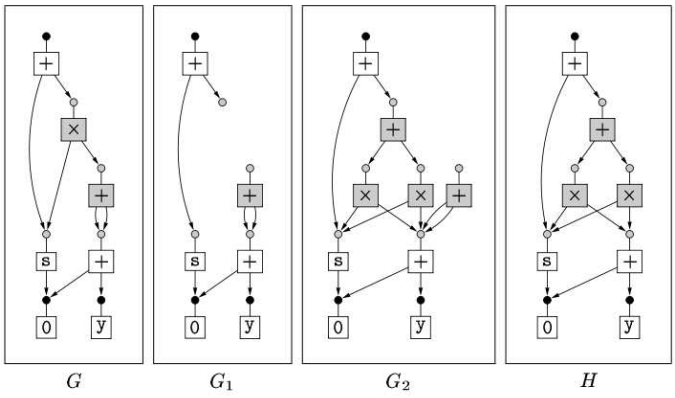


Fig. 3.1: L'applicazione di una regola

La Figura 3.1 rappresenta l'applicazione al grafo G della regola $x \times (y + z) \to (x \times y) + (x \times z)$. Le sezioni di grafo grige in G e H rappresentano i term graph rispettivamente di $x \times (y + z)$ e di $(x \times y) + (x \times z)$.

Capitolo 4

Term graph rewriting e macchine di Schönhage

In questo capitolo verranno illustrati i risultati originali ottenuti in seguito all'identificazione del modello di Schönhage come potenziale simulatore dei meccanismi di riscrittura su grafi. Ciò che in particolare si è voluto dimostrare è la possibilità di realizzare una SMM, opportunamente programmata, capace di applicare in maniera automatica una regola di riscrittura ad un term graph dato. Parte del lavoro ha avuto come obiettivo la creazione di strutture ausiliarie che permettessero di astrarre i dettagli implementativi più complessi al momento di occuparsi della simulazione vera e propria. Queste strutture verranno illustrate nella seguente sezione.

4.1 Strutture ausiliarie

L'insieme di istruzioni che realizzano il linguaggio delle macchine di Schönhage è sostanzialmente primitivo e limitato. In particolare, la mancanza di cicli *while* rende la realizzazione di certi procedimenti estremamente meccanica e ripetitiva. Per questa ragione, al momento di discutere di concetti relativi a term graph rewriting si utilizzeranno istruzioni non appartenenti alla lista precedentemente descritta ed espressioni generiche come *inserire un puntatore*. In questa sezione si vuole dimostrare che i meccanismi di astrazione utilizzati sono legittimi e cioè che è possibile sostituirli con una sequenza di codice composta dalle istruzioni originali di Schönhage.

4.1.1 Caratteri di input/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. Stabiliamo 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 un carattere di un byte. Come è noto, Δ è finito e pertanto l'insieme delle possibili funzioni utilizzabili Δ_f è noto prima della compilazione. Dato quindi $\Delta_f = \{f_0, f_1, \ldots, f_n\}$ con $\Delta_f \in \Delta$ vogliamo poter utilizzare l'istruzione

`input` λ_{f_0}, λ_{f_1}, ..., λ_{f_n}

che legga non più un solo bit ma la quantità di bit che corrisponde alla lunghezza standard prestabilita e che, a seconda del valore della sequenza di bit letti f_i, trasferisca il controllo all'istruzione la cui etichetta è input λ_{f_i}. Realizzare questa istruzione è molto semplice: basta utilizzare la codifica binaria dei singoli caratteri e utilizzare istruzioni `input` λ_0 λ_1 in successione scegliendo in maniera opportuna i valori di λ_0 λ_1. Si finga ad esempio che Δ_f contenga i nomi di funzioni identificati dalle stringhe binarie $010, 100$ e 011. Un esempio di codice che simuli il comando

`input` λ_{010}, λ_{100} λ_{011}

potrebbe essere il seguente:

λ_0 `input` λ_1 λ_2
λ_1 `input` λ_3 λ_4
λ_2 `input` λ_5 λ_6
λ_3 `input` λ_e λ_{err}
λ_4 `input` λ_{010} λ_{011}
λ_5 `input` λ_{100} λ_{err}
λ_6 `input` λ_{err} λ_{err}

dove λ_{err} è l'etichetta corrispondente all'istruzione che si vuole eseguire nel caso di lettura di un carattere non valido o inaspettato, ad esempio halt.

4.1.2 Esistenza di un nodo

Una caratteristica che ha inizialmente ostacolato il tentativo di individuare una modalità per automatizzare la ricerca di particolari pattern all'interno della struttura è la mancanza di istruzioni riguardanti gli spostamenti all'interno del grafo. L'unica maniera per accedere a un nodo è utilizzare il particolare percorso che porta ad esso e che altro non è che il valore del nodo stesso. Si ricordi infatti che l'informazione, nei grafi di Schönhage, è memorizzata non nei singoli nodi ma nella serie di etichette degli archi utilizzati per accedervi. Per un operatore umano con la possibiltà 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. Inoltre, essendo l'input fornito in maniera sequenziale, non è neppure possibile accedere al log delle istruzioni precedenti. Si è deciso quindi di introdurre una serie di puntatori ausiliari, che serviranno a semplificare la manipolazione delle strutture.

Il primo di questi elementi ausiliari è il nodo `exist.` Esso viene utilizzato per stabilire se un nodo identificato da un particolare percorso di etichette sia effettivamente esistente all'interno del grafo. Il nodo `exist` è puntato direttamente dal centro della struttura con un puntatore dedicato etichettato con l'elemento `exist`.

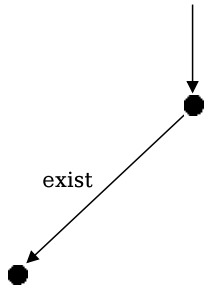


Fig. 4.1: Inizializzazione del nodo `Exist`

Ogni nodo, al momento del suo inserimento, punta a questo elemento sempre utilizzando l'arco etichettato con `exist`. Ad esempio, la Figura 4.2 rappresenta lo stato del grafo della figura 4.1 dopo la creazione di un nodo.

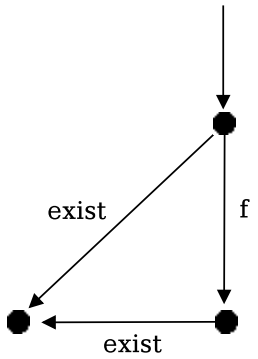


Fig. 4.2: Inizializzazione di un nodo a `exist`

Le istruzioni della macchina corrispondenti a questo passaggio sono veloci e intuitive:

```
new f
set fexist to exist
```

Si consideri adesso l'istruzione

```
if U = V then σ
```

Si suppone che se U è un elemento appartenente alla struttura mentre V non lo è, la clausola dell'if viene valutata come falsa e l'istruzione σ saltata. Per questa ragione, l'istruzione

```
if exist = elementoexist then σ
```

per ogni *elemento* $\in \Delta^*$ verrà saltata solamente se l'elemento non è mai stato inizializzato. Questa istruzione verrà espressa nella forma semplificata

```
if exist elemento then σ
```

Puntatore null

Consideriamo adesso una situazione come quella rappresentata dalla Figura 4.3.

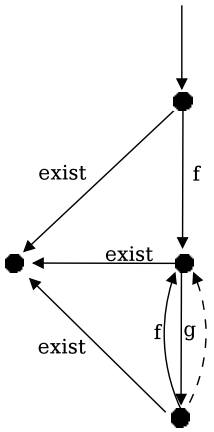


Fig. 4.3: Inizialmente i puntatori conducono al nodo padre

Tutti i puntatori uscenti dal nuovo nodo g non inizializzati puntano a f. Supponiamo 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 a un nodo

effettivamente esistente, e cioè f stesso. Per questa ragione si è deciso di introdurre il nodo `null`.

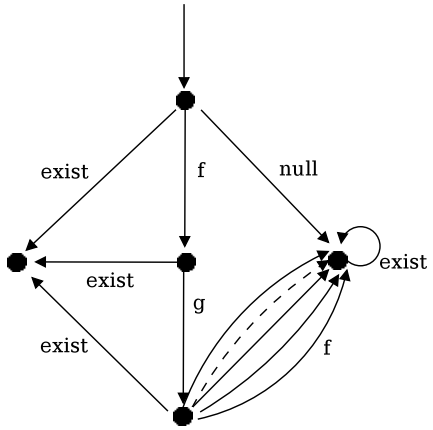


Fig. 4.4: Nodo null

Al momento della creazione di ogni nodo del grafo tutti i puntatori diretti al padre per default vengono indirizzati al nodo `null`. Ciò è possibile perché conosciamo fin dall'inizio gli elementi di Δ.

```
new f
set ff0 to null
set ff1 to null
⋮
set fnull to null
set fexist to exist
```

per ogni $f\delta$ con $\delta \in \Delta$. Ovviamente tutti i puntatori uscenti da *null* puntano a *null* stesso, compreso *nullexist*. Sia *null* che *exist* fanno parte di $\Delta_a \in \Delta$, l'insieme di elementi ausiliari. In questa maniera, il percorso fgf punterà al nodo *null* e il controllo

$$fgfexist = exist$$

restituirà un valore falso. Nei grafi che verranno presentati a partire da questo momento tutti i puntatori a *null* e a *exist* non verranno rappresentati per questioni di ordine, a parte i casi in cui essi vengano effettivamente manipolati. Inoltre, il processo di inizializzazione di questi due elementi e dei puntatori ad essi verrà omesso dal codice mostrato. Ciò nonostante si ricordi che le azioni sopra elencate verranno eseguite al momento della creazione di ogni nodo appartenente al grafo.

4.1.3 Rappresentazione di funzioni

Fino a questo momento il Δ della macchina di Schönhage qui descritta comprende i sottoinsiemi

- Δ_f: insieme di nomi di funzioni, ognuno dei quali può essere rappresentato con una specifica codifica binaria;
- Δ_a: insieme di elementi ausiliari, come exist e null.

A questo punto si vuole introdurre un nuovo sottoinsieme di Δ detto Δ_{fa}. Esso contiene, per ogni elemento $f_i \in \Delta_f$ tanti elementi quanto vale l'arietà di f_i. Ad esempio, data la funzione binaria $g \in \Delta_f$ esistono $g_1, g_2 \in \Delta_{fa}$. Data la funzione unaria h esiste il corrispettivo h_1, mentre per la funzione nullaria l non esistono elementi associati. La Δ struttura sarà costruita in modo tale che per ogni arco f, con f simbolo di una funzione binaria, vengano definiti due archi f_1 e f_2 uscenti dal suo nodo di arrivo. La Figura 4.5 rappresenta una Δ-struttura con f binario, g unario e h nullario. Nella immagine non sono stati disegnati gli archi ausiliari ma solamente quelli che rappresentano l'effettiva informazione.

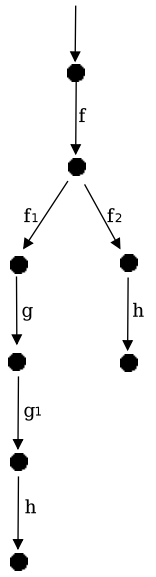


Fig. 4.5: Una Δ-struttura

4.2 Output, input e visita di un grafo

Il problema finale da risolvere consiste nel definire gli algoritmi necessari per visitare il grafo rappresentato dalla Δ-struttura. Si ricordi che Kolmogorov

sosteneva di poter utilizzare l'output di una KU come input di un'altra macchina. Quello che si vuole riuscire a realizzare è lo stesso risultato anche per quanto riguarda le SMM. Durante la sperimentazione svolta sono state testate varie modalità per permettere la visita dei grafi sia in profondità che in ampiezza. I meccanismi studiati sono del tutto analoghi a quelli di input e output. Data la complessità degli algoritmi utilizzati si è deciso di riportare un esempio: in questa sezione si procederà a mostrare come la struttura di un grafo possa essere descritta nella stringa di output e in seguito ricostruita su un'altra macchina utilizzando la stessa stringa come input. Questo risultato prova che è possibile compiere una simile operazione e può essere facilmente generalizzato ad altri casi in maniera intuitiva. Una premessa importante è quella che segue: si supporrà di lavorare su grafi con un numero massimo di nodi prestabilito (e dunque anche con una profondità massima ricavabile con un semplice calcolo basato sull'arietà minima delle funzioni di Δ_f che, come già detto, supponiamo di conoscere già in precedenza).

4.2.1 Strutture ausiliarie

Nella dimostrazione della quale questa sezione si occupa vengono utilizzati degli elementi ausiliari che si aggiungono all'insieme Δ_a precedentemente descritto. Essi sono i seguenti:

- `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 $1, 2, \ldots, p$, 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 $1, 2, \ldots, n$ 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.

Al momento dell'individuazione, queste strutture in aggiunta a `exit` e `null` vengono create. Da questo punto della trattazione, la loro presenza sarà considerata implicita.

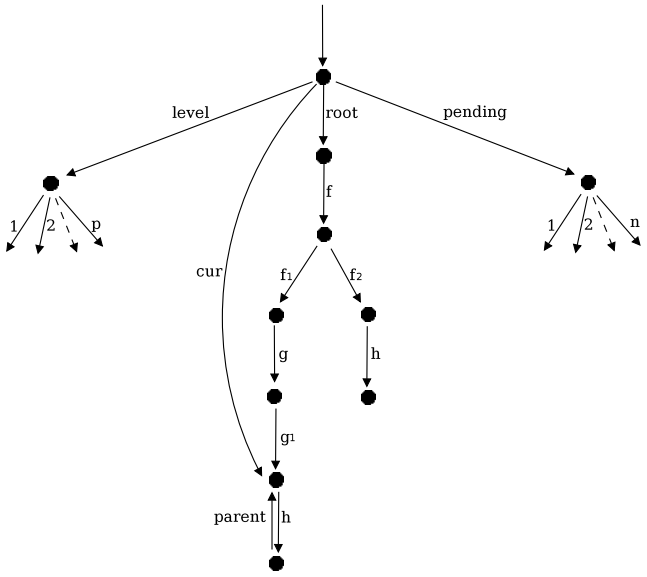


Fig. 4.6: Una Δ-struttura contenente gli elementi ausiliari

4.2.2 Visite e output

Viene ora descritto un esempio di visita in profondità utilizzata per restituire in output gli elementi della Δ-struttura. Questo procedimento si divide in due parti: individuazione nodi e individuazione archi aggiuntivi. La Figura 4.7 rappresenta il grafo scelto.

Si definiscono $\Delta_f = \{f, g, h, q\}$ e $\Delta_{fa} = \{f_1, f_2, g_1, q_1, q_2\}$. La funzione h è nullaria. Si è deciso di esprimere i caratteri delle stringhe in input/output non in binario ma direttamente con i nomi degli elementi di Δ per facilitare la comprensione. Verrà inoltre utilizzato il simbolo $\sharp$ per determinare la fine di una particolare sezione della sequenza.

Individuazione dei nodi

In questa pagina verrà presentato il codice relativo alla fase di individuazione dei nodi. Ogni etichetta inizierà con i caratteri *in* per segnalare la zona di codice nella quale il programma si trova. Ovviamente i valori p, $p-1$, n, ... verranno sostituiti con i valori effettivi. Normalmente gli elementi appartenenti a Δ dovrebbero tutti avere la stessa dimensione e lunghezza. Per facilitare la comprensione di questo esempio, verranno inserite alcune eccezioni (ad esempio `curr`) che andrebbero però sostituite con nomi adatti al momento di un'effettiva compilazione di questo programma. Il simbolo $\square$ indica il nodo centrale. L'istruzione `halt` è stata scelta come istruzione

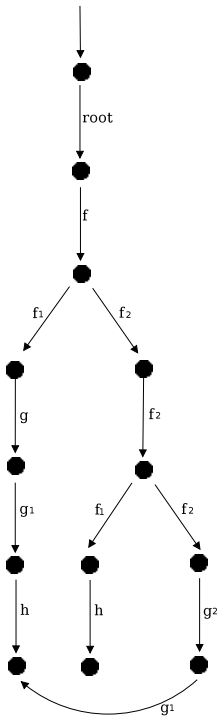


Fig. 4.7: Il grafo utilizzato come esempio

da compiere nel caso in cui il sistema finisca in uno stato incoerente. Per abbreviare e facilitare la comprensione del codice, si utilizza talvolta l'indice i per indicare che il relativo blocco di istruzioni è ripetuto per ogni valore di i compreso in un intervallo stabilito dalla natura del blocco (solitamente $1 \leq i \leq p$ oppure $1 \leq i \leq n$).

etichette istruzioni

```
/*la radice root viene inizializzata come nodo corrente*/
in0     set curr to root
        set currvisited to visited

/*viene cercato il valore successivo*/
next    if exist currf then goto inf
        if exist currg then goto ing
        if exist currh then goto inh0
        if exist currq then goto inq

/*si controlla se non sia stato visitato precedentemente*/
inf     if currfvisited = visited goto inpendf
```

```
        goto inf0

ing     if currgvisited = visited goto inpendg
        goto ing0

inq     if currqvisited = visited goto inpendq
        goto inq0

/*si manda in output il simbolo rappresentante l'elemento individuato*/
inf0    output f
incheckfif level1 = null then goto inf0l1
        if level2 = null then goto inf0l2
        ⋮
        if levelp = null then goto inf0lp

ing0    output g
        set curr to currgg1
        set currvisited to visited
        goto next

inh0    output h
        goto stop

inq0    output q
incheckqif level1 = null then goto inq0l1
        if level2 = null then goto inq0l2
        ⋮
        if levelp = null then goto inq0lp

/*se necessario, si imposta la struttura ausiliaria level*/
inf0l1  set level1f1 to currf
        set curr to currff1
        set currvisited to visited
        goto next
        ⋮
inf0lp  set levelpf1 to currf
        set curr to currffp
        set currvisited to visited
         goto next

inq0l1  set level1q1 to currq
        set curr to currqq1
```

```
        set currvisited to visited
        goto next
        ⋮
inq0lp  set levelpq1 to currq
        set curr to currqqp
        set currvisited to visited
        goto next

/*si raggiunge la fine di un ramo*/
stop    if exist levelp then goto bakcp
        if exist levelp − 1 then goto backp-1
        ⋮
        if exist level1 then goto back1
        output ♯
        goto ie

/*viene ripristinato un ramo precedentemente lasciato in sospeso*/
back1    set curr1 to level1
        if exist curr1f1 then goto f1
        if exist curr1q1 then goto q1
         ⋮
backp   set curr1 to levelp1
        if exist curr1f1 then goto fp
        if exist curr1q1 then goto qp

fi      set curr to curr1f1f2
        set cur1f1 to cur1
        set cur1 to □
        goto next

qi      set curr to curr1q1q2
        set cur1q1 to cur1
        set cur1 to □
        goto next

inpendf if pending1 = null then goto inpendf1
        ⋮
        if pendingp = null then goto inpendfp
        halt

inpendg if pending1 = null then goto inpendg1
```

```
        ⋮
        if pendingp = null then goto inpendgp
        halt

inpendq if pending1 = null then goto inpendq1
         ⋮
        if pendingp = null then goto inpendqp
        halt

inpendfiset pendingi to currf
        output xi
        goto stop

inpendgiset pendingi to currg
        output xi
        goto stop

inpendqiset pendingi to currq
        output xi
        goto stop
```

L'istruzione `goto ie` serve a passare alla seconda parte del programma. Una volta completata l'esecuzione di questa porzione di codice saranno stati individuati tutti i nodi appartenenti al grafo.
Se la Δ struttura rappresentasse il grafo inizialmente presentato come esempio, l'output relativo all'esecuzione di questa parte di codice sarebbe

```
fghfhjx1♯
```

Individuazione degli archi aggiuntivi

Prima di affrontare la seconda parte dell'algoritmo per la visita e l'output è necessario fare un breve discorso sulla possibilità di stampare in output il percorso di un nodo.

Supponiamo quindi di avere un nodo puntato da un arco ausiliare (ad esempio `curr`) diretto dal centro e di voler stampare in output il nome di quel nodo realizzato dal percorso di puntatori propri del grafo. Per farlo, si utilizzano i puntatori ausiliari `level` e `parent`.

```
set level1 to curr
if currpadre = root goto λx
set curr to currpadre
```

```
set level2 to curr
set curr to currpadre
if currpadre = root goto λx
⋮
set levelp to curr
goto λx
goto λx
```

Dopo questa fase iniziale, si visita nuovamente l'albero esaminandolo nodo per nodo. Trovata una corrispondenza tra `nodox` e `leveli` si manda in output l'ultimo arco che porta a `nodox`. Così facendo, i caratteri stampati identificheranno il percorso che conduce al nodo richiesto. Da questo momento, per abbreviare, questa procedura verrà indicata con l'istruzione

`output percorso` U

Questo codice risulta utile nella seconda fase della visita. In essa, infatti, vengono stampati in output i nomi dei nodi ai quali dovranno puntare i vari elementi identificati con pending. In particolare, il codice relativo è il seguente

etichette istruzioni

```
ie      if exist pendingp goto iep
        ⋮
        if exist pending1 goto ie1
        halt
ie1     output percorso pending1
        output ♯
        output ∗
        halt

ie2     output percorso pending1
        output ♯
        output percorso pending2
        output ♯
        output ∗
         halt

iei     output percorso pending1
        output ♯
        ⋮
        output percorso pendingi
```

```
output ♯
output ∗
halt
```

Nel caso del nostro grafo l'output finale sarà quindi

`fghfhjx1♯ff1gg1h♯∗`

4.2.3 Input

Si supponga di voler utilizzare la stringa della fase di output come input per ricostruire il grafo. Viene adesso mostrato il programma realizzato per leggere in input una Δ-struttura con le caratteristiche del grafo del paragrafo precedente. Dopo aver inizializzato i puntatori ausiliari, viene letto un carattere per volta utilizzando il seguente codice:

etichette istruzioni

`begin` `set` *curr* `to` *root*

`input` $\lambda_f\ \lambda_g\ \lambda_h\ \lambda_q\ \lambda_\sharp\ \lambda_{x1}\ \ldots\ \lambda_{xn}\ \lambda_{f1}\ \lambda_{f2}\ \lambda_{g1}\ \lambda_{q1}\ \lambda_{q2}\ \lambda_*$

*/*inserimento di un nuovo nodo*/*

λ_f
```
new currf
new currff1
new currff2
if level1 = null goto level1f
⋮
 if levelp = null goto levelpf
halt
```

λ_g
```
new currg
new currgg1
set curr to currgg1
goto input
```

λ_h
```
new currh
goto back
```

λ_q
```
new currq
new currq1
new currqq2
if level1 = null goto level1q
```

```
        ⋮
        if levelp = null goto levelpq
        halt

/*eventuale inizializzazione della struttura ausiliaria per la gestione dei livelli*/
levelif set leveli to currff2
        set curr to currff2
        goto input

leveliq  set leveli to currqq2
        set curr to currqq2
        goto input

/*viene ripristinato un livello precedentemente lasciato in sospeso*/
back    if exist levelp then goto backp
        ⋮
        if exist level1 then goto back1
        halt

backi   set curr to leveli
        set leveli to level
        goto input

λxi      set pendingi to curr
        goto back
```

Al termine della lettura della porzione di input `fghfhjx1` la struttura creatasi conterrà ogni nodo esistente. A questo punto è necessario aggiungere gli archi rimanenti. Così come è stato possibile realizzare una istruzione per leggere in input una stringa di bit, cioè un carattere invece che un singolo bit, allora intuitivamente si potrà realizzare anche una istruzione che legga una stringa di caratteri e conduca al nodo relativo. Riportato in seguito, un frammento di codice capace di realizzare questa azione

etichette istruzioni

```
begin   set curr1 to root

input   input λf λg λh λq λf1 λf2 λg1 λq1 λq2 λ♯

λf      set curr1 to currf
         input
```

$\vdots$

λ_{q2} `set curr1 to currq2`
`input`

$\lambda_\sharp$ `goto altrocodice`

Questa procedura verrà abbreviata con la singola istruzione

`set curr1 to input percorso`

A questo punto è dunque possibile aggiungere gli archi rimanenti leggendoli dalla sezione finale della stringa in input. La lettura del simbolo $*$ indica la fine della creazione della struttura.

etichette istruzioni

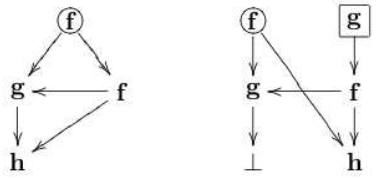
$\lambda_\sharp$ `if exist` $pending1$ `then goto setpen1`

`setpendiset curr to` $pendingiparent$
`set curr1 to input percorso`
`if` $currf1$ `=` $pendingi$ `then set` $currf1$ `to` $curr1$
`if` $currf2$ `=` $pendingi$ `then set` $currf2$ `to` $curr1$
`if` $currg1$ `=` $pendingi$ `then set` $currg1$ `to` $curr1$
`if` $currq1$ `=` $pendingi$ `then set` $currq1$ `to` $curr1$
`if` $currq2$ `=` $pendingi$ `then set` $currq2$ `to` $curr1$
`if exist` $pending(i+1)$ `then goto setpen(i+1)`
`halt`

λ_* `halt`

4.3 Applicazione di una regola di term rewriting a una Δ-struttura

Si vuole adesso dimostrare che è possibile scrivere un programma per SMM capace di riconoscere una data regola di term rewriting su un grafo in input e di applicarla. Verranno utilizzati come esempio un term graph e la regola di riscrittura tratti dalla pubblicazione di Dal Lago e Martini "Derivational Complexity is an Invariant Cost Model" [09]. In seguito è riportata l'immagine originale.



Poiché è stato dimostrato che è possibile implementare meccanismo di input, output e visite sulle macchine di Schönhage, nella seguente sezione i dettagli implementativi relativi a questi algoritmi saranno considerati impliciti per permettere una migliore comprensione dei nuovi concetti presentati. Il programma è strutturato in tre parti: la fase di individuazione, la fase di costruzione e la fase di redirezione.

4.3.1 Fase di individuazione

Nella fase di individuazione si analizza il grafo mano a mano che esso viene inserito nella Δ-struttura, cercando di riconoscere gli elementi appartenenti ad un eventuale redex su cui poter applicare la data regola. 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 del programma. Per quanto riguarda l'esempio trattato, la Figura 4.8 rappresenta l'albero degli stati del sistema, e quali particolari input causano cambiamenti di stato.

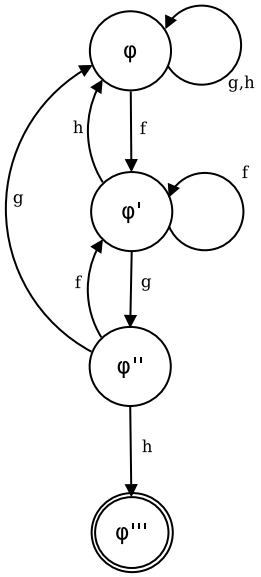


Fig. 4.8: Diagramma degli stati

Verranno ora descritte passo a passo le varie fasi della simulazione e la descrizione delle operazioni svolte. Si è deciso per questioni di semplicità di inserire i nodi per ampiezza.

Inizialmente la struttura contiene soltanto il nodo centro. Serve quindi creare la radice dell'albero. Lo stato del sistema è φ e l'istruzione utilizzata per realizzare la struttura della Figura 4.9 è `new root`. Inoltre la radice dell'albero viene settata come nodo corrente.

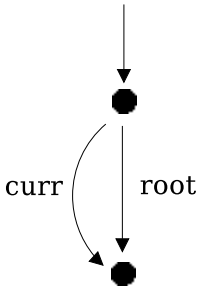


Fig. 4.9: Inizializzazione. Stato: φ

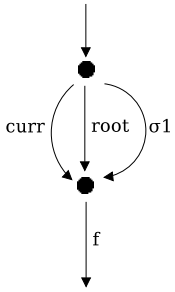


Fig. 4.10: Lettura di f. Stato: φ'

La Figura 4.10 rappresenta l'inserimento del primo nodo vero e proprio. In questo caso il primo elemento aspettato è esattamente una funzione f. Per questa ragione avviene un aggiornamento di stato e il nuovo nodo viene contrassegnato con il puntatore apposito σ_1 tramite l'istruzione `set`.

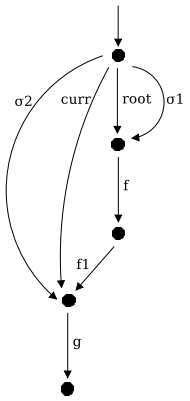


Fig. 4.11: Lettura di g. Stato: φ''

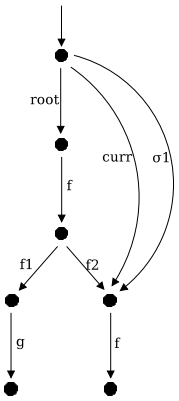


Fig. 4.12: Lettura di f. Stato: φ'

L'elemento successivamente inserito corrisponde nuovamente a quello aspettato per l'effettiva applicazione della regola. Dunque lo stato del sistema viene nuovamente aggiornato e il nodo g viene contrassegnato con il puntatore σ_2 (Figura 4.11).

Se il nodo successivo inserito fosse h allora sarebbe possibile individuare anche l'ultimo elemento necessario per l'applicazione della regola e il programma procederebbe con la fase seguente. Ma poiché il nodo inserito non è g, è necessario resettare tutti i puntatori σ dal momento che non sono state soddisfatte tutte le condizioni per l'applicazione della regola.

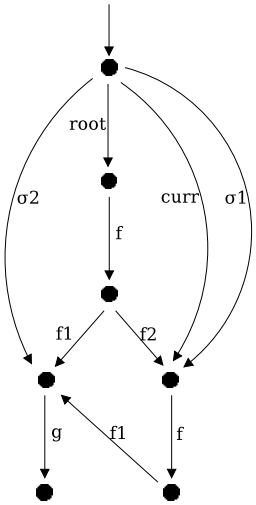


Fig. 4.13: Lettura di g. Stato: φ''

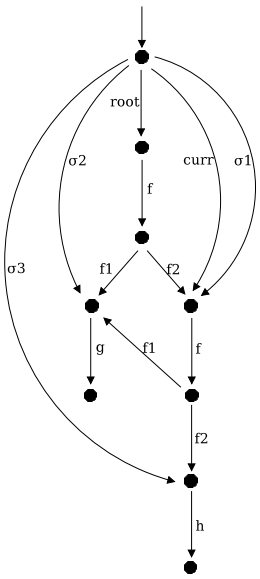


Fig. 4.14: Lettura di h. Stato: φ'''

Ciò nonostante, si può osservare come lo stato del sistema passi direttamente a φ' visto che l'ultimo elemento inserito è f. Il nuovo nodo potrebbe quindi corrispondere alla radice di un sottografo che corrisponda al redex cercato. Le immagini 4.13 e 4.14 mostrano infatti come i due successivi passi dell'inserimento conducano al riconoscimento di una struttura idonea per l'applicazione della regola. I nodi del sottografo individuato vengono puntati dagli appositi archi σ. La fase di individuazione si conclude inserendo gli elementi rimanenti che vengono letti dalla stringa dell'input (Figura 4.15). Si passa quindi alla fase di costruzione.

4.3.2 Fase di costruzione

In questa fase dell'esecuzione si costruiscono 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`.

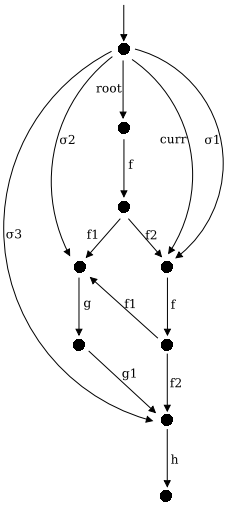


Fig. 4.15: Inserimento del grafo rimanente. Stato: φ'''

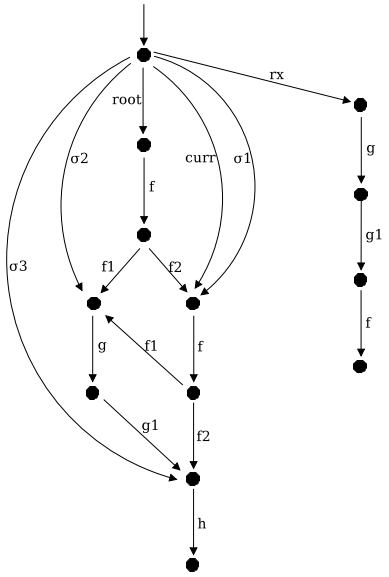


Fig. 4.16: Fase di costruzione

In questo caso particolare, le istruzioni relative (illustrate dalla Figura 4.16) saranno le seguenti:

`new` rx
`new` rxg
`new` $rxgg1$
`new` $rxgg1f$

4.3.3 Fase di redirezione

A questo punto l'unica operazione rimanente consiste nell'inizializzare i puntatori dalla struttura `rx` al grafo originale. Poiché i nodi che vengono utilizzati in questa fase sono già stati individuati si tratta di un processo molto rapido

`set` $rxgg1ff1$ `to` $\sigma 2$
`set` $rxgg1ff2$ `to` $\sigma 3$

Inoltre bisogna togliere il puntatore alla vecchia testa del grafo in modo da non potervi più accedere.

etichette istruzioni

```
        set curr to r1parent
        if currf1 = σ1 then goto λf1
        if currf2 = σ1 then goto λf2
        ⋮
λf2     set currf2 to rx
        set rx to root
        set σ1 to □
        set σ2 to □
        set σ3 to □
```

Il grafo dopo l'applicazione della regola è illustrato dalla (Figura 4.17). Si noti che la fase di garbage collection, oltre a non essere implementabile su questo modello a causa dell'assenza di istruzioni per l'eliminazione di nodi esistenti, non è necessaria: è sufficiente rendere i nodi che andrebbero eliminati non più raggiungibili.

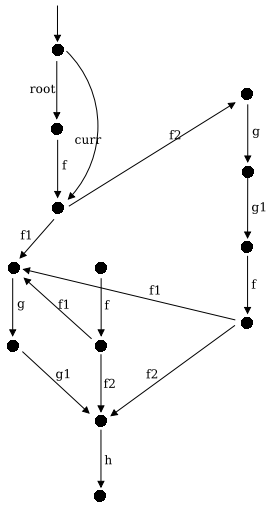


Fig. 4.17: Stato finale del term graph dopo l'applicazione della regola

In conclusione, è possibile realizzare una macchina di Schönhage programmata in modo tale da riconoscere l'eventuale presenza di un particolare redex su un term graph fornito e di applicare di conseguenza la relativa regola realizzando la riduzione del grafo stesso.

Appendice A

Macchine a registri

Le macchine a registri sono macchine astratte simili alle MdT e si distinguono da esse per l'utilizzo dei registri da cui prendono il nome. Questi registri possono contenere un singolo numero intero e sono identificati univocamente tramite indirizzi. A seconda dell'implementazione, si possono identificare quattro diverse tipologie di queste macchine: macchine a contatori, macchine a puntatori, RAM (dall'inglese *random access machines*) e RASP (sigla che indica *random access stored program machine model*). La sezione successiva si limiterà a descrivere brevemente tre di queste, poiché la categoria di macchine a puntatori (comprendente SMM, KU e Linking Automata) è già stata discussa nei capitoli precedenti.

A.1 Macchine a contatori

Rappresentano la categoria più elementare di macchine a registri. L'insieme di istruzioni da esse fornito è solitamente di dimensioni ristrette: nella maggior parte dei casi ognuno dei modelli proposti in letteratura presenta, in aggiunta all'istruzione di halt, una scelta di tre tra le seguenti istruzioni:

- **INC** (r); incrementa il valore memorizzato dal registro r di un'unità;
- **DEC** (r); decrementa il valore memorizzato dal registro r di un'unità;
- **CLR** (r); modifica il valore memorizzato dal registro r assegnandogli valore zero;
- **CPY** (r_i, r_j); copia il valore memorizzato dal registro r_i nel registro r_j;
- **JZ** (r, x); salta all'istruzione x se il valore memorizzato dal registro r è uguale a zero, altrimenti prosegue con l'istruzione successiva;

- **JE** (r_i, r_j, x); salta all'istruzione x se il valore memorizzato dal registro r_i è uguale a quello memorizzato dal registro r_j, altrimenti prosegue con l'istruzione successiva.

Formalmente, una macchina a contatori consiste in tre componenti: un insieme potenzialmente infinito di registri etichettati, un registro per lo stato e una lista di istruzioni a loro volta etichettate. Il registro dello stato contiene l'istruzione che dev'essere eseguita in ogni momento della computazione. Tutti gli altri registri, invece, possono contenere un valore intero positivo. Talvolta viene riservato un registro apposito per utilizzarlo come accumulatore. Le istruzioni sono sequenziali e dipendono dalla specifica implementazione della macchina, sebbene facciano parte dell'insieme già elencato.
Le macchine a contatori sono Turing equivalenti, ma presentano diverse caratteristiche che le rendono poco idonee a simulazioni pratiche, soprattutto dovute al numero potenzialmente infinito dei registri contenenti i valori.

A.2 Macchine ad accesso casuale (RAM)

Come già è stato visto, Schönhage si occupò di dimostrare l'equivalenza in tempo reale delle SMM rispetto $RAM1$ e $RAM0$. Per la definizione delle RAM, l'autore utilizzò come riferimento il modello descritto da Aho, Hopcraft e Ullman [15]. Vengono riportate in seguito le caratteristiche principali di queste macchine, insieme ad alcune spiegazioni e definizioni che possono risultare utili per la comprensione.

Analogamente alle macchine a contatori, delle quali rappresentano una versione più completa, le *macchine ad accesso casuale* o RAM sono modelli di macchine a registro con un singolo accumulatore e una porzione a stati finiti dell'architettura dedicata alle istruzioni. La principale differenza con il precedente gruppo di modelli è la possibilità di servirsi dell'indirizzamento indiretto. La struttura principale di una RAM è divisa nei seguenti elementi:

- un nastro sequenziale di sola lettura per l'input. Esso è composto da una serie di celle, ognuna delle quali contiene un valore intero solitamente negativo. Ogni valore viene letto una sola volta e in seguito a questa azione la testina di lettura del nastro si sposta di un'unità nella direzione di lettura;

- un nastro sequenziale di sola scrittura per l'output, anch'esso diviso in celle inizializzate con il valore nullo. Anche in questo caso, il sistema interagisce con il nastro per mezzo di una testina che scrive un intero in ogni cella, spostandosi sequenzialmente di un'unità per volta;

- un insieme (la cui cardinalità non ha limiti superiori) di registri $r_0, r_1, \ldots$ ognuno dei quali può memorizzare un intero di grandezza arbitraria. Ovviamente bisogna tenere conto del fatto che, volendo mantenere

l'astrazione realistica, la grandezza degli interi dev'essere sufficientemente piccola da essere salvata in una parola della memoria reale. Allo stesso modo la dimensione del programma non può essere eccessivamente grande;

- un programma composto da istruzioni etichettate, non memorizzato nei registri ma in una zona ad esso riservata. A causa di questa sua collocazione il programma non può essere modificato. La natura delle istruzioni ricorda quella utilizzata nelle macchine a contatori, in aggiunta a ulteriori operazioni come *load*, *store*, operazioni aritmetiche, *read* e *write*. Ogni istruzione è composta da due parti: codice e indirizzo (ad esempio LOAD a, oppure JUMP b);

- un registro accumulatore r_0 nel quale avviene la computazione vera e propria

In generale, il programma rappresenta una funzione dal nastro di input al nastro di output. Questa funzione è parziale, poiché non è detto che ogni programma termini su ogni input.

A.3 Macchine RASP

L'ultima categoria di macchine a registri è quella delle macchine $RASP$, ovvero *random access stored program*. Basato sull'architettura di Von Neumann (in opposizione all'architettura Harvard su cui sono strutturate le altre macchine a registri), questo modello sfrutta una CPU e la memoria ad accesso casuale delle RAM per elaborare l'input in output. A differenza delle macchine ad accesso casuale, le $RASP$ memorizzano il programma nei registri utilizzati durante l'esecuzione delle istruzioni. La struttura di questo modello è la seguente:

- Nastri di input e output, con le medesime caratteristiche di quelli già osservati;

- CPU, a sua volta costituita da una unità aritmetico logica, un registro contenente l'istruzione corrente, un puntatore all'istruzione successiva e un accumulatore;

- Memoria ad accesso casuale, potenzialmente infinita e utilizzata per memorizzare e accedere a dati in qualsiasi registro. Essa è collegata sia ai nastri che alla CPU, facendo da tramite tra essi.

L'insieme di istruzioni utilizzate è analogo a quello di un comune linguaggio assembly.

Appendice B

Macchine a Stati Astratti

Gurevich si servì di un particolare modello di macchina a stati astratti per dimostrare la lock-step equivalenza tra esso e le SMM. In questa sezione ne vengono descritte brevemente le caratteristiche per permettere una migliore comprensione della dimostrazione di equivalenza.

B.1 Struttura dati

Una macchina a stati astratti $\mathcal{A}$ è definita come una tupla $\{V, p, A_0\}$ dove V è detto vocabolario, p è un programma e A_0 è lo stato iniziale. Il *vocabolario* V di $\mathcal{A}$ è un insieme finito di nomi di funzioni e nomi di relazioni ciascuna delle quali ha una sua arietà nota. Uno stato di $\mathcal{A}$ è rappresentato tramite un insieme detto *superuniverso* X insieme a una rappresentazione dei nomi delle funzioni del vocabolario. 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). Mentre il nome di una funzione n-aria è interpretato come una funzione da X^n a X, il nome di una relazione è interpretato come una funzione da X a {true, false}. Per quanto riguarda l'algebra booleana, infatti, essa viene espressa basandosi su tre elementi contenuti nel superuniverso X detti *costanti logiche*: true, false e undef. Un *universo* U è una relazione unaria identificata con l'insieme $\{x : U(x)\}$, come ad esempio la costante logica Bool={true, false}. La funzione f dall'universo U all'universo V rappresenta l'operazione unaria che associa un valore $x = f(a) \in V$ ad ogni $a \in U$ e $x =$ undef altrmenti. Ogni macchina a stati astratti contiene l'universo Modes={Initial, Working, Final} nel suo vocabolario. 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.

B.2 Regole

Un programma è una regola di transazione. In seguito vengono elencati alcuni esempi di transazioni, che rappresentano le tipologie di istruzioni delle macchine a stati astratti.

- **update**; scritto nella forma $f(\bar{t}) := t_0$ con $\bar{t} = t_1, \ldots, t_n$. L'effetto di questa regola è quello di modificare il valore di $f(\bar{t})$ in t_0 nello stato successivo;

- **block rule**; espresso nella forma block $R0, \ldots, Rk$ endblock. Vengono eseguite tutte le transazioni $R0, \ldots, Rk$ simultaneamente. Se una coppia di queste regole modifica il valore di uno stesso elemento allora l'update corrispondente si dice *inconsistente* e nessuna delle regole viene eseguita. In caso contrario l'insieme si dice *consistente*;

- **conditional rule**; si presenta nella forma

  if g_0 then R_0
  elseif g_1 then R_1
  $\vdots$
  elseif g_k then R_k
  endif
  Gli elementi $g_0, g_1, \ldots$ detti *sentinelle* vengono valutati sequenzialmente fino al primo g_i valutato true. Dopodiché viene eseguita la transazione identificata con R_i;

- **import rule**; espressa nella forma import v R_0 endimport è utilizzata per creare un nuovo elemento (ad esempio il nodo di un grafo). Solitamente v è un elemento appartenente all'universo Reserve.

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.

Bibliografia

[01] 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, June 1988, 71-82

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

[03] Robert E. Tarjan (1977), Reference machines require non-linear time to maintain disjoint sets, Proc. 9th Annual ACM Symposium on Theory of Computing, Boulder, Colorado March 1977, 18-29

[04] Knuth D. E. (1968), The Art of Computer Programming, vol.1, Addison - Wesley, Reading, MA

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

[06] Scott Dexter, Patrick Doyle and Yuri Gurevich (1997), Gurevich Abstract State Machines and Schönhage Storage Modification Machines, Journal of Universal Computer Science, vol. 3, no. 4 (1997), 279–303

[07] Vladimir A. Uspensky (1992), Kolmogorov and Mathematical Logic, J. Symbolic Logic Volume 57, Issue 2 (1992), 385-412

[08] Vladimir A. Uspensky and Alexei Semenov (1993), Algorithms: Main Ideas and Applications, Kluwer, Dordrecht

[09] 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

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

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

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

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

[14] Vladimir A. Uspensky and Andrei Nikolaevich Kolmogorov (1958), On the definition of an algorithm, Uspekhi Mat. Nauk, 13:4(82)

[15] Andreas Blass and Yuri Gurevich (1994), Evolving Algebras and Linear Time Hierarchy. IFIP World Computer Congress, 383-390

[16] A. V. Aho, J. E. Hopcroft and J. D. Ullman (1974), The Design and Analysis of. Computer Algorithms, Addison-Wesley

[17] D. Grigoryev (1976), Kolmogorov Algorithms are Stronger Than Turing Machines, Investigations on Constructive Mathematics and Mathematical Logic. VIII, Notes of hte Leningrad Branch of Steklov Math. Institute 60, 29-37