UNIVERSITA' DEGLI STUDI DI BARI

FACOLTA' DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN INFORMATICA

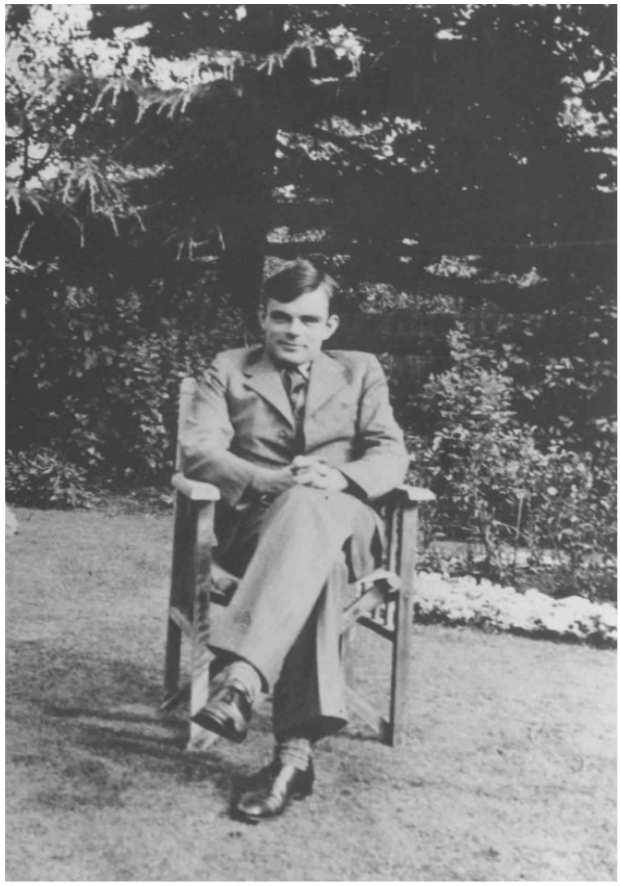
TESI DI LAUREA

Una misura di complessità per linguaggi di programmazione imperativi: σ-measure

Relatore:
Chiar.mo Prof. Emanuele Covino

Laureanda:
Teresa Fasano

ANNO ACCADEMICO 2004 – 2005



Propongo di considerare questa domanda:

“Le macchine sono in grado di pensare?”

Alan Mathison Turing

A Gigi

I N D I C E

CAPITOLO II

CAPITOLO III

INTRODUZIONE

1. Contesto

In ogni settore tecnologico, gli strumenti e le metodologie che quotidianamente vengono utilizzati, e che sono in continua evoluzione per adattarsi a nuove esigenze e nuove applicazioni, si basano su alcuni fondamenti teorici che ne definiscono le proprietà essenziali. Ad esempio, nel caso dell'informatica, lo sviluppo delle tecnologie e delle metodologie che ne guidano l'impegno si fonda su alcuni presupposti teorici la cui conoscenza è indispensabile per chi debba contribuire all'innovazione dei sistemi o debba utilizzare con competenza i sistemi esistenti per realizzare applicazioni complesse. La teoria informatica si basa su principi di tipo logico-matematico ed algebrico usati per caratterizzare le proprietà dei concetti di algoritmo, calcolo e complessità alla base di questa disciplina.

Un ruolo molto importante per lo studio dei fondamenti teorici dell'informatica lo ha assunto la Macchina di Turing poichè associa un'elevata semplicità di struttura e di funzionamento al pieno potere computazionale realizzabile da un dispositivo di calcolo automatico; infatti, questo modello è diventato fondamentale sia per la teoria della calcolabilità sia per la teoria della complessità computazionale. Ci concentreremo sulla teoria della complessità, introducendo il concetto di classe di complessità, intesa come l'insieme dei problemi risolubili con un determinato limite sulle risorse di calcolo. Tuttavia, attraverso la complessità computazionale implicita forniremo una visione alternativa alla descrizione classica delle classi di complessità, poiché questa mira alla valutazione di fattori definiti "impliciti" per classificare i problemi e non più ai "limiti espliciti" relativi alle risorse. Il primo contributo, in questo senso, è stato avanzato da Cobham che indagò sulla difficoltà computazionale intrinseca delle funzioni e introdusse il concetto di indipendenza delle soluzioni dei problemi dagli algoritmi e dai modelli di calcolo, sottolineando la necessità di definire le classi di complessità

computazionale ricorrendo ad opportune classi di funzioni, e non a limiti sulle risorse usate dai modelli di calcolo. Successivamente, Bellantoni & Cook definiscono una nuova caratterizzazione predicativa delle funzioni in Polytime rinunciando alla ricorsione limitata e utilizzando la ricorsione safe, la quale, nonostante riesca a catturare Polytime, lo fa passando attraverso il modello di Turing in modo inefficiente. Infatti, ad esempio, semplici ordinamenti non possono essere descritti con la ricorsione safe, e semplici funzioni (come il minimo) sono calcolate con complessità troppo alta.

Nel 1999 Neil Jones, propone un approccio alternativo che mette in discussione i risultati raggiunti negli anni precedenti; egli, infatti, afferma che l'efficienza dei programmi può dipendere dallo stile di programmazione adottato, e i problemi e le incoerenze incontrate per definire le classi computazionali attraverso i linguaggi funzionali, oggetto di analisi negli anni precedenti, sono la conseguenza dello stile scelto, che può essere sostituito con quello imperativo.

Definiremo, quindi, uno stack-language caratterizzato da stack-program, dove gli stack sono usati come variabili. Dopo aver

introdotto tutte le caratteristiche sintattiche e semantiche degli stack-program, verrà affrontato il problema dell'influenza dei loop annidati sulla complessità computazionale dei programmi; infatti, il metodo proposto si basa sull'assegnazione ad ogni stack-program P di un numero naturale $\mu(P)$, la μ-measure, calcolabile dalla sintassi del programma stesso. Questa misura permette di estrarre le informazioni necessarie per distinguere i programmi con complessità polinomiale da quelli che hanno complessità esponenziale e permette di risalire alla classe della gerarchia di Grzegorczyk in cui collocare le funzioni che tali programmi calcolano. Mostreremo che ogni funzione calcolata da un stack-program di misura 0 ha tempo di esecuzione polinomiale; mentre, nel caso generale, dimostreremo che ogni stack-program con $\mu = n$ può essere simulato da una Macchina di Turing con tempo di esecuzione nella classe ε^{n+2} della gerarchia di Grzegorczyk, e allo stesso modo, ogni Macchina di Turing con tempo di esecuzione in ε^{n+2} può essere simulata da uno stack-program di misura $\mu = n$.

2. Risultato

L'obiettivo di questa tesi è quello di definire un miglioramento per la misura μ: la σ-measure, che permette di calcolare la complessità computazionale degli stack-program, in modo molto più efficiente rispetto alla μ-measure. Infatti, distingueremo i cicli in due tipi: quelli increasing, che determinano un aumento delle dimensioni degli stack coinvolti nel ciclo stesso, e quelli not increasing, che lasciano invariata la dimensione complessiva degli stack. In definitiva, potremo affermare che mentre μ cresce ogni volta che un top circle appare nel corpo di un ciclo, cioè cresce anche se annidiamo delle istruzioni o dei cicli che non modificano lo spazio complessivo e che non comportano una crescita nella complessità della funzione calcolata; σ aumenta, invece, soltanto per i top circle increasing, perciò avremo che $\sigma(P)$ sarà minore di $\mu(P)$, per ogni stack-program P.

Mostreremo anche che, applicando la procedura di riduzione $\rightsquigarrow$ ad uno stack-program P con $\sigma(P) < \mu(P)$, sarà possibile ridurre il

valore della μ-measure, in modo tale da ottenere $\mu(\rightsquigarrow P) = \sigma(\rightsquigarrow P)$. In conclusione, così come è stato dimostrato per la μ-measure, potremo affermare, che ogni stack-program con σ-measure pari ad n può essere simulato da una Macchina di Turing con complessità temporale in ε^{n+2}, e viceversa, una Macchina di Turing con complessità temporale in ε^{n+2} può essere simulata da uno stack-program con σ-measure pari ad n.

CAPITOLO I

1. Modelli di calcolo e classi di complessità computazionale

1.1 Generalità Matematiche

Sia dato un insieme A. Con $x \in A$ indichiamo che l'elemento x appartiene ad A, con $B \subseteq A$ indichiamo che l'insieme B è un sottoinsieme proprio di A. Se A è un insieme costituito da un numero finito *n* di elementi, con |A| indichiamo la sua cardinalità, cioè il numero di elementi che lo compongono (|A| = n.

Definizione 1.1 *L'insieme composto da tutti i sottoinsiemi di un insieme A si dice insieme delle parti di A, e si indica con P(A).*

Si noti che P(A) contiene A e l'insieme vuoto ∅. Se |A| è finita e pari ad n, allora |P(A)| è 2^n. Per questa ragione l'insieme P(A) è indicato con 2^A.

Definizione 1.2 *Due insiemi A e B si dicono uguali, e si scrive A = B, se ogni elemento di A è anche elemento di B e, viceversa, se ogni elemento di B è anche elemento di A, cioè*

$$A = B \Leftrightarrow (A \subseteq B \ \wedge \ B \subseteq A).$$

Nel caso in cui si debbano dimostrare proprietà relative ad insiemi finiti esse possono essere verificate esaustivamente su tutti gli elementi. Nel caso in cui si debbano dimostrare proprietà relative ad insiemi infiniti, come quello dei numeri naturali, tale procedimento è chiaramente non accettabile e si utilizza il principio di induzione matematica che afferma quanto segue:

data una proposizione P(n) definita per un generico numero naturale n, si ha che essa è vera per tutti i naturali se

- P(0) è vera (passo base dell'induzione);
- Per ogni numero naturale k, P(k) vera (ipotesi induttiva) implica P(k+1) vera (passo induttivo).

Il principio di induzione completa afferma quanto segue:

data una proposizione P(n) definita per un generico numero naturale $n \geq n_0$ se:

- $P(n_0)$ è vera (passo base dell'induzione);
- Per ogni numero naturale $k \geq n_0$, P(i) vera per ogni i, $n_0 \leq i \leq k$ (ipotesi induttiva), implica P(k+1) vera (passo induttivo).

I due principi sono equivalenti.

Definizione 1.3 *Il prodotto cartesiano di A e B, denotano con AxB, è l'insieme*

$$C = \{ \langle x , y \rangle \mid x \in A \ \wedge \ y \in B \},$$

cioè C è costituito da tutte le possibili coppie ordinate ove il primo elemento appartiene ad A ed il secondo a B.

Si usa la notazione A^n per indicare il prodotto cartesiano di A con se stesso (Ax …xA).

Definizione 1.4 Una *relazione n-aria R su* $A_1, A_2, ..., A_n$ *è un sottoinsieme del prodotto cartesiano* $A_1 \times A_2 \times ... \times A_n$*:*

$$R \subseteq A_1 \times A_2 \times ... \times A_n .$$

Il generico elemento di una relazione è indicato con il simbolo $\langle a_1, a_2, ..., a_n \rangle \in R$, oppure con il simbolo.

Definizione 1.5 *Una relazione* $R \subseteq A^2$ *si dice* relazione d'ordine *se per ogni x, y, z* $\in A$ *valgono le seguenti proprietà*

1. $\langle x, x \rangle \in R$ *(riflessività);*
2. $\langle x, y \rangle \in R \wedge \langle y, x \rangle \in R \Leftrightarrow x = y$ *(antisimmetria);*
3. $\langle x, y \rangle \in R \wedge \langle y, z \rangle \in R \Leftrightarrow \langle x, z \rangle \in R$ *(transitività).*

Un insieme A su cui è definita una relazione d'ordine è detto *insieme parzialmente ordinato*.

Definizione 1.6 *Una relazione d'ordine* $R \subseteq A^2$ *tale che*

$$\langle a, b \rangle \in A^2 \Leftrightarrow aRb \vee bRa ,$$

dove cioè ogni elemento è in relazione con ogni altro elemento, si dice relazione di ordine totale.

Definizione 1.7 *Una relazione* $R \subseteq A^2$ *si dice relazione d'equivalenza se, per ogni x, y, z* $\in A$*, valgono le seguenti proprietà:*

1. $\langle x, x \rangle \in R$ *(riflessività);*
2. $\langle x, y \rangle \in R \Leftrightarrow \langle y, x \rangle \in R$ *(simmetria) ;*
3. $\langle x, y \rangle \in R \wedge \langle y, z \rangle \in R \Leftrightarrow \langle x, z \rangle \in R$ *(transitività).*

Definizione 1.8 *Data una relazione d'equivalenza* $R \subseteq A^2$, *si dice* <u>*indice di R*</u>**,** *e si denota con ind(R), il numero di elementi di A/R.*

Le operazioni che si possono definire per le relazioni non differiscono molto da quelle note per insiemi generici. Ad esempio l'unione è definita da $R_1 \cup R_2 = \{\langle x, y \rangle | \ \langle x,y\rangle \in R_1 \vee \langle x,y\rangle \in R_2\}$, mentre la complementazione è definita da $\overline{R} = \{\langle x, y \rangle | \ \langle x,y\rangle \notin R\}$.

Definizione 1.9 *Sia R una relazione su* A^2*; si definisce* <u>*chiusura transitiva*</u> *di R, denotato con* R^+*, la relazione*

$$R^+ = \{\langle x, y\rangle \mid \exists\, y_1,...,y_n \in A, \text{ con } n\geq 2,\ y_1 = x, y_n = y, \ni' \langle y_i, y_{i+1}\rangle \in R, i = 1,...,n-1\}.$$

Definizione 1.10 *Sia R una relazione su* A^2*; si definisce* <u>*chiusura transitiva e riflessiva*</u> *di R, denotata con* R^**, la relazione*

$$R^* = R^+ \cup \{\langle x, x\rangle \mid x \in A\}.$$

Definizione 1.11 *Si dice che* $R \subseteq X_1 \times X_2 \times ... \times X_n$ *(n≥2) è una* <u>*relazione funzionale*</u> *tra una (n-1)-pla di elementi e l'n-esimo elemento, se* $\forall \langle x_1,...,x_{n-1}\rangle \in X_1 \times ... \times X_{n-1}$ *esiste al più un elemento* $x_n \in X_n$ *tale che* $\langle x_1,...,x_{n-1}\rangle \in R$.

In tal caso si definisce funzione (o applicazione, o corrispondenza univoca) la legge che all'elemento $\langle x_1,...,x_{n-1}\rangle \in X_1 \times ... \times X_{n-1}$ associa, se esiste, l'unico elemento $x_n \in X_n$ tale che $x_1,...,x_n \in R$. La notazione usata per indicare l'esistenza di una relazione di tipo funzionale tra gli elementi $x_1,...,x_n$ è $f(x_1,...,x_n)=x_n$. In tal caso, diciamo che la funzione f ha arità n-1. Per convenzione si ammette che una funzione possa anche avere arità 0; cioè, un valore costante "a" può essere considerato come il risultato della applicazione di una funzione 0-aria: f()= a. La notazione generalmente usata per indicare tra quali insiemi è realizzata la corrispondenza, cioè per stabilire il ruolo degli insiemi in gioco, è: $f : X_1 \times ... \times X_{n-1} \to X_n$

L'espressione $X_1 \times ... \times X_{n-1} \to X_n$ è detta "tipo" della funzione f^n. Nel caso in cui $X_1 = ... = X_{n-1} = X_n = X$ il tipo della funzione può essere indicato da $X^{n-1} \to X$.

Definizione 1.12 *Data una funzione* $f : X_1 \times ... \times X_{n-1} \to X_n$*, l'insieme* $X_1 \times ... \times X_{n-1}$ *è detto* <u>dominio</u> *della funzione, dom(f), e l'insieme* X_n <u>codominio</u>*, cod(f).*

Definizione 1.13 *Data una funzione* $f : X_1 \times ... \times X_{n-1} \to X_n$ *si chiama* <u>*dominio di definizione*</u> *della funzione f, e si indica con la notazione def(f), il sottoinsieme di dom(f) così definito:*

$$def(f) = \{ \langle x_1,...,x_{n-1} \rangle \in dom(f) \mid \exists x_n \in cod(f) \ f(x_1,...,x_{n-1}) = x_n \}.$$

Il dominio di definizione di f è cioè l'insieme di tutte le (n-1)-ple $x_1,...,x_{n-1}$ per cui $f(x_1,...,x_{n-1})$ è definita.

Definizione 1.14 *Si definisce* <u>*immagine*</u> *delle funzione f, e si indica con la notazione imm(f), il sottoinsieme di* X_n *così definito:*

$$imm(f) = \{ \ x_n \in X_n \mid \langle x_1,...,x_{n-1} \rangle \in \ dom(f) \ \ f(x_1,...,x_{n-1}) = x_n \ \}.$$

L'immagine di f è quindi l'insieme di tutti i valori assunti da f.

Definizione 1.15 *Data una funzione f e considerato un generico elemento* $x_n \in$ *cod(f) si dice* <u>*controimmagine*</u> *(o fibra) di* x_n*, e si indica con* $f^{-1}(x_n)$*, il seguente sottoinsieme del dominio di f:*

$$f^{-1}(x_n) = \{ \langle x_1,...,x_{n-1} \rangle \mid \langle x_1,...,x_{n-1} \rangle \in def(f) \wedge f(x_1,...,x_{n-1}) = x_n \}.$$

La controimmagine di x_n è l'insieme di tutte le (n-1)-ple per cui f assume valore x_n. Si noti che il dominio di definizione di una funzione può o meno coincidere con il dominio.

Definizione 1.16 *Una funzione* $f : X_1 \times ... \times X_{n-1} \to X_n$ *è detta totale se def(f) = dom(f). Nel caso più generale in cui def(f)* $\subseteq$ *dom(f), f è detta funzione parziale.*

Tutte le funzioni sono parziali; mentre, quelle totali sono un caso particolare. Analogamente a quanto accade per il dominio di una funzione possiamo osservare che l'immagine di una funzione può coincidere o no con il codominio relativo.

Definizione 1.17 *Una funzione* $f : X_1 \times ... \times X_{n-1} \to X_n$ *è suriettiva se*

$$imm(f) = cod(f).$$

Definizione 1.18 *Una funzione f si dice iniettiva o uno-ad-uno (1:1) se fa corrispondere ad elementi diversi del dominio di definizione elementi diversi del codominio, come mostrato di seguito*

$$\forall \langle x_1', ..., x_{n-1}' \rangle \in X_1 \times ... \times X_{n-1}, \ \forall \langle x_1'', ..., x_{n-1}'' \rangle \in X_1 \times ... \times X_{n-1},$$

$$\langle x_1', ..., x_{n-1}' \rangle \neq \langle x_1'', ..., x_{n-1}'' \rangle \Leftrightarrow f(x_1', ..., x_{n-1}') \neq f(x_1'', ..., x_{n-1}'').$$

Equivalentemente, possiamo anche notare che, in tal caso,

$$\langle x_1', ..., x_{n-1}' \rangle = \langle x_1'', ..., x_{n-1}'' \rangle \Leftrightarrow f(x_1', ..., x_{n-1}') = f(x_1'', ..., x_{n-1}'').$$

Definizione 1.19 *Una funzione si dice biiettiva o biunivoca se è allo stesso tempo iniettiva e suriettiva.*

Esplicitiamo ora la notazione che esprime l'andamento asintotico di una funzione rispetto ad un'altra.

Definizione 1.20 *Data una funzione f: N→N, la notazione O(f(n)) denota l'insieme delle funzioni:*

$$\{g \mid (\exists c > 0)(\exists n_0 > 0)(\forall n \geq n_0)(g(n) \leq cf(n))\}.$$

Se una funzione g(n) è tale che g(n)$\in O$(f(n)) si ha quindi che asintoticamente, per valori sufficientemente grandi, la funzione assume valori non superiori a quelli assunti dalla funzione f(n), a meno di una costante moltiplicativa prefissata. In tal caso affermiamo che g(n) cresce al più come f(n). Con un abuso di notazione scriveremo anche g(n) = O(f(n)).

Definizione 1.21 *Data una funzione f: N→N, la notazione Ω(f(n)) denota l'insieme delle funzioni:*

$$\{g \mid (\exists c > 0)(\exists n_0 > 0)(\forall n \geq n_0)(g(n) \geq cf(n))\}.$$

Se g(n)$\in\Omega$(f(n)) si dice che g(n) cresce almeno come f(n). Con un abuso di notazione si scrive anche g(n) = Ω(f(n)).

Definizione 1.22 *Data una funzione f: N→N, la notazione θ(f(n)) denota l'insieme delle funzioni:*

$$\{g \mid (\exists c_1 > 0)(\exists c_2 \geq c_1)(\exists n_0 > 0)(\forall n \geq n_0)(c_1 f(n) \leq g(n) \leq c_2 f(n))\}.$$

Se g(n) $\in$θ(f(n)) si dice che g(n) ed f(n) crescono nello stesso modo. Con un abuso di notazione si scrive anche g(n) = θ(f(n)).

Definizione 1.23 *Date due funzioni f, g: N→N, con la notazione g(n) = o(f(n)) indichiamo che:*

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0 .$$

Se g(n) = *o*(f(n)) diciamo che g(n) è un infinitesimo rispetto a f(n) o, in modo equivalente, che f(n) è un infinito rispetto a g(n). La stessa relazione può essere espressa mediante la notazione f(n) = ω(g(n)).

Valgono ovviamente le seguenti proprietà:

f(n) = *O*(g(n)) ⇔ g(n) = Ω(f(n))

f(n) = θ(g(n)) ⇔ g(n) = θ(f(n))

f(n) = θ(g(n)) ⇔ f(n) = *O*(g(n)) ∧ f(n) = Ω(g(n)).

La notazione asintotica è utilizzata in particolare per rappresentare il costo di esecuzione degli algoritmi e la complessità dei problemi.

1.2 Macchine di Turing e Turing Calcolabilità

Le Macchine di Turing (MT) sono il modello di calcolo di riferimento fondamentale sia nell'ambito della teoria della calcolabilità sia in quello della teoria della complessità computazionale. Quando il logico inglese A. M. Turing ha introdotto questo modello di calcolo si poneva l'obiettivo di formalizzare il concetto di calcolo allo scopo di stabilire l'esistenza di metodi algoritmici per il riconoscimento dei teoremi nell'ambito dell'aritmetica. Nonostante ciò accadesse in un periodo in cui i primi elaboratori elettronici non fossero ancora stati progettati, il modello ha assunto un ruolo molto importante per lo studio dei fondamenti teorici dell'informatica perché associa un'elevata semplicità di struttura e di funzionamento al pieno potere computazionale realizzabile da un dispositivo di calcolo automatico.

La MT è definita come un dispositivo operante su stringhe contenute su una memoria esterna (nastro) mediante passi elementari, definiti da un'opportuna funzione di transizione. La MT accede ad un nastro

illimitato diviso in celle contenenti ciascuna un simbolo appartenente ad un dato alfabeto Γ, ampliato con il carattere speciale $\flat$ (cella vuota), e opera su tale nastro tramite una testina, la quale può scorrere su di esso in entrambe le direzioni e può leggere o scrivere caratteri appartenenti all'alfabeto Γ oppure il simbolo $\flat$.

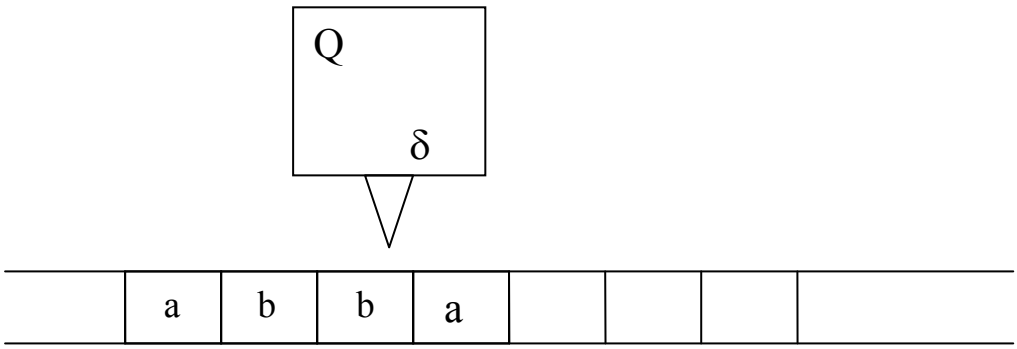


Figura 1.1 Macchina di Turing

Il meccanismo che fa evolvere la computazione della macchina è la funzione di transizione δ la quale, a partire da uno stato e da un carattere osservato sulla cella del nastro, porta la macchina in un altro stato, determinando la scrittura di un carattere su tale cella ed eventualmente lo spostamento della testina stessa.

Definizione 1.24 *Una macchina di Turing deterministica (MTD) è una sestupla* $\boldsymbol{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$**,** *dove* Γ *è l'alfabeto dei simboli di*

nastro, $b \notin \Gamma$ *è un carattere speciale denominato blank, Q è un insieme finito e non vuoto di stati,* $q_0 \in Q$ *è lo stato iniziale,* $F \subseteq Q$ *è l'insieme degli stati finali e* δ *è la funzione (parziale) di transizione, definita come* $\delta : (Q - F) \times (\Gamma \cup \{b\}) \to Q \times (\Gamma \cup \{b\}) \times \{d, s, i\}$*, in cui d, s, i indicano, rispettivamente, lo spostamento a destra, lo spostamento a sinistra e l'assenza di spostamento della testina.*

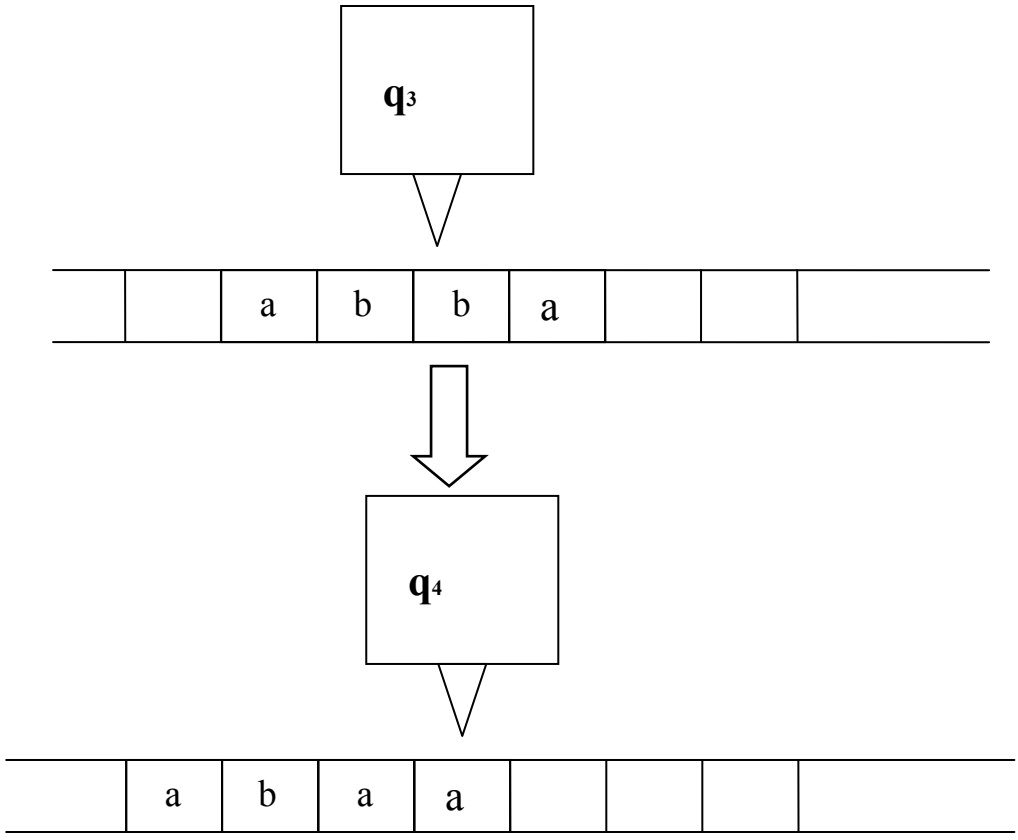


Figura 1.2 Transizione deterministica da δ(q_3,b) = (q_4,a,d)

Nel seguito dato un alfabeto Γ tale che $b \notin \Gamma$, si indicherà con $\overline{\Gamma}$ l'insieme $\Gamma \cup \{b\}$.

Le Macchine di Turing deterministiche possono essere utilizzate sia come strumenti per il calcolo di funzioni, sia per riconoscere o accettare stringhe su un alfabeto di input $\Sigma \subseteq \Gamma$, cioè per stabilire se esse appartengono ad un certo linguaggio oppure no. Si noti che, in realtà, anche questo secondo caso può essere visto come il calcolo di una particolare funzione, la funzione caratteristica del linguaggio, che, definita da $\Sigma^* a \{0,1\}$, assume valore 1 per ogni stringa del linguaggio e 0 altrimenti. Le macchine usate per accettare stringhe sono dette di tipo *riconoscitore*, mentre quelle usate per calcolare funzioni di tipo *trasduttore*. In entrambi i casi, all'inizio del calcolo, solo una porzione finita del nastro contiene simboli diversi da blank che costituiscono l'input del calcolo stesso.

Introduciamo ora i concetti di configurazione e transizione fra configurazioni per le macchine di Turing (deterministiche e non deterministiche).

1.2.1 Configurazioni e transizioni delle Macchine di Turing

La configurazione istantanea (o configurazione) di una MT è l'insieme delle informazioni relative al contenuto del nastro, alla posizione della testina e allo stato corrente.

Una possibile rappresentazione di una configurazione è data da una stringa di lunghezza finita contenente tutti i caratteri presenti sul nastro, oltre a un simbolo speciale corrispondente allo stato interno, posto immediatamente prima del carattere individuato dalla posizione della testina.

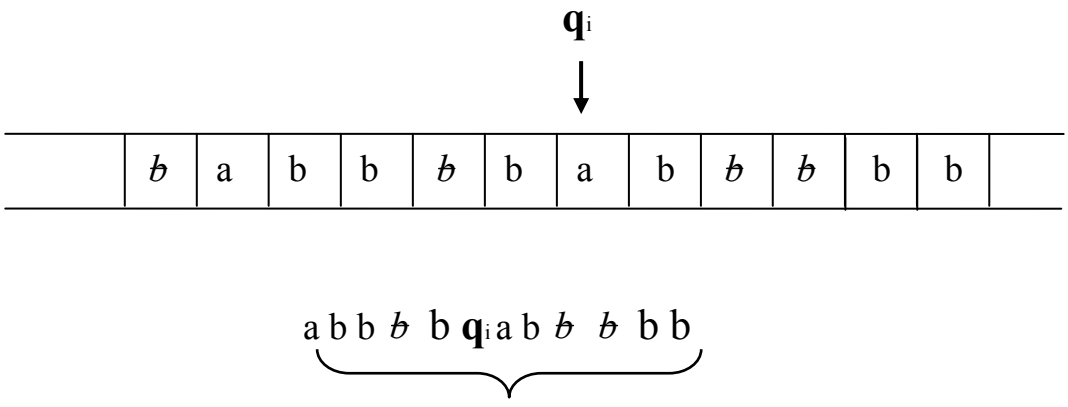


Figura 1.3 Esempio di configurazione istantanea

Definizione 1.25 *Si definisce configurazione istantanea o configurazione di una Macchina di Turing con alfabeto di nastro Γ ed insieme degli stati Q, una stringa c = xqy, con:*

1. $x \in \Gamma\ \overline{\Gamma}^{*} \cup \{\varepsilon\}$;
2. $q \in Q$;
3. $y \in \overline{\Gamma}^{*}\Gamma \cup \{b\}$.

L'interpretazione della stringa xqy è che essa rappresenti il contenuto della sezione non vuota del nastro, che lo stato attuale sia q e che la testina sia posizionata sul primo carattere di y. Nel caso in cui x = ε abbiamo che a sinistra della testina compaiono solo simboli b, mentre se y = b sulla cella attuale a destra della testina compaiono soltanto simboli b.

Nel seguito, per brevità, indichiamo con L_Γ il linguaggio $\Gamma\overline{\Gamma}^{*} \cup \{\varepsilon\}$ delle stringhe che possono comparire a sinistra del simbolo di stato in una configurazione, e con $\Re_\Gamma$ il linguaggio $\overline{\Gamma}^{*}\Gamma \cup \{b\}$ delle stringhe che possono comparire alla sua destra.

Un particolare tipo di configurazione è rappresentato dalla configurazione iniziale. Indichiamo con tale termine una configurazione che, data una qualunque stringa $x \in \Gamma^*$, rappresenti stato e posizione della testina all'inizio di una computazione su input x. Per quanto riguarda lo stato, ricordiamo che per definizione esso dovrà essere q_0, mentre, prima di definire la posizione della testina, è opportuno stabilire una convenzione sulla maniera di specificare l'input $x \in \Gamma^*$ su cui la macchina deve operare. La convenzione universalmente adottata prevede che all'inizio della computazione la macchina abbia il nastro contenente tale input su una sequenza di |x| celle contigue, che tutte le altre celle siano poste a $\not b$ e che la testina sia inizialmente posizionata sul primo carattere di x. Queste considerazioni ci portano alla seguente definizione.

Definizione 1.26 *Una configurazione c = xqy si dice iniziale se* $x = \varepsilon$, $q = q_0$, $y \in \Gamma^+ \cup \{\not b\}$.

Definizione 1.27 *Una configurazione c = xqy, con* $x \in L_\Gamma$, $y \in \Re_\Gamma$ *si dice finale se* $q \in F$.

Quindi una Macchina di Turing si trova in una configurazione finale se il suo stato attuale è uno stato finale, indipendentemente dal contenuto del nastro e della posizione della testina. Chiaramente, una configurazione descrive in modo statico la situazione in cui una macchina di Turing si trova in un certo istante. Per caratterizzare il comportamento "dinamico" di una macchina di Turing, che determina il suo passaggio da configurazione a configurazione, si deve fare riferimento alla relativa funzione di transizione.

La funzione di transizione può essere rappresentata mediante matrici di transizione e grafi di transizione; le colonne della matrice di transizione corrispondono ai caratteri osservabili sotto la testina (elementi di $\bar{\Gamma}$) e le righe ai possibili stati interni della macchina (elementi di Q); mentre gli elementi della matrice (nel caso di Macchina di Turing deterministica) sono triple che rappresentano il nuovo stato, il carattere da scrivere nella cella corrente e lo spostamento della testina.

Nella fig. 1.4 è riportata una possibile matrice di transizione per una MT deterministica con alfabeto di nastro $\Gamma = \{0,1,*,\$\}$ ed insieme

degli stati Q = $\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_F\}$, dove q_F è l'unico stato finale.

	0	1	*	$	$\not{b}$
q_0	(q_1,*,d)	(q_2,$,d)	–	–	(q_F, $\not{b}$,i)
q_1	(q_1,0,d)	(q_1,1,d)	–	–	(q_3, $\not{b}$,d)
q_2	(q_2,0,d)	(q_2,1,d)	–	–	(q_4, $\not{b}$,d)
q_3	(q_3,0,d)	(q_3,1,d)	–	–	(q_5,0,s)
q_4	(q_4,0,d)	(q_4,1,d)	–	–	(q_6,1,s)
q_5	(q_5,0,s)	(q_5,1,s)	(q_0,*,d)	–	(q_5, $\not{b}$,s)
q_6	(q_6,0,s)	(q_6,1,s)	–	(q_0,1,d)	(q_6, $\not{b}$,s)
q_F	–	–	–	–	–

Figura 1.4 Matrice di Transizione

Nella rappresentazione della funzione di transizione mediante grafo i nodi sono etichettati con gli stati e gli archi con una tripla composta dal carattere letto, il carattere scritto e lo spostamento della testina. Più precisamente, nel grafo di transizione esiste un arco dal nodo q_i, al nodo q_j, e tale arco è etichettato con la tripla a, b $\in \overline{\Gamma}$, m $\in$ {s,d,i} se e solo se $\delta(q_i,a) = (q_j,b,m)$.

Data una configurazione c, una applicazione della funzione di transizione δ della macchina di Turing deterministica ***M*** su c permette di ottenere la configurazione successiva c' secondo le modalità seguenti:

1. Se c = xqay, con x $\in L_\Gamma$, y $\in \overline{\Gamma}^*\Gamma$, a $\in \overline{\Gamma}$, e se δ(q,a) = (q',a',d), allora c' = xa'q'y;
2. Se c = xqa, con x $\in L_\Gamma$, a $\in \overline{\Gamma}$, e se δ(q,a) = (q',a',d), allora c' = xa'q$\not b$;
3. Se c = xaqby con xa $\in \Gamma\,\overline{\Gamma}^*$, y $\in \overline{\Gamma}^*\Gamma \cup \{\not b\}$, b $\in \overline{\Gamma}$, e se δ(q,b) = (q',b',s), allora c' = xq'ab'y;
4. Se c = qby, con y $\in \overline{\Gamma}^*\Gamma \cup \{\not b\}$, b $\in \overline{\Gamma}$, e se δ(q,b) = (q',b',s), allora c' = q'bb'y;
5. Se c = xqay, con x $\in L_\Gamma$, β$\in \overline{\Gamma}^*\Gamma \cup \{\not b\}$, a $\in \overline{\Gamma}$, δ(q,a) = (q',a',i), allora c' = xq'a'y.

La relazione tra c e c' (detta relazione di transizione) è indicata per mezzo della notazione $c \vdash_{\boldsymbol{M}} c'$.

In fig. 1.5 è riportato il grafo di transizione corrispondente alla matrice di fig. 1.4.

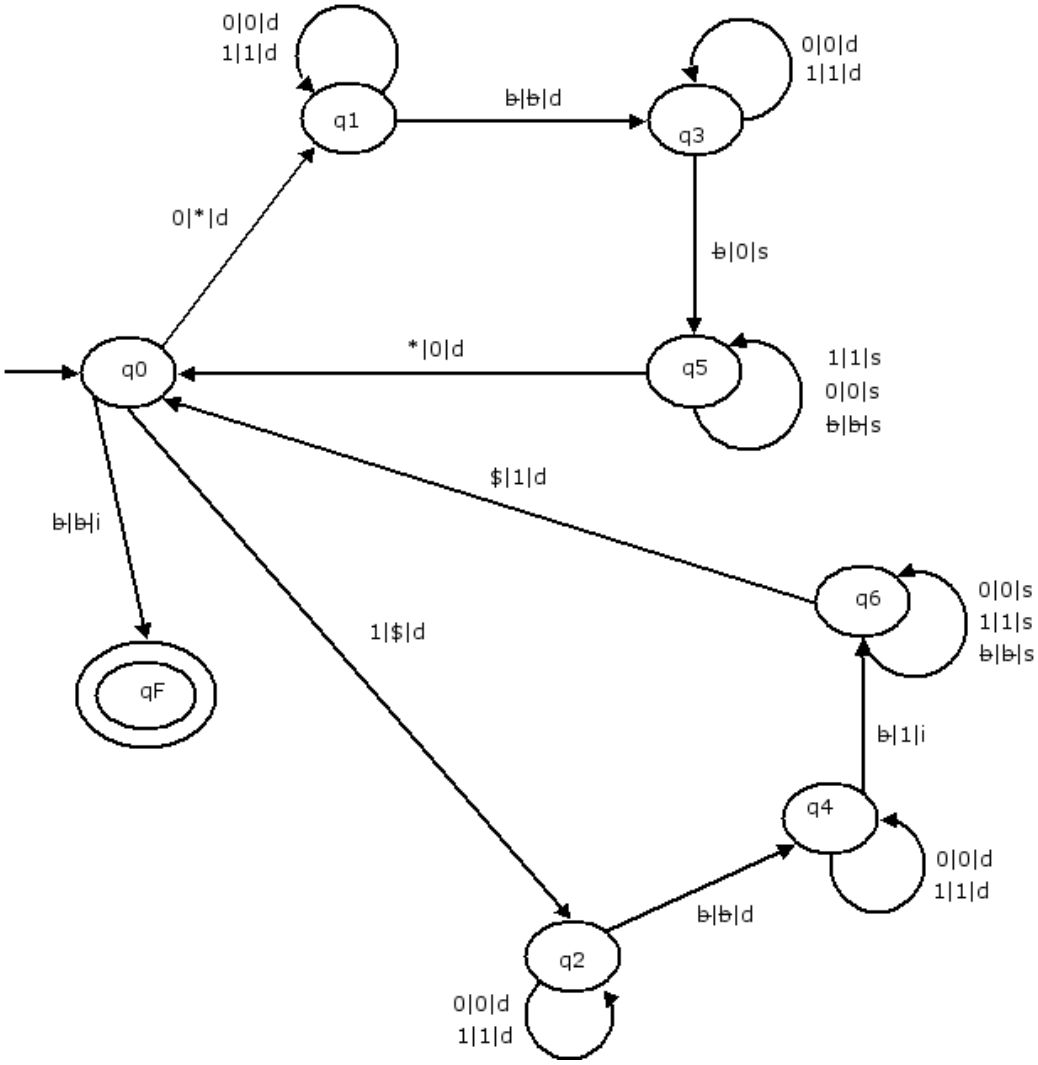


Figura 1.5 Grafo di transizione corrispondente alla matrice in figura 1.4

Le regole di transizione possono essere interpretate come regole di riscrittura del nastro.

1.2.2 Riconoscimento di linguaggi e calcolo di funzioni con MT deterministiche

Le MT deterministiche rappresentano uno strumento teorico potente per affrontare in maniera adeguata il problema del riconoscimento di linguaggi; quindi, si parla delle MT come di dispositivi riconoscitori. Si noti che una MT può essere vista come un dispositivo che classifica le stringhe in Σ^* in funzione del tipo di computazione indotto.

Definizione 1.28 *Data una macchina di Turing deterministica* ***M*** *con alfabeto* Γ *e stato* q_0, *si definisce computazione di accettazione una computazione che termina in una configurazione finale (di accettazione), e si definisce computazione di rifiuto una computazione massimale che si conclude in una configurazione non finale. Dato un alfabeto di input* $\Sigma \subseteq \Gamma$, *una stringa* $x \in \Sigma^*$ *è accettata (rifiuta) da M se esiste una computazione di accettazione (di rifiuto) di* ***M*** *con* $c_o = q_0 x$.

Vi è anche la possibilità che per una MT non esista alcuna computazione massimale con $c_0 = q_0 x$, cioè può accadere che la computazione di ***M*** su input x non termini. Si osservi inoltre che, mentre per ipotesi è possibile verificare in tempo finito se ***M*** accetta o rifiuta una stringa x, la verifica di questo terzo caso non presenta soluzione, poiché, dopo una qualunque computazione non massimale di lunghezza finita, non possiamo sapere se tale computazione terminerà in un qualche istante futuro o meno. In effetti, per risolvere tale problema sarebbe necessario avere a disposizione una diversa macchina di Turing ***M'*** capace di determinare in un tempo finito, date ***M*** e x, se la computazione eseguita da ***M*** su input x termina o no. Questo problema, detto problema di terminazione, non è risolubile, nel senso che non esiste alcuna macchina di Turing con questa capacità.

Poiché una computazione di una macchina di Turing può terminare oppure no, appare necessario fornire alcune precisazioni che chiariscano l'ambito e il modo con cui un linguaggio può essere riconosciuto.

Definizione 1.29 *Sia* $\boldsymbol{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *una macchina di Turing deterministica. Diciamo che M riconosce (decide) un linguaggio* $L \in \Sigma^*$ *(dove* $\Sigma \subseteq \Gamma$*) se e solo se per ogni* $x \in \Sigma^*$ *esiste una computazione massimale* $q_0 x \overset{*}{\underset{\boldsymbol{M}}{\vdash}} wqz$*, con* $w \in \Gamma \overline{\Gamma}^* \cup \{\varepsilon\}$*,* $z \in \overline{\Gamma}^* \Gamma \cup \{\flat\}$*, e dove* $q \in F$ *se e solo se* $x \in L$ *.*

La definizione precedente non esprime alcuna condizione per il caso $x \notin \Sigma^*$: è facile però osservare che potrebbe essere introdotta una Macchina di Turing ***M'*** che verifichi inizialmente se la condizione $x \in \Sigma^*$ è vera, ed operi poi nel modo seguente :

- Se $x \in \Sigma^*$, si comporta come ***M***;
- Altrimenti, se individua in x l'occorrenza di un qualche carattere in $\Gamma - \Sigma$, termina la computazione in un qualche stato non finale.

Definizione 1.30 *Sia* $\boldsymbol{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *una macchina di Turing deterministica. Diciamo che* ***M*** *accetta un linguaggio* $L \in \Sigma^*$ *(dove* $\Sigma \subseteq \Gamma$*) se e solo se* $L = \{ x \in \Sigma^* \mid q_0 x \overset{*}{\underset{\boldsymbol{M}}{\vdash}} wqz \}$*, con* $w \in \Gamma \overline{\Gamma}^* \cup \{\varepsilon\}$*,* $z \in \overline{\Gamma}^* \Gamma \cup \{\flat\}$*, e* $q \in F$ *.*

Definizione 1.31 *Dato un trasduttore* $\boldsymbol{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *ed una funzione f:* $\Sigma^* \to \Sigma^* (\Sigma \subseteq \Gamma)$, ***M*** *calcola la funzione f se e solo se per ogni* $x \in \Gamma^*$:

1. *se* $x \in \Sigma^*$ *e f(x) = y allora* $q_0 x \overset{*}{\underset{M}{\vdash}} x \flat qy$, *con* $q \in F$;
2. *se* $x \notin \Sigma^*$ *oppure se* $x \in \Sigma^*$ e f(x) *non è definita, allora, assumendo la configurazione iniziale* $q_0 x$, *o non esistono computazioni massimali o esistono computazioni massimali che non terminano in uno stato finale.*

Definiamo le nozioni di linguaggio riconosciuto da una MT e di funzione calcolata da una MT.

Definizione 1.32 *Un linguaggio è detto decidibile secondo Turing (T-decidibile) se esiste una macchina di Turing che lo riconosce.*

Definizione 1.33 *Un linguaggio è detto semidecidibile secondo Turing (T-semidecidibile) se esiste una macchina di Turing che lo accetta.*

Definizione 1.34 *Una funzione è detta calcolabile secondo Turing (T-calcolabile) se esiste una macchina di Turing che la calcola.*

1.2.3 Teoria della Complessità

L'obiettivo della teoria della complessità è la caratterizzazione di insiemi di problemi, cioè di classi di complessità, in funzione delle risorse di calcolo necessarie alla loro risoluzione.

Indichiamo con $\overset{\wedge}{t_A}(x)$ il tempo di esecuzione dell'algoritmo A su input x, inteso come numero di passi elementari effettuati. Il tempo di esecuzione nel caso peggiore (worst case) di A sarà il tempo utilizzato da A sull'istanza più difficile e sarà espresso come $t_A(n) = \max\left\{\overset{\wedge}{t_A}(x) \mid \forall : \mid x \mid \leq n\right\}$.

Allo stesso modo, sia $\overset{\wedge}{s_A}(x)$ lo spazio di memoria utilizzato dall'algoritmo A su input x. Lo spazio di esecuzione nel caso peggiore di A, definito come lo spazio utilizzato da A sull'istanza peggiore, è dato da $s_A(n) = \max\left\{\overset{\wedge}{s_A}(x) \mid \forall : \mid x \mid \leq n\right\}$.

Nel caso della complessità spaziale si farà riferimento allo spazio di lavoro addizionale, cioè alla quantità di celle di nastro utilizzate per eseguire la computazione, non considerando lo spazio richiesto per la descrizione iniziale dell'istanza. Assumiamo che la descrizione dell'istanza sia rappresentata su una memoria di sola lettura (read-only) la cui dimensione non sia considerata nella valutazione della complessità spaziale, e a cui sia possibile accedere in modo one-way, muovendosi cioè dall'inizio alla fine, in modo tale che la descrizione dell'input possa essere letta una sola volta.
Data una istanza di un problema, la lunghezza cui si fa riferimento è definita come il numero di caratteri necessari per descrivere tale istanza nell'ambito di un qualche metodo di codifica. In tal modo sia la lunghezza associata ad una istanza che la complessità di un algoritmo saranno dipendenti dal metodo di codifica adottato.

Definiamo prima l'accettazione di una stringa da parte di una data MT (deterministica o non deterministica), in un numero limitato di passi, e poi la condizione di rifiuto di una stringa da parte di una MT deterministica.

Definizione 1.35 *Dato un alfabeto Σ, una macchina di Turing (deterministica o non deterministica)* $\boldsymbol{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *con* $(\Sigma \subseteq \Gamma)$ *ed un intero* $\tau > 0$, *una stringa* $x \in \Sigma^*$ *è* <u>*accettata*</u> *da* $\boldsymbol{M}$ *in* τ *passi se esiste una computazione* $q_0 x \vdash_{\boldsymbol{M}} c$ *composta da r passi, con c configurazione finale, e* $r \leq \tau$.

Definizione 1.36 *Dato un alfabeto* Σ, *una macchina di Turing deterministica* $\boldsymbol{M_D} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *con* $(\Sigma \subseteq \Gamma)$ *ed un* intero τ >0, *una stringa* $x \in \Sigma^*$ *è* <u>*rifiutata*</u> *da* $\boldsymbol{M_D}$ *in* τ *passi se esiste una computazione* $q_0 x \vdash_{\boldsymbol{M}} c$ *composta da* r *passi, con* c *configurazione massimale e non finale, e* $r \leq \tau$.

Introduciamo la proprietà di accettabilità di un linguaggio in tempo limitato da parte di una MT (deterministica o non deterministica).

Definizione 1.37 *Dato un alfabeto Σ, una macchina di Turing (deterministica o non deterministica)* $\boldsymbol{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *con* $(\Sigma \subseteq \Gamma)$ *e una funzione t: N→N, diciamo che* $\boldsymbol{M}$ <u>*accetta*</u> $L \subseteq \Sigma^*$ <u>*in tempo t(n)*</u> *se, per ogni* $x \in \Sigma^*$, $\boldsymbol{M}$ *accetta x in tempo al più t(|x|) se x* $\in L$, *mentre, se* $x \notin L$, *allora* $\boldsymbol{M}$ *non la accetta*.

Definiamo la proprietà di decidibilità in un numero di passi limitato di un linguaggio da parte di una MT deterministica.

Definizione 1.38 *Dato un alfabeto Σ, una macchina di Turing deterministica* $M_D = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *con* $(\Sigma \subseteq \Gamma)$ *e una funzione t: N→N, diciamo che* M_D *accetta* $L \subseteq \Sigma^*$ *in tempo t(n) se, per ogni* $x \in \Sigma^*$, M_D *accetta x in tempo al più t(|x|) se* $x \in L$ *e rifiuta x, ancora in tempo al più t(|x|), se* $x \notin L$.

Quindi, un linguaggio L è accettabile in tempo deterministico t(n) se esiste una M_D che accetta L in tempo t(n), che è accettabile in tempo non deterministico t(n) se esiste una M_{ND} che accetta L in tempo t(n), e che è decidibile in tempo t(n) se esiste una M_D che decide L in tempo t(n).

Per la complessità spaziale, faremo riferimento alle celle del nastro (o nastri) di lavoro accedute durante la computazione. Assumeremo che la stringa di input sia contenuta in un nastro aggiuntivo one way, su cui sia possibile muovere la testina soltanto verso destra; non terremo conto delle celle usate su tale nastro per la rappresentazione dell'input, ma conteremo le celle utilizzate sugli altri nastri.

Definizione 1.39 *Dato un alfabeto Σ, una macchina di Turing (deterministica o non deterministica)* $\boldsymbol{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *con* $(\Sigma \subseteq \Gamma)$ *ed un intero* $\sigma > 0$*, una stringa* $x \in \Sigma^*$ *è <u>accettata</u> da* $\boldsymbol{M}$ *<u>in spazio</u>* σ *se esiste una computazione* $q_0 \# x \vdash_{\boldsymbol{M}} c$*, nel corso della quale sono accedute v celle di nastro, con* c *configurazione finale, e* $v \leq \sigma$.

Definizione 1.40 *Dato un alfabeto Σ, una macchina di Turing deterministica* $\boldsymbol{M_D} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *(con* $\Sigma \subseteq \Gamma$*) ed un intero* $\sigma > 0$*, una stringa* $x \in \Sigma^*$ *è <u>rifiutata</u> da* $\boldsymbol{M_D}$ *<u>in spazio</u>* σ *se esiste una computazione* $q_0 \# x \vdash_{\boldsymbol{M}} c$*, nel corso della quale sono accedute v celle, con c configurazione massimale e non finale, e* $\sigma \leq \tau$.

Definizione 1.41 *Dato un alfabeto Σ, una macchina di Turing (deterministica o non deterministica)* $\boldsymbol{M} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *(con* $\Sigma \subseteq \Gamma$*) e una funzione s: N→N, diciamo che* $\boldsymbol{M}$ *<u>accetta</u>* $L \subseteq \Sigma^*$ *<u>in spazio s(n)</u> se, per ogni* $x \in \Sigma^*$*,* $\boldsymbol{M}$ *accetta x in spazio s(|x|) se* $x \in L$*, mentre, se* $x \notin L$*, allora* $\boldsymbol{M}$ *non la accetta.*

Definizione 1.42 *Dato un alfabeto Σ, una macchina di Turing deterministica* $\boldsymbol{M_D} = \langle \Gamma, \flat, Q, q_0, F, \delta \rangle$ *(con* $\Sigma \subseteq \Gamma$*) e una funzione s: N→N, diciamo che* $\boldsymbol{M_D}$ *riconosce* $L \subseteq \Sigma^*$ *in spazio s(n) se, per ogni* $x \in \Sigma^*$, $\boldsymbol{M_D}$ *accetta x in spazio s(|x|) se* $x \in L$ *e rifiuta x,* (*ancora in spazio s(|x|)), se* $x \notin L$.

Un linguaggio L è accettabile in spazio deterministico s(n) se esiste una $\boldsymbol{M_D}$ che accetta L in spazio s(n); è accettabile in spazio non deterministico s(n) se una $\boldsymbol{M_{ND}}$ accetta L in spazio s(n); è decidibile in spazio s(n) se esiste una $\boldsymbol{M_D}$ che decide L in spazio s(n).

L'obiettivo della teoria della complessità è quello di catalogare l'insieme di tutti i problemi in funzione della quantità di risorse di calcolo necessarie per la relativa risoluzione. A tale scopo, risulta di primaria rilevanza il concetto di classe di complessità intesa come l'insieme dei problemi (funzioni) risolubili (calcolabili) con un determinato limite sulle risorse di calcolo (tipicamente spazio o tempo). La definizione di molte importanti classi di complessità fa riferimento a problemi di decisione e, quindi, tali classi sono tipicamente definite in termini di riconoscimento di linguaggi.

Data una funzione f: N→N, è possibile definire le classi seguenti:

- DTIME(f(n)) è l'insieme dei linguaggi decisi da una MT deterministica ad un nastro in tempo *O*f(n);
- DSPACE(f(n)) è la classe dei linguaggi decisi da una $\boldsymbol{M_D}$ ad un nastro (più un nastro one way di input) in spazio *O*f(n);
- NTIME(f(n)) è l'insieme dei linguaggi accettati da una MT non deterministica ad un nastro in tempo *O*f(n);
- NSPACE(f(n)) è l'insieme dei linguaggi accettati da una $\boldsymbol{M_{ND}}$ ad un nastro (più un nastro one way di input) in spazio *O*f(n);

La classe di complessità P è l'insieme dei linguaggi decidibili da una $\boldsymbol{M_D}$ ad un nastro in tempo polinomiale; mentre la classe P_K è l'insieme dei linguaggi decidibili da una $\boldsymbol{M_D}$ a k nastri in tempo polinomiale. Ogni $\boldsymbol{M_K}$ operante in tempo polinomiale p(n) può essere simulata da una MT ad un nastro in tempo $O(p(n)^2)$. Dato che un linguaggio deciso da una MT ad un nastro in tempo t(n) è decidibile da una $\boldsymbol{M_K}$ nello stesso tempo, si ha che $P=P_K$: cioè, P è l'insieme dei problemi decidibili in tempo polinomiale da una $\boldsymbol{M_D}$, indipendentemente dal numero di nastri.

È sufficiente disporre di una classificazione, limitata a stabilire se, il problema è risolubile in tempo polinomiale o esponenziale. Quindi, definiamo le seguenti classi di complessità:

1) dei linguaggi decidibili da una $\boldsymbol{M_D}$ in tempo proporzionale ad un polinomio nella dimensione dell'input: $P = \cup_{k=0}^{\infty} DTIME(n^k)$;
2) dei linguaggi decidibili da una $\boldsymbol{M_D}$ in spazio proporzionale ad un polinomio: $PSPACE = \cup_{k=0}^{\infty} DSPACE(n^k)$;
3) dei linguaggi decidibili da una $\boldsymbol{M_D}$ in spazio proporzionale al logaritmo: $PSPACE = DSPACE(\log n)$;
4) dei linguaggi decidibili da una $\boldsymbol{M_D}$ in spazio proporzionale ad un esponenziale: $EXPTIME = \cup_{k=0}^{\infty} DTIME(2^{nk})$;
5) dei linguaggi accettabili da una $\boldsymbol{M_{ND}}$ in tempo proporzionale ad un polinomio: $NP = \cup_{k=0}^{\infty} NTIME(n^k)$;
6) dei linguaggi accettabili da una $\boldsymbol{M_{ND}}$ in spazio proporzionale ad un polinomio: $NPSPACE = \cup_{k=0}^{\infty} NSPACE(n^k)$;
7) dei linguaggi accettabili da una $\boldsymbol{M_{ND}}$ in tempo proporzionale ad un esponenziale nella: $NEXPTIME = \cup_{k=0}^{\infty} NTIME(2^{nk})$.

1.2.4 Trattabilità e Intrattabilità dei problemi

L'analisi della complessità dei problemi individua per quali di questi sia possibile determinare le soluzioni in modo efficiente, e per quali, al contrario, sia invece necessario usare algoritmi di costo esponenziale, o super-polinomiale. I problemi del primo tipo sono detti trattabili e quelli del secondo tipo intrattabili.

Consideriamo alcune classi di complessità più legate alla determinazione della trattabilità o intrattabilità computazionale: le classi P, NP e PSPACE. Anche se è possibile mostrare le relazioni di inclusione tra tali classi ($P \subseteq NP \subseteq PSPACE$), tuttavia, per ogni coppia di tali classi, il problema di determinare se l'inclusione è propria o meno, o quello di determinare se esse coincidono o no, è un problema aperto. Il più importante di tali problemi si chiede se la classe P dei problemi decidibili in tempo polinomiale mediante una M_D e la classe NP dei problemi accettabili in tempo polinomiale da una macchina di M_{ND} coincidono o meno.

La classe P viene considerata come una caratterizzazione formale dei problemi trattabili. Ciò è dovuto alle seguenti osservazioni:

1) La crescita delle funzioni non polinomiali (esponenziali) è tale che i loro valori diventano rapidamente troppo grandi.
2) Per n sufficientemente piccolo, un algoritmo polinomiale può eseguire più passi di uno esponenziale, implicando quindi che eventualmente un algoritmo esponenziale potrebbe fornire migliori prestazioni di uno polinomiale su tutte le istanze trattate "in pratica". Tale fenomeno è piuttosto raro, poiché gli algoritmi polinomiali individuati hanno una complessità limitata da un polinomio di grado basso, con costanti moltiplicative piccole; quindi, un algoritmo polinomiale richiede molto meno tempo di calcolo di uno non polinomiale.
3) Gli algoritmi polinomiali sono gli unici per i quali eventuali avanzamenti tecnologici comportano la possibilità di trattare istanze di dimensioni significativamente maggiori.

È un problema aperto se tutti i problemi appartenenti alla classe P abbiano effettivamente un livello di complessità equivalente.

2. Gerarchie di funzioni

Questo paragrafo si propone di introdurre e definire alcune delle funzioni più comunemente usate nella teoria dei numeri elementari, ossia le funzioni ottenute dall'applicazione degli schemi di composizione, somma e prodotto limitati e ricorsione primitiva. L'introduzione di tali definizioni ha come obiettivo la caratterizzazione di due classi di funzioni: le funzioni elementari e le funzioni ricorsive primitive. Il lettore deve rifersi [25] per le definizioni ed i teoremi relativi a questo paragrafo.

2.1 Funzioni elementari

Definizione 2.1 *Si dicono funzioni base le funzioni:*

1. Zero $o(x) = 0$;
2. Successore $s(x) = x + 1$;
3. Proiezione $i_j^n(x_1, x_2, ..., x_n) = x_j \quad 1 \leq j \leq n \ \ e \ \ n = 1,2,...$

Definizione 2.2 *Sia data una funzione f ad m argomenti, e* $g_1,...,g_m$ *funzioni ad n argomenti, definiamo la funzione* $Cfg_1,...,g_m$ *ottenuta per composizione tale che:*

$$Cfg_1,...,g_m\ (x_1,x_2,...,x_n) = f(g_1\ (x_1,...,x_n),...,g_m\ (x_1,...,x_n))\,.$$

Non è indispensabile che ognuna delle $g_1,...,g_m$ abbia lo stesso numero di argomenti dal momento che si può ricorrere alla proiezione per "riempire" i vuoti. Per esempio la funzione:

$$k(x,y,z,z_1) = f(g(x,z),h(y),j(x,y,z_1))\,.$$

Si può definire come segue, sia:

$$g_1(x,y,z,z_1) = Cgi_1^4 gi_3^4(x,y,z,z_1) = g(x,z)$$

$$h_1(x,y,z,z_1) = Chi_2^4(x,y,z,z_1) = h(y)$$

$$j_1(x,y,z,z_1) = Cji_1^4 i_2^4 i_4^4(x,y,z,z_1) = j(x,y,z_1)$$

e così

$$k(x,y,z,z_1) = Cfg_1 h_1 j_1(x,y,z,z_1)\,.$$

L'Addizione può essere definita mediante lo schema di composizione applicato alla funzione successore

$$x + y = x + 1 + 1 + \ldots + 1$$

con y applicazioni della funzione successore. Così la moltiplicazione si ottiene dall'addizione con

$$x \cdot y = x + x + \ldots + x$$

con y-1 applicazioni, se $y > 0$.

Definiamo la funzione Predecessore $x\dot{-}1$ come il numero che precede x nel generico ordinamento dei numeri casuali, e se $x = 0$ il risultato sarà 0.

$$(x+1)\dot{-}1 = x,$$

$$(x\dot{-}1)+1 = x, \text{ se } x \neq 0,$$

$$(0\dot{-}1)+1 = 1.$$

L'applicazione ripetuta genera la funzione $x\dot{-}y = (\ldots((x\dot{-}1)\dot{-}1)\ldots)\dot{-}1)$, con y applicazioni della funzione predecessore. Ma solo se:

$$x > y \Rightarrow x\dot{-}y = x - y \quad \text{altrimenti} \quad x\dot{-}y = 0.$$

La funzione $x\dot{-}y$ è chiamata funzione sottrazione modificata.

Definizione 2.3 *Se f ha n+1 argomenti ed è stata precedentemente introdotta, definiamo la somma limitata come segue:*

$$\sum_{t<x} f(t, x_1, ..., x_n) = f(0, x_1, ..., x_n) + f(1, x_1, ..., x_n) + ... + f(x, x_1, ..., x_n),$$

e il prodotto limitato:

$$\prod_{t<x} f(t, x_1, ..., x_n) = f(0, x_1, ..., x_n) \cdot f(1, x_1, ..., x_n) \cdot ... \cdot f(x, x_1, ..., x_n).$$

Definizione 2.4 Le **funzioni elementari** ε′

ε′ è la classe di funzioni contenente il successore, la proiezione, lo zero, l'addizione, la moltiplicazione, la funzione sottrazione moltiplicata ed è chiusa rispetto alla composizione, somma e prodotto limitati. Se l'operazione di prodotto limitato non è inclusa allora si parlerà di classe di funzioni elementari inferiori.

Definizione 2.5 *Una relazione P appartiene alla classe di funzioni ε′ se c'è una funzione p in ε′ che soddisfa la proprietà:*

$$p(x_1, x_n) = 0 \Leftrightarrow P(x_1, ..., x_n).$$

Per esempio la relazione:

"x ≤ y" è rappresentato in ε′ con $x \dot{-} y = 0$

"x ≤ y" è rappresentato in ε′ con $(x+1) \dot{-} y = 0$

Introduciamo alcune generiche funzioni elementari.

Definizione 2.6 $a(x) = 1\dot{-}(1\dot{-}x)$,

$$|x, y| = (x\dot{-}y)+(y\dot{-}x)$$

$$\bar{a}(x, y) = (|x, y|)$$

Consideriamo lo Schema di Minimalizzazione, ricorrendo alla notazione:

$$(\mu t \leq x)\ [f(t) = 0]$$

che rappresenta il minimo $t \leq x$ per cui $f(t) = 0$.

Anche le operazioni logiche possono essere definite in ε′ nel seguente modo. Se P e Q sono relazioni di ε′ rappresentate con le funzioni $\bar{p}$ e $\bar{q}$, allora tutte le relazioni che si ottengono utilizzando gli operatori logici possono essere rappresentate in ε′ da funzioni costruite a partire da $\bar{p}$ e $\bar{q}$ usando le operazioni elementari.

Si hanno:

P & Q (P and Q) è rappresentata con $\bar{p} + \bar{q} = 0$

P $\vee$ Q (P or Q) è rappresentata con $\bar{p} \cdot \bar{q} = 0$

$\neg$P (not P) è rappresentata con $1\dot{-}\bar{p} = 0$

P $\Rightarrow$ Q (P implies Q) è rappresentata con $(1\dot{-}\bar{p}) \cdot \bar{q} = 0$

Il modus ponens P, P $\Rightarrow$ Q $\vdash$Q è rimpiazzato in ε′ dalla procedura: $\bar{p} = 0$, $(1 \dot{-} \bar{p}) \cdot \bar{q} = 0$ dalla quale si ottiene $\bar{q} = 0$.

Così tutte le tautologie del calcolo preposizionale possono essere rappresentate mediante delle apposite equazioni ancora vere in ε′.

In modo simile, anche i quantificatori limitati possono essere definiti in ε′ come segue:

sia F una relazione elementare, usiamo la notazione $(\forall t \leq x)$ [f(t)] per indicare che F(t) è vera per ogni $t \leq x$ ed $(\exists\ t \leq x)$ [f(t)] per indicare che F(t) è vera per una sola $t \leq x$. Se F è rappresentata in ε′ da f si ha che:

$(\forall t \leq x)$ [f(t)] è rappresentata con $\sum_{t<x} f(t) = 0$ in ε′,

e

$(\exists t \leq x)$ [f(t)] è rappresentata con $\prod_{t<x} f(t) = 0$ in ε′.

Così come le tautologie del calcolo preposizionale sono vere, anche quelle del calcolo dei predicati sono valide in ε′ le quali coinvolgono i quantificatori universali ed esistenziali.

2.2 Funzioni ricorsive primitive

Definizione 2.8 *Sia g una funzione ad n argomenti, h ad n+2 argomenti e J a n+1 argomenti. Definiamo f ad n+1 argomenti per ricorsione primitiva (p.r.) come segue:*

$$\begin{cases} f(0, x_1, ..., x_n) = g(x_1, ..., x_n), \\ f(y+1, x_1, ..., x_n) = h(y, f(y, x_1, ..., x_n), x_1, ..., x_n) \end{cases}$$

Se f soddisfa anche: $f(y, x_1, ..., x_n) \leq J(y, x_1, ..., x_n)$ diremo che f è definita per ricorsione limitata (p.l.).

Questo nuovo schema, permette di introdurre la nuova classe PR.

Definizione 2.9 *La classe delle funzioni ricorsive PR è la classe contenente lo Zero, Successore, Proiezione, è chiusa rispetto alla composizione e alla ricorsione primitiva.*

Tutte le funzioni introdotte precedentemente e ottenute mediante lo schema di composizione sono ricorsive primitive.

L'addizione:

$$\begin{cases} x + 0 = x, \\ x + (y + 1) = (x + y) + 1; \end{cases}$$

la moltiplicazione:

$$\begin{cases} x \cdot 0 = 0, \\ x \cdot (y + 1) = (x + y) + 1; \end{cases}$$

il predecessore:

$$\begin{cases} \overset{\bullet}{x - y} = 0, \\ (x+1)\overset{\bullet}{-}1 = x. \end{cases}$$

e la somma limitata:

$$\begin{cases} \sum_{t \le 0} f(t) = f(0), \\ \sum_{t \le x+1} f(t) = \sum_{t \le x} f(t) + f(x+1). \end{cases}$$

Questa corrispondenza tra alcune componenti di PR ed ε′ porta a dedurre una caratteristica della classe PR ossia il fatto che include ε′ (ε′$\subset$ PR) come appare in fig. 2.1. Ma PR è una classe molto più vasta, infatti racchiude non solo le funzioni ricorsive elementari ma anche le non elementari. A tal proposito introduciamo il teorema 2.10, che definisce la condizione necessaria e sufficiente affinché una funzione ricorsiva primitiva sia anche elementare e quindi possa essere classificata come appartenente ad entrambe le classi.

Teorema 2.10 *Se f è una funzione ricorsiva primitiva e se f è limitata da una funzione elementare g e le funzioni ausiliarie sono elementari, allora f è elementare.*

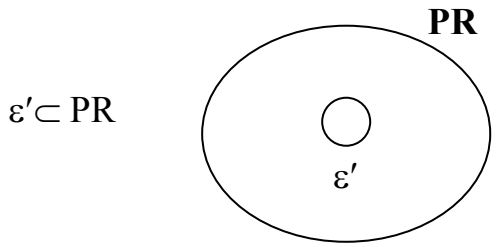


Figura 2.1 Le classi di funzioni

Non può esserci coincidenza tra i due insiemi raffiguranti le classi, poiché le condizioni richieste dal teorema 2.10 per ottenere funzioni elementari non sempre sono verificate; esistono funzioni ricorsive primitive che non appartengono all'insieme ε′. L'attenzione si sposta sulla valutazione della parte di PR che ha intersezione vuota con ε′

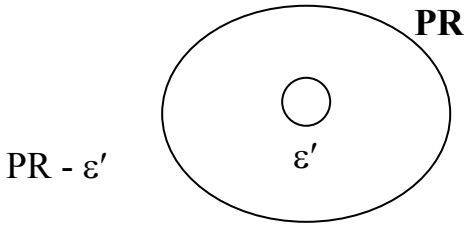


Figura 2.2 Le funzioni non elementari

e ci si chiede se sia possibile introdurre una classificazione di tutte le restanti funzioni di PR che non appartengono all'insieme ε′.

2.3 La gerarchia di Grzegorczyk

Una prima soluzione al problema è stata avanzata da Grzegorczyk che ricorre all'introduzione della gerarchia di funzioni $\varepsilon^0, \varepsilon^1, \varepsilon^2, ...$ che godono della proprietà $\cup_j \varepsilon^j = PR$. Per definire questa gerarchia di funzioni è necessario considerare un'altra funzione (denominata di Ackermann) esterna a PR.

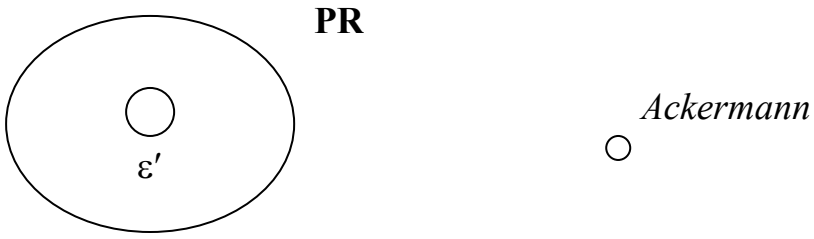


Figura 2.3 Legami tra due classi di funzioni e le funzioni di Ackermann

Definiamo tale funzione osservandone le caratteristiche e soprattutto il ruolo che esse giocano nella definizione della gerarchia di Grzegorczyk.

Definizione 2.11 *La sequenza di funzioni ricorsive primitive* $E_n(n = 0,1,...)$ *o* *rami della funzione di Ackermann* *è definita come segue:*

1) $E_0(x, y) = x + y$

2) $E_1(x) = x^2 + 2$

3) $E_{n+2}(0) = 2$

4) $E_{n+1}(x+1) = E_{n+1}(E_{n+2}(x))$.

Tutte le altre funzioni $E_2, E_3, ...$ si generano a partire dalle funzioni di ordine inferiore, per esempio E_2 da E_1, procedendo come segue:

$E_2(0) = 2$

$E_2(1) = E_1(E_2(0)) = E_1(2) = 2^2 + 2$

$E_2(2) = E_1(E_2(1)) = E_1(2^2) = 2^{2^2}$

In generale E_2(k) assume valore:

$E_2(k) = 2^{2^{\cdot^{\cdot^{2}}}}$ k volte l'input

È evidente che la funzione esponenziale cresce molto velocemente in funzione dell'input ricevuto. Successivamente calcoliamo E_3 sfruttando i risultati ottenuti precedentemente da E_2:

$E_3(0) = 2$

$E_3(1) = E_2(E_3(0)) = E_2(2) = 2^{2^2}$

$E_3(2) = E_2(E_3(1)) = E_2(2^{2^2}) = 2^{2^{\cdot^{\cdot^2}}}$ tante volte quanto vale 2^{2^2}

$E_3(3) = E_2(E_3(2)) = E_2(2^{2^{\cdot^{\cdot^2}}}) = 2^{2^{\cdot^{\cdot^2}}}$ tante volte quanto vale l'input

La funzione di Ackermann non è ricorsiva primitiva (come mostra la figura 2.3), ma ogni ramo E_i è ricorsivo primitivo.

Introduciamo un teorema che si propone di sottolineare la proprietà di monotonicità di cui godono le funzioni di Ackermann.

Teorema 2.12 *Per n ≥ 1, avremo che:*

i. $E_n(x) \geq x + 1$

ii. $E_n(x+1) \geq E_n(x)$

iii. $E_{n+1}(x) \geq E_n(x)$

iv. $E_n^t(x+1) \leq E_{n+1}(x+t)$

Definiamo gli elementi che costituiscono la gerarchia:

ε^0 : indica la classe a cui appartengono le funzioni Zero, Successore, Proiezione, è chiusa rispetto alla composizione, e alla ricorsione limitata;

ε^1: indica la classe a cui appartengono l0 Zero, Successore, Proiezione, il primo ramo della funzione di Ackermann E_0, è chiusa rispetto alla composizione e alla ricorsione limitata;

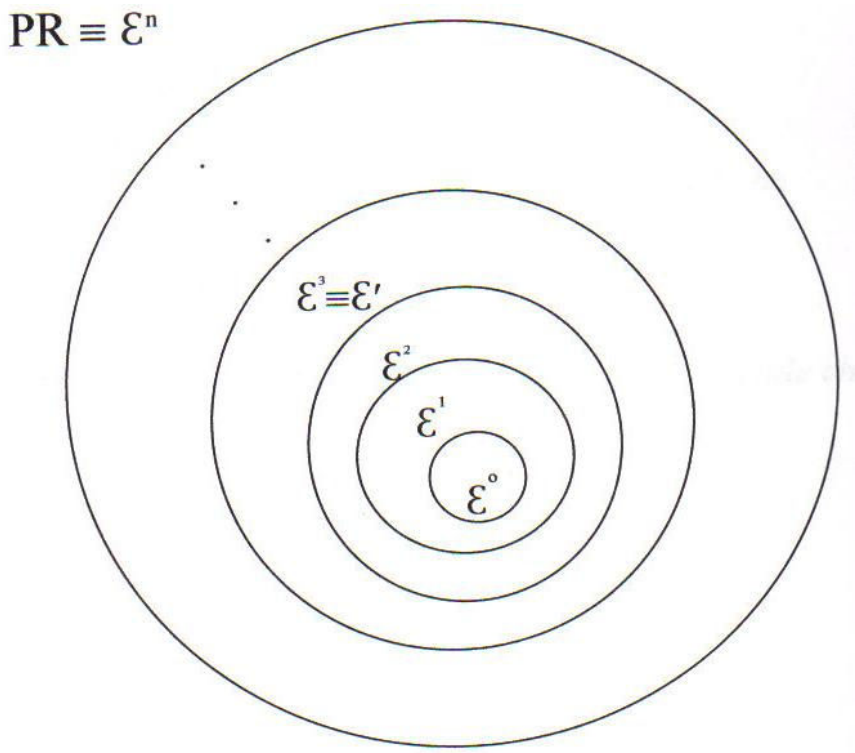
ε^2: indica la classe a cui appartengono lo Zero, Successore, Proiezione, il secondo ramo della funzione di Ackermann E_1, oltre ad E_0, è chiusa rispetto alla composizione e alla ricorsione limitata;

ε^{n+1}: indica la classe a cui appartengono lo Zero, Successore, Proiezione, i rami della funzione di Ackermann $E_0, E_1, \ldots, E_{n-1}, E_n$, è chiusa rispetto alla composizione e alla ricorsione limitata;

…

Tutte le classi della gerarchia di Grzegorzyk sono definite con lo stesso criterio cioè inglobare tutte le componenti che costituiscono le classi di ordine inferiore, alle quali si aggiunge, di volta in volta, un ramo della funzione di Ackermann. Quindi, tra esse sussiste la seguente relazione:

$$\varepsilon^0 \subset \varepsilon^1 \subset \varepsilon^2 \subset \ldots \subset \varepsilon^n \subset \ldots$$



Lemma 2.13 *Se m < n allora* $E_m \in \varepsilon^n$.

Teorema 2.14 *La classe* ε^3 *è la classe delle funzioni elementari ε′.*

Teorema 2.15 *Ogni classe* ε^n *è contenuta propriamente nella classe* ε^{n+1}.

Teorema 2.16 *L'unione* $\bigcup_{n<\omega} \varepsilon^n$ *è la classe delle funzioni ricorsive primitive.*

Lemma 2.17 *Se* $f \in \varepsilon^n$ *allora l'iterata* $f^y \in \varepsilon^{n+1}$.

Corollario 2.18 *Se g e h* $\in \varepsilon^n$ *ed f è definita per ricorsone primitiva usando g e h, allora f appartiene alla classe* ε^{n+1}.

3. Caratterizzazione di classi di complessità computazionale attraverso linguaggi funzionali

Le macchine di Turing (MT) sono il modello di calcolo di riferimento fondamentale sia nell'ambito della teoria della calcolabilità sia in quello della complessità computazionale. La teoria della calcolabilità si interessa dell'esistenza o meno di algoritmi per la soluzione di specifici problemi, oltre che della relazione tra i diversi modelli di calcolo. Quando il problema è stato riconosciuto come risolubile da un determinato modello di calcolo la teoria della calcolabilità non è interessata a determinare il costo di soluzione del problema in termini di risorse di calcolo utilizzate. Ad esempio, un determinato problema può risultare risolubile ma richiedere un tempo intollerabilmente lungo per il calcolo della soluzione, risultando così non risolubile in pratica. Quindi, la teoria della calcolabilità si limita a considerare aspetti qualitativi della

risolubilità di problemi, distinguendo ciò che è risolubile da ciò che non lo è.

La teoria della complessità si occupa della caratterizzazione e della classificazione dei problemi dal punto di vista della quantità di risorse di calcolo (tempo e memoria) necessarie per risolverli; tale classificazione discende dalle informazioni temporali e spaziali risultato dell'esecuzione vera e propria attraverso i modelli di calcolo degli algoritmi. La complessità computazionale implicita (vedi [12, 13] per un saggio) si propone invece di delineare una visione alternativa alla descrizione classica delle classi di complessità, sopra descritta, poiché mira alla valutazione di fattori definiti "impliciti" per classificare i problemi e non più ai "limiti espliciti" relativi alle risorse. Questo approccio ha come obiettivo l'analisi dei seguenti interrogativi:

- ✓ Possiamo descrivere le classi con linguaggi, piuttosto che con modelli di calcolo?
- ✓ Quali restrizioni dobbiamo imporre ai linguaggi?
- ✓ Esiste un principio comune a tutte le restrizioni?

3.1 Classificazione di Polytime con linguaggi funzionali

Il primo contributo è stato avanzato da Cobham che indagò sulla difficoltà computazionale intrinseca delle funzioni, partendo da un interrogativo apparentemente banale:

"*È più difficile moltiplicare o addizionare?*"

La risposta intuitiva è sicuramente moltiplicare, ma perché? Cobham introdusse il concetto di indipendenza delle soluzioni dei problemi dagli algoritmi e dai modelli di calcolo, sottolineando la necessità di definire le classi di complessità computazionale ricorrendo ad opportune classi di funzioni, e non a limiti sulle risorse usate dai modelli di calcolo. La classe di funzioni candidata da Cobham era la gerarchia di Grzegorczyk, dove è presente una naturale divisione tra funzioni progressivamente più complicate:

$x + y \in \varepsilon^0$

$x * y \in \varepsilon^1$

$x^y \in \varepsilon^2$

$x^{x^{\cdot^{\cdot^{x}}}}$ (x volte) $\in \varepsilon^3$.

Per le funzioni appartenenti a queste classi vale il seguente teorema:

f < g (f è "più semplice" di g) $\Rightarrow f \in \varepsilon^k$, $g \in \varepsilon^h$, *con k < h,*

cioè la quantità di risorse impiegate da f per essere computata sono minori di quelle necessarie per g. Il teorema afferma che da esse si può dedurre la classe a cui appartiene ciascuna delle due funzioni, e certamente la classe di f avrà ordine inferiore a quella in cui si colloca g, ma il viceversa non è sempre vero. Cobham fornisce la prima classificazione di Polytime. La classe C è la chiusura delle funzioni 1-4 tramite 5 e 6.

1. Zero: $o(x) = 0$;
2. Successore: $s(x) = x + 1$;
3. Proiezione: $i_j^n(x_1, x_2, ..., x_n) = x_j \quad 1 \leq j \leq n \;\; e \;\; n = 1,2,...$
4. Smash: $2^{|x||y|}$
5. $f(x) = g(x)$ **Composizione**
6. $\begin{cases} f(0,x) = g(x) & \textbf{Ricorsione limitata} \\ f(y_i,x) = h_i(x,y,f(x,y)) & \text{con } f(y,x) \leq k(y,x) \end{cases}$

Cobham prova che C è equivalente alla classe di funzioni calcolabili da MT in tempo polinomiale.

3.2 Predicatività dei linguaggi

Introduciamo le regole per calcolare le operazioni di addizione e moltiplicazione, sfruttando lo schema di ricorsione primitiva:

$$\begin{cases} \oplus(0,x) = x \\ \oplus(y+1,x) = (\oplus(y,x))+1 \end{cases}$$

$$\begin{cases} \otimes(0,a) = 0 \\ \otimes(b+1,a) = \oplus(a, \otimes(b,a)) \end{cases}$$

Per calcolare 3+5=8 e 3*5=15, si opera come segue:

$\oplus(3,5) = (\oplus(2,5))+1 =$

$= ((\oplus(1,5))+1)+1 =$

$= (((\oplus(0,5))+1)+1)+1 =$

$= (((5)+1)+1)+1 = 8$

$\otimes(3,5) = \oplus(5, \otimes(2,5)) = \ldots =$

$= \otimes(2,5) +1+1+1+1+1 =$

$= \oplus(5, \otimes(1,5))) +1+\underbrace{\ldots+1 =}_{\text{5 volte}} \ldots =$

$$= \otimes(1,5) \underbrace{+1+\ldots+}_{\text{10 volte}}1 = \ldots = 15$$

Si osserva che per definire la funzione prodotto abbiamo utilizzato la definizione di somma $\oplus$.

Ma questo non è l'unico modo per calcolare il prodotto, introduciamone un secondo e poniamoli a confronto.

$$\begin{cases} \otimes(0,a) = 0 \\ \otimes(b+1,a) = \oplus(\otimes(b,a), a) \end{cases}$$

Si osserva come la differenza tra le due definizioni risiede nell'inversione dei ruoli dei due fattori su cui il prodotto viene effettuato. Infatti, nella prima definizione la prima variabile ospita un numero (scriveremo $\oplus(\underline{\mathbf{5}}, \otimes(2,5))$), e quindi la funzione successore viene applicata 5 volte al secondo fattore; invece nella seconda definizione, non compare un numero come prima variabile, ma scriveremo $\oplus(\ \underline{\mathbf{\otimes(2,5)}}, 5)$; se applichiamo lo schema di ricorsione della somma quante volte si deve calcolare il successore di 5? $\otimes$**(2,5)** volte. L'inversione porta in quest'ultimo caso a definire la funzione prodotto in termini di se stessa, e ciò rende tale approccio

di calcolo meno efficiente del primo. L'introduzione di questa distinzione relativa all'ordine dei fattori è importante perchè permette di classificare le definizioni delle funzioni in predicative e impredicative. La prima definizione di $\otimes$ è predicativa, mentre la seconda è impredicativa perché $\otimes$ è definito in termini di se stesso.

Consideriamo altri due esempi di definizioni impredicative:

1. La definizione di Cobham di Polytime è impredicativa, poiché nell'asserire che una funzione f appartiene alla classe C si impone che $f(y,x) \leq k(y,x)$, dove k per ipotesi appartiene alla stessa classe.
2. La funzione esponente $\uparrow(x,2) = 2^x$:

$$\begin{cases} \uparrow(0,2) = 1 \\ \uparrow(y+1,2) = \oplus(\uparrow(y,2), \uparrow(y,2)) \end{cases}$$

è impredicativa perché uno dei due valori $\uparrow(y,2)$ deve essere necessariamente usato come variabile della ricorsione in $\oplus$. La definizione della funzione esponente si può ottenere solo in termini di se stessa.

3.3 Classificazione di Polytime con linguaggi funzionali predicativi

Bellantoni & Cook introducono una nuova caratterizzazione predicativa delle funzioni in polytime rinunciando alla ricorsione limitata. Ogni funzione ha un set di variabili safe e uno normal.

$$f(x,\ldots;\ y,\ldots)$$

normal **safe**

Le variabili safe non devono essere utilizzate come variabili della ricorsione di ogni altra funzione. La classe B è la chiusura delle funzioni 1-4 sotto 5 e 6:

1. Zero: $o(x) = 0$
2. Predecessore: $p(x_i) = x$
3. Successore: $s_0(;a) = a_0$

   $s_1(;a) = a_1$;
4. if the else: $if(;a,b,c) = \begin{cases} b & a = 0 \\ c & a \neq 0 \end{cases}$
5. $f(\vec{x};\vec{a}) = h(r(\vec{x};);t(\vec{x};\vec{a}))$ **composizione safe**

6. $\begin{cases} f(0,\vec{x};\vec{a}) = g((\vec{x};\vec{a}) & \textbf{ricorsione safe} \\ f(y_i,\vec{x};\vec{a}) = h_i(y,\vec{x};\vec{a},f(y,\vec{x};\vec{a})) \end{cases}$

Osserviamo che con la ricorsione safe possiamo esprimere somma e prodotto, ma non la funzione esponente. B è equivalente alla classe di funzioni calcolabili da MT in tempo polinomiale.

$$\begin{cases} \oplus(0\ ;\ x) = x \\ \oplus(y+1\ ;\ x) = s(\ ;\ \oplus(y\ ;\ x)) \end{cases} \qquad \begin{cases} \otimes(0,a) = 0 \\ \otimes(b+1,a\ ;\ ) = \oplus(a\ ;\ \otimes(b,a\ ;\ )) \end{cases}$$

3.4 Estensione alle classi superiori

Con la nuova caratterizzazione predicativa, la classe Polytime è caratterizzata dalle funzioni iniziali chiuse sotto safe recursione e safe composition. Cosa succede se violiamo il vincolo di non trasportare le variabili dalla zona safe a quella unsafe? K violazioni producono la k-esima classe della gerarchia di Grzegorczyk. C'è quindi uno stretto rapporto fra livelli di impredicatività e gerarchia di funzioni con difficoltà di calcolo progressivamente crescente.

4. Caratterizzazione di classi di complessità con linguaggi imperativi

Anche se la ricorsione safe riesce a catturare Polytime, lo fa passando attraverso il modello di Turing in modo inefficiente. Infatti, per esempio, semplici ordinamenti (polinomiali) non possono essere descritti con la ricorsione safe, e semplici funzioni (come il minimo) sono calcolate con complessità troppo alta.

Valutiamo l'analisi della funzione "minimo" ad opera di Colson. Supponiamo di voler calcolare la funzione minimo usando la MT a due nastri, su ciascuno dei quali vi è un numero in unario. Entrambe le testine sono posizionate sulla prima cella del contenuto; poi, si confrontano i simboli correnti: se i valori sono diversi dal b si può passare alle celle adiacenti e proseguire con i confronti, altrimenti se uno o entrambi sono uguali ci si arresta. Il minimo è sul nastro su cui compare il primo b, e la complessità di tale funzione corrisponde al numero di confronti effettuati.

Per esempio, min(2,5) ha complessità $O(\min(2,5))= O(2)$ dove 2 indica il numero di confronti necessari. La definizione ricorsiva della funzione minimo è

$$\begin{cases} \textbf{min}(0, y) = 0 \\ \textbf{min}(s(x), 0) = 0 \\ \textbf{min}(s(x), s(y)) = s(\min(x,y)) \end{cases}$$

Ad esempio, min(2,5) = s(min(1,4)) = s(s(min(0,3))) = s(s(0)) = 2, e il tempo di calcolo è $O(\min(2,5))$.

La definizione ricorsiva primitiva della funzione minimo è min'(x,y) = if(sub(x,y),y,x), dove:

$$\text{if } (a,b,c) = \begin{cases} b & \text{se } a \neq 0 \\ c & \text{se } a = 0 \end{cases}$$

Il tempo di calcolo di min' è $O(y)$. Si ha che min'(2, 5) = if(sub(2,5), 5, 2) = 2 ha tempo di calcolo $O(5)$; invertendo l'ordine: min'(5,2) = if(sub(5,2), 2, 5) = 2 ha tempo di calcolo $O(2)$. La complessità è sempre $O(y)$, e quindi dipende dal calcolo dei due valori in input; affinché la complessità sia minima è necessario che y sia il minimo tra i due input. Si deve quindi ricorrere alla definizione di minimo (funzione che stiamo ancora definendo) per ridurre la complessità.

Nel 1999 Neil Jones, propone un approccio alternativo che mette in discussione i risultati raggiunti negli anni precedenti. Riassumiamo il suo pensiero attraverso il seguente interrogativo:

"...what is the effect of the programmino style we employ (functional, imperative,...) on the efficiency of the programs we can possibly write?"

L'efficienza dei programmi può dipendere dallo stile di programmazione adottato. Quindi, per Jones, i problemi e le incoerenze incontrate per definire le classi computazionali attraverso i linguaggi funzionali, oggetto di analisi negli anni precedenti, sono la conseguenza dello stile scelto, che può essere sostituito con quello imperativo.

CAPITOLO II

1. Caratterizzazione di gerarchie di funzioni con linguaggi imperativi

1.1 Stack-program

Fissato un alfabeto $\Sigma := \{a_1,...,a_l\}$, definiamo un linguaggio di programmazione su Σ (Stack-language) caratterizzato da programmi (Stack-program) le cui *istruzioni primitive* o *comandi* sono:

- **push(a, X)** per $a \in \Sigma$;
- **pop(X)**;
- **nil(X)**;

alle quali si aggiunge *un'istruzione condizionale*:

- **if top(X) ≡ a[P];**

ed *un'istruzione ciclica*:

- **foreach X[P]** (indicata nel resto della tesi con **∀ X[P]).**

Definizione 2.1 *Gli stack-program sono definiti induttivamente come segue:*

- Ogni comando "imperativo" push(a, X), pop(X), nil(X) è uno stack-program;
- Se P_1, P_2 sono stack-program, allora anche la sequenza d'istruzioni $P_1; P_2$ è uno stack-program;
- Se P è uno stack-program, allora anche ogni istruzione condizionale if top(X) ≡ a[P] è uno stack-program;
- Se P è uno stack-program senza alcun'occorrenza di push(a, X), pop(X) o nil(X), allora anche l'istruzione ciclica foreach X[P] è uno stack-program.

Con V (P) denoteremo l'insieme delle variabili utilizzate da P.

Nota 2.2 Ogni stack-program può essere scritto nella forma $P_1;...;P_k$ э′ ciascun P_i sia un ciclo, un comando o un'istruzione condizionale.

Per specificare le pre- e le post-condizioni degli stack-program si utilizza la seguente notazione:

$$\{ A \} \; P \; \{ B \}$$

la quale significa che se la condizione A è soddisfatta prima di eseguire P, allora sarà anche soddisfatta la condizione B dopo l'esecuzione di P.

Per esempio, $\{\overrightarrow{X} = \vec{w}\}P\{\overrightarrow{X} = \vec{w}'\}$ indica che se le parole $\vec{w}$ sono memorizzate inizialmente nello stack $\overrightarrow{X}$, prima dell'esecuzione di P, allora le parole $\vec{w}'$ saranno memorizzate in $\overrightarrow{X}$ dopo l'esecuzione di P; ossia il contenuto dello stack $\overrightarrow{X}$ sarà stato modificato.

Un altro tipico esempio: $\{\overrightarrow{X} = \vec{w}\}P\{ X_1 |= f_1(|\vec{w}|),...,| X_n |\leq f_n(|\vec{w}|)\}$ significa che se le parole $\vec{w}$ sono caricate negli stack $\overrightarrow{X}$, prima dell'esecuzione di P, allora, poi, ogni parola memorizzata in X_i dopo l'esecuzione di P avrà lunghezza limitata da $f_i\left(\left|\vec{w}\right|\right)$.

Definizione 2.3 (Semantica Operazionale degli stack-program) *I comandi imperativi, le istruzioni condizionali e i cicli in pr* $P_1;...;P_k$ *sono eseguiti uno dopo l'altro da sinistra verso destra, dove:*

- push(a, X) mette la lettera a nel top dello stack;
- pop(X) rimuove il simbolo presente in cima allo stack, altrimenti se lo stack è vuoto l'istruzione è ignorata;
- nil(X) svuota lo stack;
- if top(X) ≡ a[P] esegue il corpo P se il simbolo in cima allo stack coincide con la lettera a, altrimenti l'istruzione condizionale è ignorata;
- L'istruzione ciclica foreach X[P] è eseguita call-by-value: è fatta una copia locale U così da poter modificare il contenuto della copia dello stack durante l'esecuzione, e P è eseguito un numero di volte pari alla lunghezza della parola in X.

Uno stack-program P calcola una funzione $f:(\Sigma^*)^n \to \Sigma^*$, se P ha una variabile di output O e variabili di input $X_{i1},...,X_{il}$ э′ $\{X_{i1} = wn_{i1},...,X_{il} = w_{il}\}P\{O = f(w_1,...,w_n)\}$; spesso abbreviata con : $\{\vec{X} = \vec{w}\}P\{O = f(\vec{w})\}$.

1.2 La misura μ negli stack-program

In questo paragrafo viene affrontato il problema dell'influenza dei loop annidati sulla complessità computazionale dei programmi; infatti, il metodo proposto si basa sull'assegnazione ad ogni stack-program P di un numero naturale μ(P), la sua μ-measure, calcolabile dalla sintassi del programma stesso. Questa misura permette di estrarre le informazioni necessarie per distinguere i programmi con complessità polinomiale da quelli che hanno complessità esponenziale. Osserveremo alcuni programmi con lo scopo di evidenziare i fattori responsabili dell'aumento della complessità di certi stack-program rispetto ad altri, ossia individuare i loop che possono causare un'esplosione nella complessità computazionale e quelli che non provocano tale effetto.

Definiamo i seguenti insiemi:

$U(P)$:= {X| P contiene almeno una push(a,X)

$C(P)$:= {X| P contiene un ciclo foreach X[Q] con $U(Q) \neq \varnothing$ }.

Definizione 2.4 *Sia P uno stack-program. La* relazione di controllo in P*, denotata* $\xrightarrow{P}$*, è definita come la chiusura transitiva della relazione binaria* π_P *su V (P):*

$$X\ \pi_P\ Y: \Leftrightarrow P \text{ contiene un ciclo foreach } X[Q] \text{ e } Y \in U(Q).$$

Quindi, si ha che X controlla Y in P ($X \xrightarrow{P} Y$) se esistono le variabili $X \equiv X_1, X_2, ..., X_l \equiv Y\ \ni'\ X_1\ \pi_P\ X_2\ \pi_P\ ...\ \pi_P\ X_{l-1}\ \pi_P\ X_l$.

Un esempio di relazione diretta π_P tra le variabili di un programma è la seguente: P ≡ …∀ X[…push(a,Y)]; infatti, in P compare un ciclo ∀X nel cui corpo Q c'è almeno una push su Y, $\ni'\ U(Q) \neq \varnothing$.

Mentre un esempio di relazione di controllo è fornito dal seguente programma: P ≡ …∀ X[…∀A[…∀B[…push(a,Y)…]…]…]; in cui compaiono più variabili in relazione diretta, e quindi per la proprietà transitiva si può parlare di controllo su Y da parte di X.

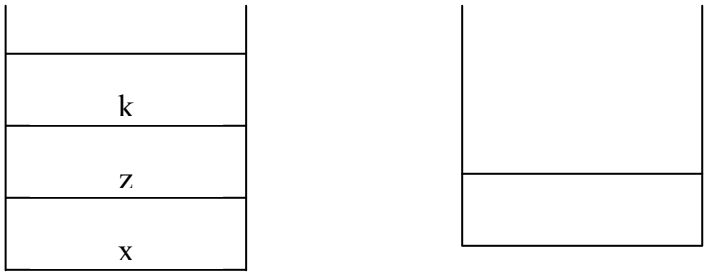
Sfruttando le definizioni introdotte, consideriamo tre esempi di stack-program, tutti con loop annidati. L'obiettivo sarà calcolarne la complessità computazionale per mostrare come in realtà la presenza di loop annidati con una certa profondità sia una condizione necessaria ma non sufficiente affinché si abbia un'esplosione nella

complessità del programma, la cui vera causa sarà individuata nelle relazioni tra le variabili. Infatti, certi programmi con un alta profondità di loop annidati sono eseguiti in tempo polinomiale.

Il primo programma è il seguente:

$$P_1 \equiv \forall X[\forall X[\forall X[\forall[\forall X[push(a,Y)]]]]]$$

Le variabili che compaiono in P_1 sono X e Y, quindi $V(P_1)$={X,Y} con cardinalità $|V(P_1)|$ = 2. Supponendo che inizialmente nello stack X sia stata memorizzata la parola v = "xzk" arbitrariamente scelta sull'alfabeto Σ, si ha:



Stack X **Stack Y**

Le dimensioni degli stack sono |X| = 3, |Y| = 0.

Adesso scriviamo P_1 nel linguaggio Pascal-like, dove utilizzeremo cinque variabili intere: "i" (per scandire il ciclo più esterno sullo stack X), "j" (per scandire il secondo ciclo su X), "r" (per scandire il

terzo ciclo sempre sullo stack X), “s” (per scandire il quarto ciclo su X) e, infine, “t” (per scandire il ciclo più interno sempre su X). Tutte queste variabili assumono valori da 1 a 3 poiché |X| = 3 e X non subisce variazioni durante l’esecuzione di P_1.

begin

for i←1 a 3 do

for j←1 a 3 do

for r←1 a 3 do

for s←1 a 3 do

for t←1 a 3 do

Y←Ya

end;

Analizziamo il programma, rappresentando il contenuto dello stack Y man mano che i contatori vengono incrementati:

Contatori	Stack Y
i=1 j=1 r=1 s=1 t=1,2,3	$Y \leftarrow a^3$
i=1 j=1 r=1 s=2 t=1,2,3	$Y \leftarrow a^6$
i=1 j=1 r=1 s=3 t=1,2,3	$Y \leftarrow a^9$

Contatori	Stack Y
i=1 j=1 r=2 s=1 t=1,2,3	$Y \leftarrow a^{12}$
i=1 j=1 r=2 s=2 t=1,2,3	$Y \leftarrow a^{15}$
i=1 j=1 r=2 s=3 t=1,2,3	$Y \leftarrow a^{18}$
i=1 j=1 r=3 s=1 t=1,2,3	$Y \leftarrow a^{21}$
i=1 j=1 r=3 s=2 t=1,2,3	$Y \leftarrow a^{24}$
i=1 j=1 r=3 s=3 t=1,2,3	$Y \leftarrow a^{27}$
i=1 j=2 r=1 s=1 t=1,2,3	$Y \leftarrow a^{30}$
i=1 j=2 r=1 s=2 t=1,2,3	$Y \leftarrow a^{33}$
i=1 j=2 r=1 s=3 t=1,2,3	$Y \leftarrow a^{36}$
i=1 j=2 r=2 s=1 t=1,2,3	$Y \leftarrow a^{39}$
i=1 j=2 r=2 s=2 t=1,2,3	$Y \leftarrow a^{42}$
i=1 j=2 r=2 s=3 t=1,2,3	$Y \leftarrow a^{45}$
i=1 j=2 r=3 s=1 t=1,2,3	$Y \leftarrow a^{48}$
i=1 j=2 r=3 s=2 t=1,2,3	$Y \leftarrow a^{51}$
i=1 j=2 r=3 s=3 t=1,2,3	$Y \leftarrow a^{54}$
i=1 j=3 r=1 s=1 t=1,2,3	$Y \leftarrow a^{57}$
i=1 j=3 r=1 s=2 t=1,2,3	$Y \leftarrow a^{60}$

Contatori	Stack Y
i=1 j=3 r=1 s=3 t=1,2,3	$Y \leftarrow a^{63}$
i=1 j=3 r=2 s=1 t=1,2,3	$Y \leftarrow a^{66}$
i=1 j=3 r=2 s=2 t=1,2,3	$Y \leftarrow a^{69}$
i=1 j=3 r=2 s=3 t=1,2,3	$Y \leftarrow a^{72}$
i=1 j=3 r=3 s=1 t=1,2,3	$Y \leftarrow a^{75}$
i=1 j=3 r=3 s=2 t=1,2,3	$Y \leftarrow a^{78}$
i=1 j=3 r=3 s=3 t=1,2,3	$Y \leftarrow a^{81}$
i=2 j=1 r=1 s=1 t=1,2,3	$Y \leftarrow a^{84}$
…….	
i=3 j=3 r=3 s=3 t=1,2,3	$Y \leftarrow a^{243}$

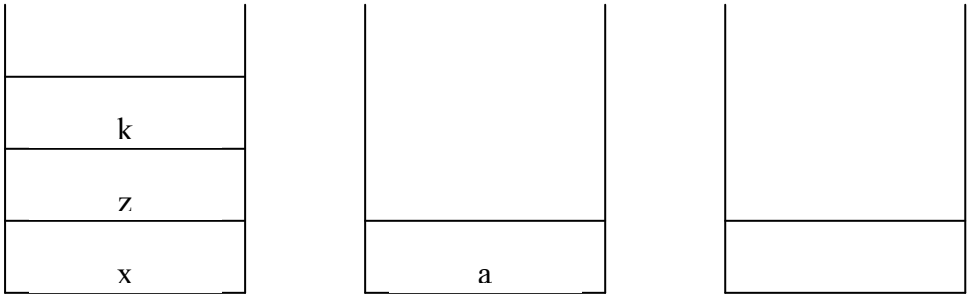
Al termine dell'esecuzione di P_1 il numero delle "a" nello stack Y è pari a 243, ovvero 3^{27}. Si deduce che $|Y| = |X|^{27}$, quindi questo programma è in grado di calcolare i polinomi di grado 27. P_1, pur avendo 5 cicli annidati, ha complessità polinomiale. Possiamo dedurre che la profondità d'annidamento non è la causa primaria che provoca l'aumento della complessità di un programma.

Il secondo programma, caratterizzato da profondità di annodamento uguale a 2, è il seguente:

$$P_2 : \equiv \text{nil(Y); push(a,Y); nil(Z);}$$

$$\forall X[\ \forall Y[\ \text{push(a,Z)}\ ;\ \text{push(a,Z)}]\ ;\ \text{push(a,Y)}]$$

Le variabili che compaiono in P_2 sono X, Y, Z quindi $V(P_2)=\{X,Y,Z\}$ con cardinalità $|V(P_2)| = 3$. Supponendo che inizialmente nello stack X sia stata memorizzata la parola v = "xzk" arbitrariamente scelta sull'alfabeto Σ, dopo l'esecuzione dei primi tre comandi imperativi ci troveremo nella seguente situazione:



Stack X **Stack Y** **Stack Z**

Le dimensioni degli stack sono: |X| = 3, |Y| = 1, |Z| = 0.

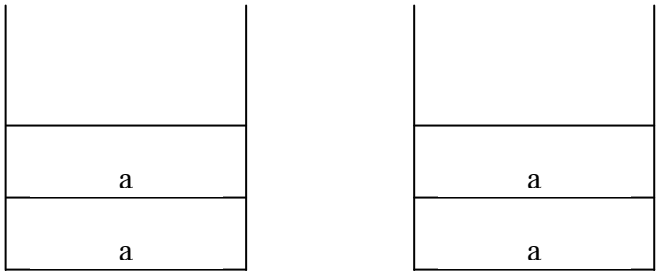
Utilizzeremo due variabili intere: "i" (per scandire il ciclo più esterno sullo stack X) e "j" (per scandire il ciclo più interno sullo

stack Y). Anche in questo caso scriviamo la seconda parte di P_2 (quella contenente i due loop annidati) nel linguaggio Pascal-like:

```
begin
    for  i←1  a  3  do
    begin
        for  j←1  a  1  do
            Z←2a
    end
    Y←a
end;
```

In generale, il numero di volte che un ciclo è ripetuto coincide con la cardinalità dello stack a cui si riferisce. Notiamo che, il ciclo più esterno, scandito dal contatore i, viene ripetuto tre volte poiché |X| = 3; inoltre, X non subisce modifiche durante l'esecuzione di P_2 e quindi i valori dell'indice non saranno alterati duranti i vari passi. Mentre, il ciclo più interno, scandito dal contatore j (inizializzato a 1), viene ripetuto solo una volta perchè |Y| = 1. Prima della chiusura del ciclo esterno, viene eseguita l'istruzione $Y \leftarrow a$.

Quindi, dopo la prima fase, ci troveremo nella seguente situazione:



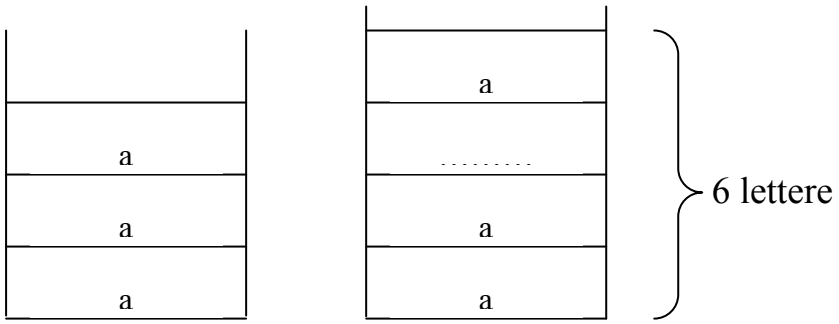
Stack Y **Stack Z**

Le dimensioni, a questo punto, sono: |X| = 3, |Y| = 2, |Z| = 2.

Nella seconda fase, il ciclo più esterno è incrementato di una unità, mentre j viene reinizzializzato. Osserviamo che lo stack Y ha cardinalità pari a 2 e quindi il ciclo sarà eseguito due volte.

```
begin
    for   i←2   a   3   do
    begin
        for   j←1   a   2   do
            Z←2a
    end
    Y← a
end;
```

Dopo la seconda fase, ci troveremo nella seguente situazione:

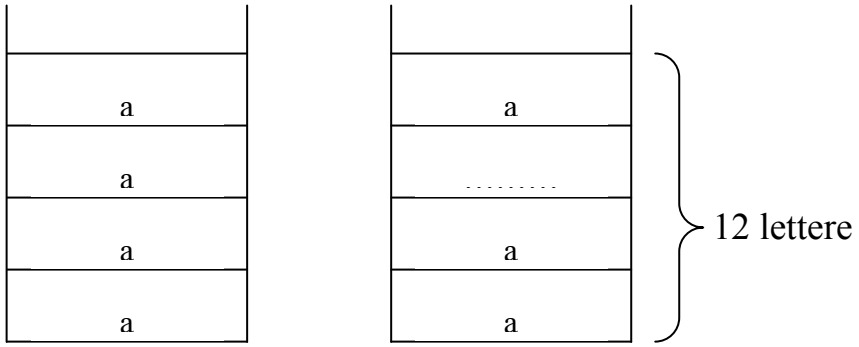


Le dimensioni, adesso, sono: |X| = 3, |Y| = 3, |Z| = 6.

Nella terza fase, il ciclo più esterno è incrementato nuovamente di una unità, mentre j viene reinizzializzato. Notiamo che Y ha cardinalità pari a 3 e quindi il ciclo sarà eseguito tre volte.

```
begin
   for  i←3  a  3  do
   begin
      for  j←1  a  3  do
         Z←2a
   end
   Y← a
end;
```

Dopo la terza fase, la situazione finale è la seguente:



Le dimensioni al termine dell'esecuzione di P_2 sono: |X| = 3, |Y| = 4, |Z| = 12.

Il risultato ottenuto con P_2 e contenuto in Z, può essere generalizzato nel seguente modo: $a^{|w|\cdot(|w|+1)}$. Siamo partiti con |X| = 3 = |w| e applicando la formula precedente otteniamo $a^{3(3+1)} = a^{12}$. Quindi anche P_2, come P_1, è stato eseguito in tempo polinomiale.

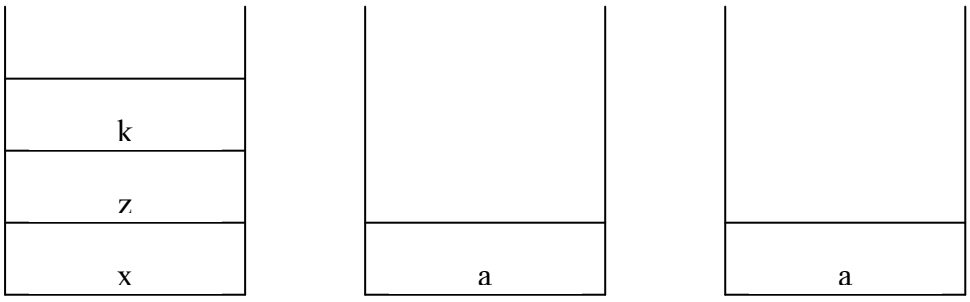
Il terzo programma, apparentemente molto simile a P_2, è il seguente:

P_3: ≡ nil(Y); push(a,Y); nil(Z); push(a,Z) ;

∀X[nil(Z); ∀Y[push(a,Z) ; push(a,Z)] ;

nil(Y); ∀Z[push(a,Y)]]

Le variabili che compaiono in P_3 sono X, Y, Z quindi $V(P_3)$={X,Y,Z} con cardinalità $|V(P_3)|$ = 3. Supponendo che inizialmente nello stack X sia stata memorizzata la parola v = "xzk" arbitrariamente scelta sull'alfabeto Σ, dopo l'esecuzione dei primi quattro comandi imperativi ci troveremo nella seguente situazione:



Stack X **Stack Y** **Stack Z**

Le dimensioni degli stack sono: |X| = 3, |Y| = 1, |Z| = 1.

Useremo tre variabili intere: "i" (per scandire il ciclo più esterno sullo stack X), "j" (per scandire il ciclo sullo stack Y) e "r" (per scandire il ciclo su Z). Come sempre, rappresentiamo la seconda parte di P_3 nel linguaggio Pascal-like:

begin

for i←1 a 3 do

begin

Azzera Z

for j←1 a 1 do

Z←2a

Azzera Y

for r←1 a 2 do

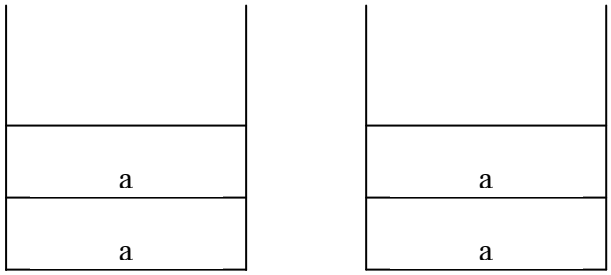
Y← a

end

end;

Anche in questo caso, come in P_1 e in P_2, il ciclo più esterno, scandito dal contatore i, viene ripetuto tre volte poiché |X| = 3; inoltre, X non subisce modifiche durante l'esecuzione di P_3 e quindi i valori dell'indice i resteranno immutati. Mentre il secondo ciclo, scandito dal contatore j (inizializzato a 1), viene ripetuto solo una volta perchè |Y| = 1 e quindi viene eseguita solo una volta l'istruzione di assegnazione *Z←2a*. Infine, l'ultimo loop, scandito dal contatore r, viene ripetuto due volte poiché all'interno di questa stessa fase il contenuto dello stack Z è stato modificato.

Quindi, al termine della prima fase, ci troveremo nella seguente situazione:



Stack Y **Stack Z**

Le dimensioni, a questo punto, sono: |X| = 3, |Y| = 2, |Z| = 2.

Nella seconda fase, il ciclo for più esterno è incrementato di una unità, j viene reinizzializzato e Z azzerato completamente. Notiamo che lo stack Y adesso ha cardinalità pari a 2 e quindi il ciclo sarà eseguito due volte ottenendo così |Z| = 4 e determinando in tal modo la ripetizione del ciclo più interno di ben quattro volte.

begin

for i←2 a 3 do

begin

Azzera Z

for j←1 a 2 do

Z←2a

Azzera Y

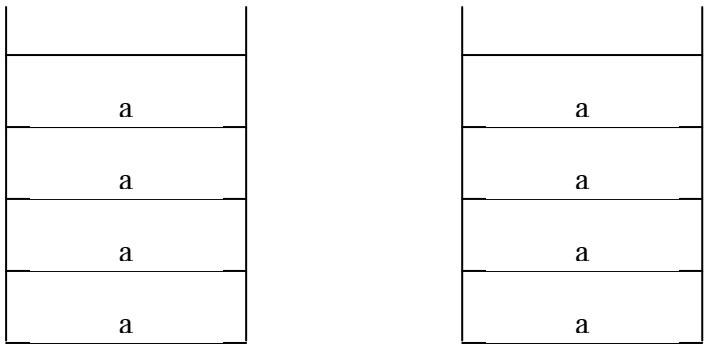
for r←1 a 4 do

Y← a

end

end;

Al termine della seconda fase, ci troveremo nella situazione:



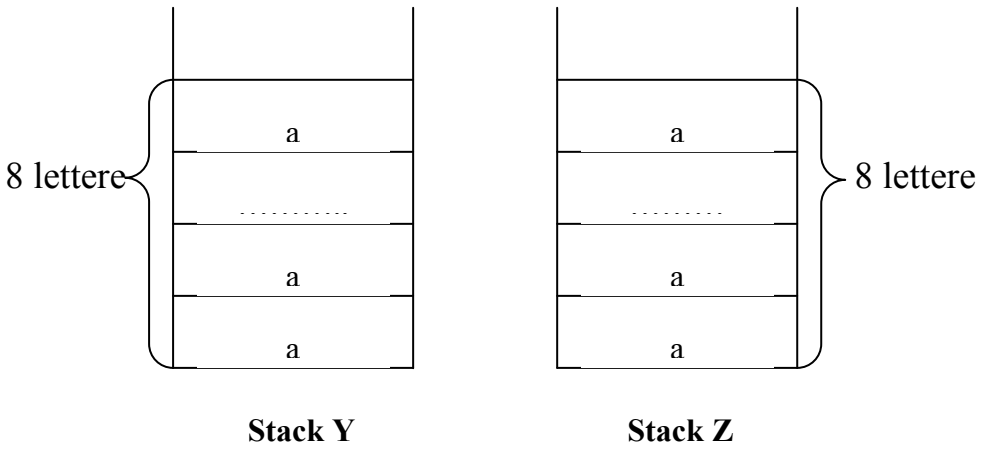
Stack Y **Stack Z**

Le dimensioni, adesso, sono: |X| = 3, |Y| = 4, |Z| = 4.

Nella terza fase, il ciclo più esterno è incrementato di una unità, j viene reinizzializzato e Z. Y ha cardinalità pari a 4 e il ciclo sarà eseguito 4 volte ottenendo |Z| = 8 determinando la ripetizione del ciclo interno otto volte.

```
begin
    for  i←3  a  3  do
    begin
          Azzera Z
         for  j←1  a  4  do
              Z←2a
         Azzera Y
        for  r←1  a  8  do
             Y← a
   end
end;
```

Dopo la terza fase, la situazione finale è la seguente:



Le dimensioni al termine dell'esecuzione di P_3 sono: |X| = 3, |Y| = 8, |Z| = 8.

Il risultato finale, contenuto nello stack Y e Z, mostra che se inizialmente |X| = 3, questo influenza il risultato ottenuto perché |Y| = 2^3, cioè $a^{2^{|w|}}$. Quindi il programma P_3 è in grado di calcolare la funzione $f : X \to 2^X$, dove 2 sta ad indicare il numero di push presenti nel ciclo sullo stack Y. Ovviamente, se ci fossero tre push P_3 sarebbe in grado di calcolare la funzione $f : X \to 3^X$.

Osserviamo che P_2 e P_3, pur avendo lo stesso livello di annidamento pari a 2, hanno complessità computazionale diversa; infatti, P_2 ha prodotto come risultato $a^{|w|\cdot(|w|+1)}$ all'interno di Z, mentre P_3 $a^{2^{|w|}}$ sempre in Z. Da ciò si deduce che P_2 è eseguito in tempo polinomiale e P_3 in tempo esponenziale. Questa differenza dipende dalle relazioni esistenti tra le variabili in ciascun programma. Consideriamo il programma P_3: all'interno del ciclo più esterno su X, Y controlla Z (Y→Z) e in quello successivo Z controlla Y (Z→Y).

P_3: ≡ nil(Y); push(a,Y); nil(Z); push(a,Z) ;

∀X[nil(Z); ∀Y[push(a,Z) ; push(a,Z)] ;

nil(Y); ∀Z[push(a,Y)]]

Se consideriamo, invece, il programma P_2 non si verifica una situazione simile ma l'unica relazione esistente tra le variabili che si può individuare è che Y controlla Z (Y→Z).

P_2 : ≡ nil(Y); push(a,Y); nil(Z);

∀X[∀Y[push(a,Z) ; push(a,Z)] ; push(a,Y)]

La stessa situazione si verifica nel programma P_1, dove X controlla Y (X→Y).

$$P_1 \equiv \forall X[\forall X[\forall X[\forall[\forall X[push(a,Y)]]]]]$$

Per quanto riguarda P_1 e P_2 possiamo parlare di *programmi liberi da circoli*; mentre, quando la relazione di controllo crea un circolo chiuso, come in P_3, allora si parlerà di *programmi con circoli*. Da ciò deduciamo la differenza che si trova alla base dell'esplosione nella complessità computazionale: gli stack-program privi di circoli saranno eseguiti in tempo polinomiale, mentre quelli con circoli avranno tempi di esecuzione almeno esponenziale.

Introduciamo il concetto di μ-measure: un criterio sintattico per calcolare la complessità computazionale degli stack-program e che permette di risalire alla classe della gerarchia di Grzegorczyk in cui collocare le funzioni che tali programmi calcolano.

Consideriamo il programma P costituito dai sottoprogrammi $Q_1, Q_2, ..., Q_m$.

1. Se in P tutti i sottoprogrammi sono comandi imperativi, allora non esistono controlli tra le variabili e perciò non ci sono circoli:

$$\mu(Q_i) = 0 \quad \forall \text{ i} = 1,\ldots, \text{m} \quad \Rightarrow \quad \mu(\text{P}) = 0$$

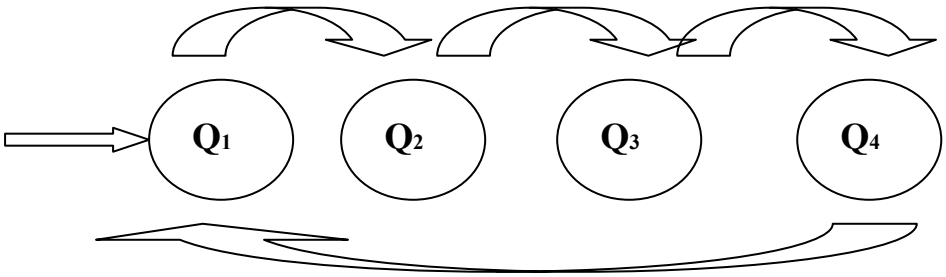
2. Supponiamo di aver calcolato la μ-measure relativa a ciascun sottoprogramma di P; possiamo trovarci in una delle tre seguenti casi:

Non esiste una sequenza di controlli tra le variabili dei sottoprogrammi tali da creare un circolo chiuso, quindi il calcolo della misura μ di P è:

$$\mu(P) = \mu(Q_1; Q_2; ...; Q_m) = \max\{\mu(Q_1); ...; \mu(Q_m)\}$$

2.2 Esiste una relazione di controllo tra le variabili di tutti i sottoprogrammi e si possono verificare due situazioni diverse. Vediamo la prima attraverso un esempio:

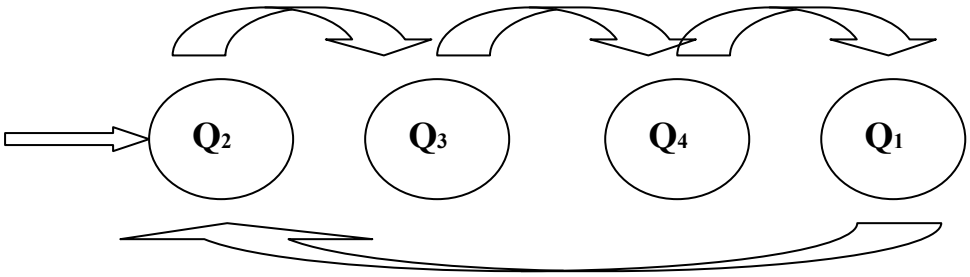
Sia $P := Q_1; Q_2; Q_3; Q_4$ e $\mu(Q_1) = 21$, $\mu(Q_2) = 17$, $\mu(Q_3) = 15$, $\mu(Q_4) = 10$. Supponendo di partire dal sottoprogramma Q_1 e di individuare delle relazioni di controllo tra le variabili che generano un circolo chiuso che torna in Q_1, allora $\mu(P) = \mu(Q_1) + 1 = 22$.



In questo caso, notiamo che il circolo formatosi è un ***Top Circle*** poiché il valore di $\mu(P)$ coincide con quello del sottoprogramma di partenza Q_1 che ha μ massima tra tutte le μ, incrementata di uno.

2.3 Nella seconda situazione, si parlerà semplicemente di ***Circolo*** quando, nell'esempio precedente, supponiamo di partire da Q_2. In questo caso, il circolo si chiude proprio in Q_2 e quindi andrà incrementata di una unità la μ relativa a tale sottoprogramma, cioè $\mu(Q_2) = \mu(Q_2) + 1 = 17 + 1 = 18$. Di conseguenza, avremo che:

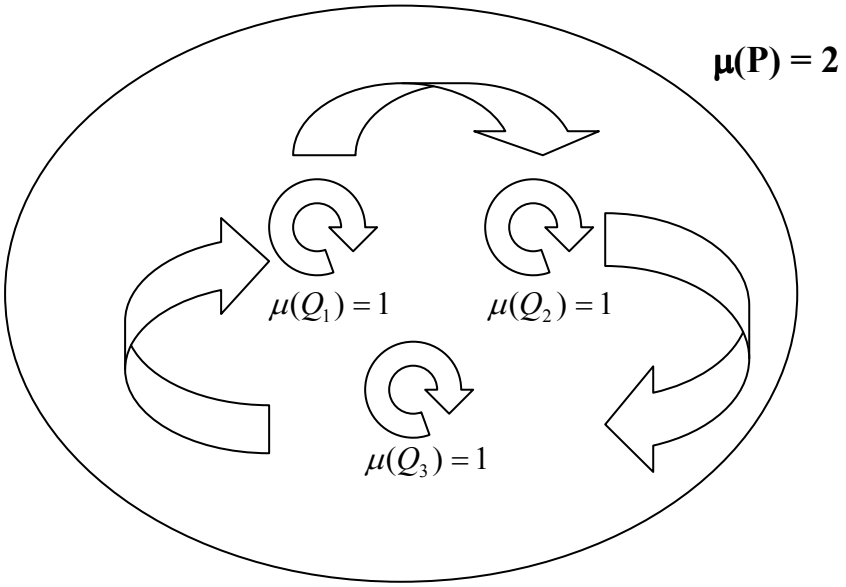
$$\mu(P) = \mu(Q_1; Q_2; Q_3; Q_4) = \max\{\mu(Q_1); \mu(Q_2); \mu(Q_3); \mu(Q_4)\} = \mu(Q_1)$$



Il valore di $\mu(P)$, in questo caso, non coincide con quello di Q_2, poiché il massimo tra tutte le μ dei sottoprogrammi è quello relativo a Q_1.

Solo se in P è presente un top circle, ci sarà un incremento nella complessità computazionale, ciò invece non si verificherà se ci sono dei semplici circoli, o dei cicli liberi.

Il valore $\mu(P)$ corrisponde al massimo μ dei sottoprogrammi che lo compongono. A P_1 e P_2 sarà assegnata $\mu(P_1) = \mu(P_2) = 0$, per P_3 $\mu(P_3) = 1$. Gli stack-program con μ = 0 calcolano funzioni polinomiali, mentre la presenza di un top circle permette il calcolo della funzione exp(x) = 2^X. Se il numero di top circle fosse 2:



Usando la regola per il calcolo di μ,il valore: $\mu(P) = \mu(Q_i) + 1 = 2$. P calcolerà una funzione exp(x) = 2^{2^X}; se i top circle sono tre, la funzione calcolata da P sarà: exp(x) = $2^{2^{2^X}}$. μ costituisce un criterio sintattico per classificare i programmi e per risalire alla classe della gerarchia a cui appartengono le funzioni. Il programma con μ(P) = 1 calcola la funzione $f_P : 2^X \in \varepsilon^2$,se invece μ(P) = 2 $f_P \in \varepsilon^3$.

Definizione 2.5 *La* **μ-measure** *di uno stack-program P, denotato con* μ(P)*, è definita in maniera induttiva nel seguente modo:*

- *Se P è un comando imperativo:*

$$\begin{cases} \mu(\text{push}) = 0 \\ \mu(\text{pop}) = 0 \\ \mu(\text{nil}) = 0 \end{cases}$$

- *Se P è un'istruzione condizionale:*

  $\mu(\text{ if top (X)} \equiv \text{a[Q] }) := \mu(\text{Q})$

- *Se P è una sequenza di istruzioni:*

  $\mu(Q_1, Q_2) = \max\{\mu(Q_1), \mu(Q_2)\}$

- *Se P è un ciclo foreach X[Q]:*

$$\mu(\text{P}) := \begin{cases} \mu(\text{Q}) + 1 & \text{se Q è una sequenza con un top circle} \\ \mu(\text{Q}) & \text{altrimenti} \end{cases}$$

Dove $Q \equiv Q_1; Q_2; ....; Q_{lm}$ *ha un* <u>top circle</u> *se* $\exists$ *un sottoprogramma* Q_i *con* $\mu(Q_i) = \mu(Q)$ *tale che* $Y \rightarrow Z$ *in* Q_i *e* $Z \rightarrow Y$ *in* $(Q_1, ..., Q_{i-1}, Q_i, Q_{i+1}, ..., Q_m)$.

1.3 Teorema del Limite per gli stack-program

Lo scopo di questo paragrafo è quello di mostrare che per qualsiasi funzione f, calcolata da uno stack-program P di misura $\mu(P) = n$, esiste una funzione b nella classe ε^{n+2} tale che $|f(w)| \leq b|(w)|$, cioè b rappresenta un limite superiore per f. È sufficiente dimostrare questo teorema del limite per una sottoclasse degli stack-program, parliamo cioè dei core program. Verifichiamo che ogni funzione calcolata da un core program di misura 0 ha tempo di esecuzione polinomiale; nel caso generale si dimostrerà che per ogni core program P di misura n+1 c'è un programma P′ di misura n+1 che calcola la stessa funzione calcolata da P ed è un limite per P stesso. Inoltre la funzione P′ è nella classe ε^{n+2}.

Definizione 2.6 *Un* **core program** *è uno stack-program costituito dalle sole istruzioni push(a,X), sequenza e ciclo.*

Quindi sono assenti i comandi pop(a, X) e nil(X); ciò implica che questi programmi possono solo aumentare la dimensione degli stack a cui sono applicate.

Osserviamo che l'istruzione ciclica viene eseguita call-by value e questo assicura che i core program siano monotoni sulle lunghezze. Infatti, se P è un core program con $\vec{X}$ variabili si ha:

$$\{\vec{X} = \vec{w}\}P\{\vec{X} = \vec{u}\} \Rightarrow |\vec{w}| \leq |\vec{u}|$$

Se $|\vec{w}| \leq |\vec{w'}|$ (cioè a due parole di lunghezza diversa applico lo stesso programma P) si ha:

$$\{\vec{X} = \vec{w}\}P\{\vec{X} = \vec{u}\} \wedge \{\vec{X} = \vec{w'}\}P\{\vec{X} = \vec{u'}\} \Rightarrow |\vec{u}| \leq |\vec{u'}|$$

La proprietà di monotonicità è importante perché sottolinea come l'incremento della dimensione dello stack avvenga in maniera proporzionale alla lunghezza iniziale della parola.

Lemma 2.7 *Per ogni core program P:≡foreach X[Q] con μ(P) = 0,* $\xrightarrow{P}$ *è irriflessiva.*

Dimostrazione: per induzione sulla struttura dei core program con μ-measure = 0.

PASSO BASE: P:= push(a,Y) è ovvio, poiché, essendo un comando imperativo con μ(P) = 0, non ci sono circoli e tanto meno relazioni di controllo in P.

PASSO INDUTTIVO: Sia P:= foreach X[Q], analizziamo tutte le possibili situazioni:

1) Se Q:= push(a,Y), si ha che X→Y ma non ci sono circoli, quindi $\xrightarrow{P}$ è irriflessiva;

2) Se Q è nella forma foreach Y[R] (P:=∀X [∀Y [R]]) e $\xrightarrow{R}$ è irriflessiva, allora, per l'ipotesi induttiva su Q e poiché X $\notin$ *U*(Q), X controlla le variabili di R ma non controlla se stesso; quindi $\xrightarrow{P}$ è irriflessiva;

3) Se Q è una sequenza di sottoprogrammi $Q_1;Q_2;...;Q_n$, allora per l'ipotesi induttiva su ogni Q_i, nessuna Y controlla Y in Q_i; quindi $\xrightarrow{P}$ è irriflessiva. Altrimenti, se qualche Y controlla Y in Q, allora Y controlla qualche Z (con Z che non coincide con Y) in qualche Q_j, e Z controlla Y nel contesto $Q_1;...;Q_{j-1};Q_{j+1};...;Q_n$; di conseguenza, Q potrebbe avere un top circle contraddicendo l'ipotesi che μ(P) = 0.

□

Poiché i core program sono irriflessivi, cioè non hanno top circle, la loro misura è sempre pari a 0; proveremo che il tempo di esecuzione è al più polinomiale.

Lemma 2.8 *Sia P un core program con* $\xrightarrow{P}$ *irriflessiva tale che:*

- *P ha le variabili* $\vec{X} := X_1,...,X_n$*;*
- *per i=1,...,n sia* V^i *la lista delle variabili* X_j *che controllano* X_i *in P, ossia* $V^i = \{X_j \mid X_j \xrightarrow{P} X_i\}$.

Allora esistono i polinomi $p_1(V^1),...,p_n(V^n)$ *tali che:*

$\forall \vec{w} := w_1,...,w_n \quad \{\vec{X} = \vec{w}\} P \{\mid X_i \mid \leq \mid w_i \mid + p_i(\mid \vec{w}^i \mid)\} \quad \forall \quad i=1...n,$ *e dove* $\vec{w^i}$ *deriva da* $\vec{w}$ *selezionando quelle* w_j *per le quali* X_j *è in* V^i.

Dimostrazione: per induzione sulla struttura dei core program con $\xrightarrow{P}$ irriflessiva.

PASSO BASE: P:= push(a, X_1) con $V^1 = \varnothing$ (poiché non ci sono cicli); quindi, avremo che:

$$\{X_1 = w_1\} P \{\mid X_1 \mid = w_1 + 1\} \Rightarrow p_1(w_1) = 1$$

La cardinalità dello stack X_1, in questo caso, si ricava facilmente perché, non essendoci variabili che controllano X_1, dopo l'esecuzione di P, il suo contenuto finale è quello iniziale incrementato solo di una unità che corrisponde all'unica push di P in X_1.

PASSO INDUTTIVO: analizziamo le due possibili situazioni:

1. P:≡ $P_1; P_2$
2. P:≡ foreach X_j [Q]

Cominciamo il primo caso, cioè P:≡ $P_1; P_2$.

Applicando l'ipotesi induttiva su P_1 e su P_2, esistono i polinomi: $q_1(V^1),...,q_n(V^n)$ per P_1 e $r_1(V^1),...,r_n(V^n)$ per P_2. Si fissa un i tra 1,…,n e si suppone che $X_{i1},...,X_{il}$ siano le variabili che controllano X_i in P. Quindi, dato che:

$$\{\overleftarrow{X} = \overrightarrow{w}\}P_1\{\mid X_i \mid \leq \mid w_i \mid + q_i(\mid \overrightarrow{w^i} \mid)\}$$

$$\{\overleftarrow{X} = \overrightarrow{w}\}P_2\{\mid X_i \mid \leq \mid w_i \mid + r_i(\mid \overrightarrow{w_i} \mid)\}$$

dopo l'esecuzione della sequenza $P_1; P_2$, avremo che:

$$\{\overleftarrow{X} = \overrightarrow{w}\}P\{\mid X_i \mid \leq \mid w_i \mid + q_i(\mid \overrightarrow{w^i} \mid) + r_i(\mid w_{i1} \mid + q_{i1}(\mid \overrightarrow{w^{i1}} \mid),...,\mid w_{il} \mid + q_{il}(\mid \overrightarrow{w^{il}} \mid))\}$$

Passiamo al secondo caso, cioè P:≡ foreach X_j [Q] con $\mid X_j \mid$=m.

Applicando l'ipotesi induttiva su Q, esistono i polinomi $p_1(V^1),..., p_n(V^n)$ tale che:

(1) $\{\overleftarrow{X} = \overrightarrow{w}\}Q\{\mid X_i \mid \leq \mid w_i \mid + p_i(\mid \overrightarrow{w^i} \mid)\}$ i=1,…,n

Poiché $\xrightarrow{P}$ è irriflessiva, allora anche $\xrightarrow{Q}$ lo è. Questo implica che $\xrightarrow{Q}$ definisce un ordine sulle variabili $\vec{X}$ ($X_1 \xrightarrow{Q} X_2 \xrightarrow{Q} \ldots \xrightarrow{Q} X_m$); perciò, possiamo procedere per side induction sull'ordinamento indotto da $\xrightarrow{Q}$ mostrando che esistono i polinomi $q_1(m,V^1),\ldots,q_n(m,V^n)$ tali che $\forall m,\vec{w}$

(2) $\left\{\vec{X}=\vec{w}\right\}Q^m\left\{|X_i| \leq |w_i| + q_i(m,|\vec{w}^i|)\right\}$ $\forall$ i=1…n

dove Q^m indica la sequenza Q;…;Q (m volte Q).

Dimostrazione per side induction su (*).

PASSO BASE: In Q c'è solo la variabile X_i; questo implica che X_i non è controllata da nessuna variabile e che $V^i = \varnothing$. L'esecuzione di m volte di Q prevede:

$\left\{\vec{X}=\vec{w}\right\}Q^m\{|X_i| \leq |w_i| + m \cdot p_i\}$.

Fissato un i tra 1 ed n, dopo l'esecuzione di P, nello stack X_i, ci sarà il suo contenuto iniziale incrementato del numero di push "p_i" (in questo caso è un polinomio costante) in Q, moltiplicato per la costante "m" (=numero di volte che Q si ripete).

PASSO INDUTTIVO: In Q ci sono $X_1,\ldots,X_n$.

Consideriamo il caso in cui $V^i = \{X_{i1},...,X_{il}\}$ con l ≥ 1 e supponiamo che vi sia il seguente ordinamento tra le variabili: $X_{i1} \to X_{i2} \to ... \to X_{il} \to X_i$; quindi, le variabili che controllano X_i la precedono nella sequenza. Applicando l'ipotesi induttiva su $X_{i1}...X_{il}$, mostriamo che esistono i polinomi $r_{i1}(m,V^{i1}),...,r_{il}(m,V^{il})$ tali che $\forall m,\vec{w}$

$$(3)\quad \{\overleftarrow{X} = \vec{w}\}Q^m\left\{|X_{ij}| \le |w_{ij}| + r_{ij}(m,|\vec{w}^{ij}|)\right\} \text{ per j=1…l,}$$

dove V^{ij} indicale variabili che controllano X_{ij} in Q. Osserviamo che $X_{ij} \notin V^{ij}$ per j=1…l e che $X_i \notin V^{i1} \cup V^{i2} \cup ... \cup V^{il} \subseteq V^i$. Quindi, dobbiamo dimostrare per induzione su m che per $\forall m,\vec{w}$

$$(4)\ \{\overleftarrow{X} = \vec{w}\}Q^m\left\{|X_i| \le |w_i| + m \cdot p_i(|w_{i1}| + r_{i1}(m,|\vec{w}^{i1}|),...,|w_{il}| + r_{il}(m,|\vec{w}^{il}|))\right\}.$$

Supposta vera (4) per m, dimostriamo per m+1:

$$\{\overleftarrow{X} = \vec{w}\}Q^m\{X_{i1} = u_{i1},...,X_{il} = u_{il}, X_i = v_i,...\}$$

$$\{X_{i1} = u_{i1},...,X_{il} = u_{il}, X_i = v_i,...\}Q\{X_i = v_i^*\}.$$

Utilizzando le diverse ipotesi induttive, otteniamo le seguenti relazioni tra le lunghezze degli stack:

$$|v_i^*| \leq |v_i| + p_i(|u_{i1}|,...,|u_{il}|) \qquad \text{per la (1)}$$

$$|v_i| \leq |w_i| + m \cdot p_i(...,|w_{ij}| + r_{ij}(m,|\vec{w}^{ij}|),...) \qquad \text{dall'I.H. su m}$$

$$|u_{ij}| \leq |w_{ij}| + r_{ij}(m,|\vec{w}^{ij}|) \quad \text{per j=1,…,l} \qquad \text{per la (3)}$$

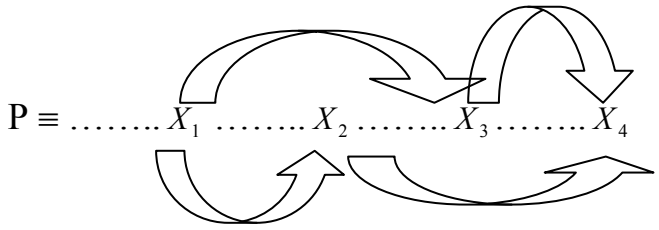
Combinando questi risultati e utilizzando la monotonicità dei polinomi, si ottiene che:

$$|v_i^*| \leq |w_i| + (m+1) \cdot p_i(...,|w_{ij}| + r_{ij}(m+1,|\vec{w}^{ij}|),...)\,. \qquad \square$$

Mostriamo un esempio per spiegare cosa si intende con l'insieme V^i. Supponiamo che P sia costituito dalle seguenti istruzioni:

$$P := \forall X_1[..push(a, X_2); push(a, X_3)]...\forall X_2[push(a, X_3); push(a, X_4)]...$$

Analizzando P dal punto di vista dei controlli tra la variabili:



Ciascuna V^i rappresenta l'insieme delle variabili che controllano la variabile che ha indice "i".

- $V^1 = \varnothing$ indica che la variabile X_1 (i = 1) non è controllata da nessuna variabile;

- $V^2 = \{X_1\}$ indica che X_2 (i = 2) è controllata da X_1;
- $V^3 = \{X_1, X_2\}$ indica X_3 (i = 3) è controllata da X_1 e X_2;
- $V^4 = \{X_2\}$ indica che X_4 (i = 4) è controllata da X_2.

Quindi, P lascia invariato $X_1; X_2$ aumenta in funzione di $X_1; X_3$ aumenta in funzione di X_1 e di $X_2; X_4$ aumenta in funzione di X_2:

- X_1 resta inalterato;
- X_2 avrà dimensione calcolata con la formula:

$$\{| X_1 |= k\} P\{| X_1 |= k \wedge | X_2 |=| X_2 | + c \cdot k\}$$

dove c $\in$ N rappresenta il numero di push presenti nel ciclo $\forall X_1$, comandi che saranno ripetuti k volte poiché $| X_1 |= k$;

- X_3, invece, avrà dimensione:

$$\{| X_1 |= k \wedge | X_2 |= h\} P\{| X_1 |= k \wedge ....\}$$

$$\{...\wedge | X_2 |=| X_2 | + c \cdot | X_1 | \wedge | X_3 |=| X_3 | + c_1 \cdot | X_2 |=\}$$
$$\{=| X_3 | + c_1 \cdot (| X_2 | + c \cdot | X_1 |)\}$$

dove c_1 è il numero di push effettuate su X_3;

- X_4, infine, avrà dimensione:

$$\{| X_2 |= h\} P\{| X_2 |= h \wedge | X_4 |=| X_4 | + c_2 \cdot | X_2 |=| X_4 | + c_2 \cdot h\}$$

dove c_2 è il numero di push effettuate su X_4.

L'esecuzione di P prevede l'incremento degli stack perché per ipotesi P è un core program, quindi è irriflessivo e l'assenza di top circle fa si che P sia eseguibile in tempo polinomiale.

Corollario 2.9 *Per ogni core program P con $\mu(P)=0$ e variabili $\vec{X} := X_1,...,X_n$ esistono i polinomi $p_1(\vec{X}),...,p_n(\vec{X})$ tali che per ogni $\vec{w} := w_1,...,w_n$:*

$$\{\vec{X} = \vec{w}\}P\{|\vec{X}_1| \leq p_1(|w|),...,|\vec{X_n}| \leq p_n(|w|)\}$$

Se P calcola una funzione f, allora f avrà tempo di esecuzione polinomiale, cioè esiste, un polinomio p che soddisfa $|f(\vec{w})| \leq p(|\vec{w}|)$.

Definizione 2.10 *Per ogni stack-program P e Q tali che $V(P)=\{X_1,...,X_k\}$ e $V(Q)=V(P)\cup\{Y_1...Y_l\}$, Q è un* **limite di lunghezza** *per P, e si denota con P<<Q, se:*

$$\begin{cases} \{\vec{X} = \vec{w}\}P\{\vec{X} = \vec{v}\} \\ \{\vec{X} = \vec{w} \wedge \vec{Y} = \vec{u}\}P\{\vec{X} = \vec{v}'\} \end{cases} \quad \Rightarrow \quad |\vec{v}| \leq |\vec{v}'|$$

Vediamo un esempio di Q come limite di lunghezza per P:

$\{X_1 = 5\}P\{X_1 = 7\}$ dopo l'esecuzione di P

$\{X_1 = 5, Y_1 = 4\}Q\{X_1 = 50\}$ dopo l'esecuzione di Q

Definizione 2.11 *Un ciclo foreach X[Q] con μ(foreach X[Q])=n+1 è* **semplice** *se μ(Q)=n. Un core program* $P:\equiv P_1 \ldots P_k$ *di misura μ(P)=n+1 è chiamato* **flattened out** *se ogni sottoprogramma* P_i *è semplice o* $\mu(P_i) \leq n$.

Vediamo alcuni esempi con P:≡ ∀X [Q]:

- μ(P)=7 μ(Q)=3 P è flattened out
- μ(P)=7 μ(Q)=7 non ci sono circoli, P non è flattened out
- μ(P)=7 μ(Q)=6 P è flattened out, Q è un ciclo semplice

Dato un core program P con μ(P)=n+1, si vuole costruire un programma core flattened out P′ con μ(P′)=n+1 ∍′ P′ sia un limite per P. È sufficiente trasformare tutti i cicli non semplici di P in cicli semplici in P′.

A tal proposito si introduce il concetto di profondità d'annidamento **deg(P)**, per i core program P, che assume i valori:

- deg(push(a,X)):= 0;
- $\deg(P_1; P_2) := \max\{\deg(P_1), \deg(P_2)\}$;
- deg(foreach X[Q]):= 1 + deg(Q).

Definizione 2.12 *Il rank di un core program, denotato con* **rk(P)***, è definito induttivamente con:*

- rk(push(a,X)):= 0 $\quad \forall$ lettera a $\in \Sigma \ \wedge$ variabile X;
- rk($P_1; P_2$):= $\max\{rk(P_1), rk(P_2)\}$;
- if P è un loop foreach X[Q] e se μ(P)=n+1, allora

$$rk(P):=\begin{cases} 0 & P \text{ è un ciclo semplice o } \mu(P) \le n \\ 1 & Q \text{ è un ciclo con } \mu(Q)=n+1 \\ 1+\Sigma_{i\le k}\deg(Q_i) & Q \text{ è una sequenza } Q_1 \ldots Q_k \text{ senza top circle e } \mu(Q)=n+1 \end{cases}$$

Lemma 2.13 (Rank zero)

Ogni core program P di misura n+1 e rank 0 è flattened out.

Dimostrazione per induzione sulla struttura del core program P con μ(P)=n+1 e rank 0.

Se P è una sequenza $P_1; P_2$ entrambi i sottoprogrammi P_i hanno rank 0 e almeno 1 ha μ = n+1; quindi l'enunciato deriva dalle ipotesi induttive sui sottoprogrammi di μ = n+1. Se P è un ciclo di μ(P) = n+1 e rank 0, allora questo ciclo è semplice per definizione, quindi P è flattened out.

□

Lemma 2.14 (Riduzione Rank)

Per ogni core program P:≡foreach X[Q] con $\mu(P)=n+1$ e rank >0 esiste un core program P′ tale che:

$$P<<P', \quad \mu(P')=n+1 \quad e \quad \mu(P)=n+1$$

Lemma 2.15 (Flattening)

Per ogni core program P di misura $\mu=n+1$, si può trovare un core program P′ flattened out di misura $\mu=n+1$ che soddisfa : $P<<P'$.

Teorema 2.16 (Limite)

Per ogni funzione f calcolata da uno stack-program P di misura $\mu=n$ esiste una funzione $b \in \varepsilon^{n+2}$ tale che $| f(\vec{w}) | \leq b | (\vec{w}) |$.

Dimostrazione è sufficiente verificare il teorema per i core program, poichè ogni stack-program P ha un core program P^* tale che $\mu(P) = \mu(P^*)$ e $Q<<P^*$ per ogni sottoprogramma Q di P.

Sia P^* ottenuto rimpiazzando tutte le occorrenze di comandi imperativi nil(X) o pop(X) in P con foreach X[push(b,V)] in P^*, e tutte le istruzioni

condizionali if top(X) ≡ a[Q] con foreach X[push(b,V)]; Q^*, con b lettera appartenente all'alfabeto Σ e V una nuova variabile.

Procediamo per induzione su n dimostrando, appunto, il teorema sui core program.

PASSO BASE: n = 0 deriva dal Corollario 2.9.

PASSO INDUTTIVO: supponendo che la tesi sia vera per n, dimostriamo per n+1. Sia P un arbitrario core program di misura n+1, ricorrendo al lemma 2.15 otteniamo un core program P′ nella forma $P_1;...;P_k$ dove ogni componente P_i è un ciclo semplice o $\mu(P_i)$ ≤ n e tale che P<< P′ e μ(P′) = n+1. Dall'ipotesi induttiva e per la chiusura di ε^{n+3} rispetto alla composizione, è sufficiente mostrare che ciascuna funzione calcolata da un ciclo semplice P_i richiede un tempo di esecuzione massimo tale da classificarla come appartenente ad ε^{n+3}.

Sia P_i:≡ foreach X[Q] un ciclo semplice. Quindi μ(Q) = n e dall'ipotesi induttiva, ciascuna funzione h_j calcolata da Q ha tempo di esecuzione limite $b_j \in \varepsilon^{n+2}$. Consideriamo un numero c > 0 tale che $b_j(\vec{x}) \leq E^c_{n+1}(\max(\vec{x}))$ per ogni limite b_j. Sia f_1 una funzione calcolata da P_i, allora, f_1, insieme a tutte le altre funzioni $f_2,...,f_m$

calcolate da P_i, può essere definita per ricorsione simultanea di vettori mediante le funzioni calcolate da Q, nel modo seguente:

$$\begin{cases} f_k(\varepsilon, \vec{w}) = w_{ki} \\ f_k(va, \vec{w}) = h_k(v, w, f_1(v, \vec{w}), ..., f_m(v, \vec{w})) \quad \text{per k = 1,…,m} \end{cases}$$

Segue per induzione su |v| che $|f_k(v, \vec{w})| \leq E_{n+1}^{c\cdot|v|}(\max(v, \vec{w}))$. Inoltre $E_{n+1}^{t}(x) \leq E_{n+2}(x+t)$ e max, la somma appartiene a ε^2, questo è il motivo per cui si ottiene che il tempo limite di esecuzione per f_1 è una funzione della classe ε^{n+3}. □

1.4 Caratterizzazione del Teorema del Limite per gli stack-program

Fissato un alfabeto qualsiasi Σ, una funzione su Σ^* è una qualsiasi k-esima funzione così definita $f : (\Sigma^*)^n \to \Sigma^*$. L'obiettivo di questo paragrafo è dimostrare che le funzioni su Σ^* calcolabili mediante uno stack-program P di misura μ = n sono le funzioni calcolabili da una Machina di Turing in tempo $b\,|(\vec{w})|$ con $b \in \varepsilon^{n+2}$. Mentre, gli stack-program di misura μ = 0 calcolano le funzioni eseguibili in tempo polinomiale.

Definizione 2.17 Per n ≥ 0, si indica con L^n la classe di tutte le funzioni f su Σ^* calcolabili con uno stack-program di misura n.

$$L^n = \{\ f \mid f \text{ è calcolabile da P, } \mu(P) = n\ \}$$

Definizione 2.18 Per n ≥ 0, si indica con G^n la classe di tutte le funzioni f su Σ^* calcolabili con una Macchina di Turing in tempo $b\,|(\vec{w})|$, per qualsiasi $b \in \varepsilon^n$.

$$G^n = \{\ f \mid f \text{ è calcolabile da MT in tempo } b\,|(\vec{w})|,\ b \in \varepsilon^n\ \}$$

Osserviamo che G^2 è la classe FP delle funzioni calcolabili in tempo polinomiale.

Il lemma successivo introduce la sequenza LE[n+1], ossia un programma di misura μ = n che permette di calcolare i rami della funzione di Achermann.

Lemma 2.19 *Per ogni n ≥ 0 si può trovare una sequenza* LE[n+1] *con un top circle tale che:*

$$\mu(LE[n+1]) = n \quad e \quad \{ Y = w \}\ LE[n+1]\ \{ |Y| = E_{n+1}(|w|) \}$$

Teorema 2.20 (Caratterizzazione)

Per n ≥ 0: $L^n = G^{n+2} \Leftrightarrow L^n \subseteq G^{n+2}$ **(1)**

$G^{n+2} \subseteq L^n$ **(2)**

Spieghiamo il significato delle due inclusioni:

(1) indica che considerato un qualsiasi stack-program con μ = n esiste una Macchina di Turing che può simularlo in tempo $b\,|(\vec{w})|$, con $b \in \varepsilon^{n+2}$;

(2) indica che considerata una Macchina di Turing con tempo di esecuzione in ε^{n+2}, esiste uno stack-program di misura μ = n che la simula.

Dimostrazione (1)

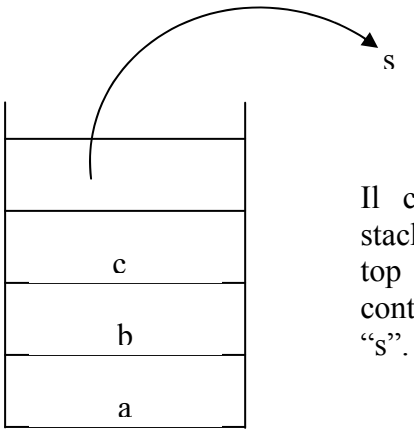
Sia P uno stack-program di misura μ(P) = n. Introduciamo delle notazioni:

$TIMEp(\vec{w})$: indica il numero di passi relativi all'esecuzione di P su input $\vec{w}$, dove un passo è l'esecuzione di un comando imperativo imp(X).

$q_{time}(n)$: indica il tempo impiegato da una Macchina di Turing per simulare un comando imperativo.

Vediamo come i comandi imperativi degli stack-program possono essere simulati mediante una MT $\boldsymbol{M} = \langle \Gamma, b, Q, q_0, F, \delta \rangle$**:**

- **pop(X)**



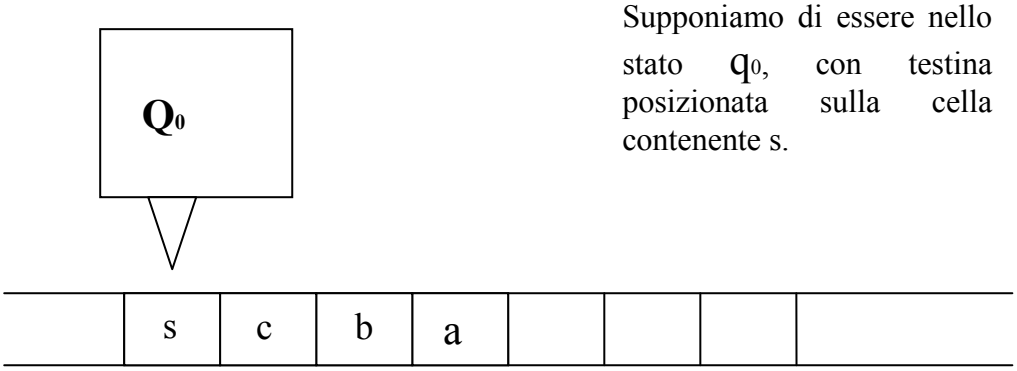
Stack X

Il comando pop(X), negli stack-program, estrae dal top dello stack la lettera contenuta, in questo caso "s".

Per realizzare la simulazione del comando pop(X) con una ***M***, si considera un nastro illimitato, in cui si dispone il contenuto di X. La

disposizione delle celle deve essere tale da garantire che il top di X corrisponda sempre al simbolo nella cella puntata dalla testina.

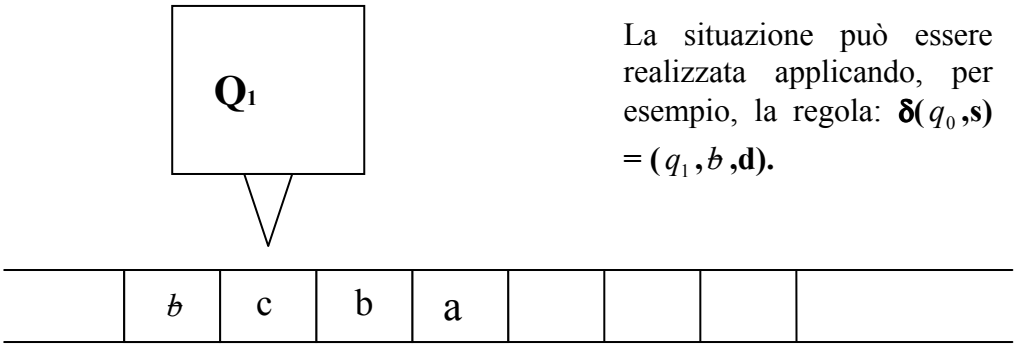
Prima:



Supponiamo di essere nello stato q_0, con testina posizionata sulla cella contenente s.

Figura 2.20.1 La configurazione iniziale è q_0 s c b a

Dopo:

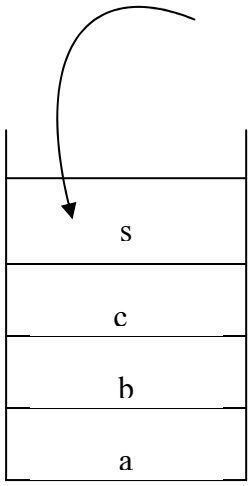


La situazione può essere realizzata applicando, per esempio, la regola: **$\delta(q_0$,s) = (q_1, b, d).**

Figura 2.20.2 La configurazione successiva è b q_1 c b a

La MT che simula la pop cancella il valore della cella puntata dalla testina ed effettua il passaggio di stato con o senza spostamento della testina.

- **push(a,X)**

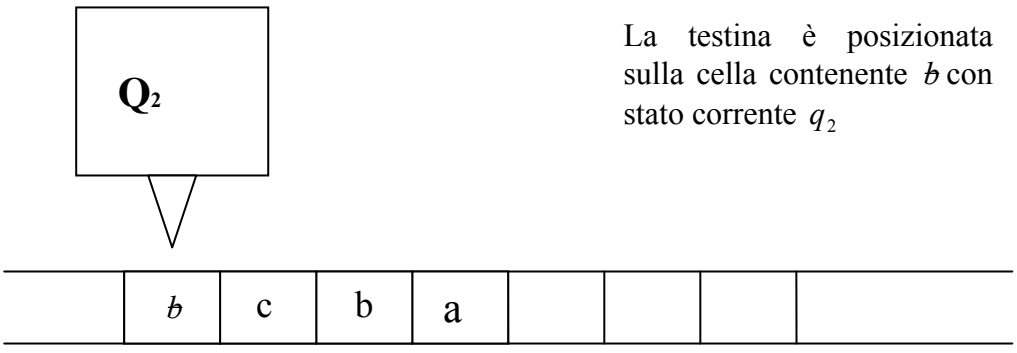


Stack X

Il comando push(a,X), negli stack-program, introduce nel top dello stack una nuova lettera in questo caso "s".

Per simulare la push, la MT deve trovarsi nella seguente situazione:

Prima:

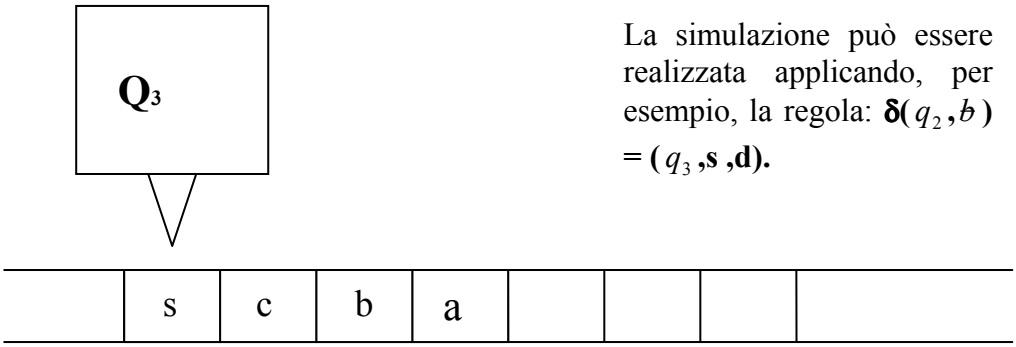


La testina è posizionata sulla cella contenente $\not{b}$ con stato corrente q_2

Figura 2.20.3 La configurazione iniziale è q_2 $\not{b}$ c b a

La testina deve essere posizionata sul primo b a sinistra del contenuto di X; poi, si può inserire la lettera, e transitare in un altro stato e/o spostare la testina.

Dopo:

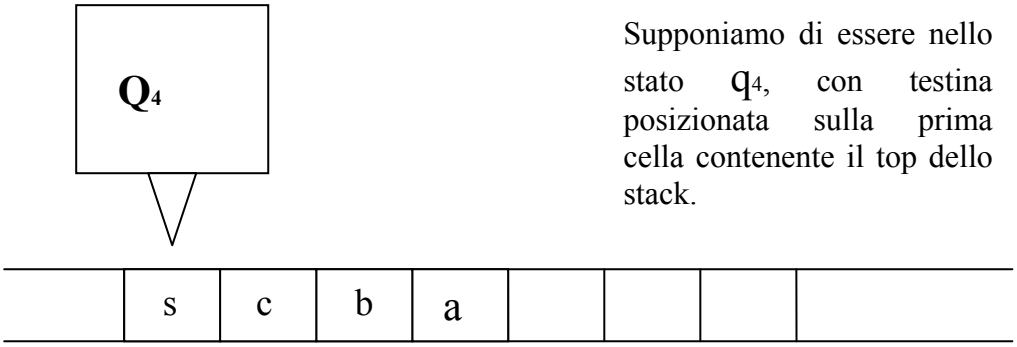


La simulazione può essere realizzata applicando, per esempio, la regola: **δ(q_2 ,b) = (q_3 ,s ,d).**

Figura 2.20.4 La configurazione finale è q_3 s c b a

- **nil(X)** prevede di azzerare X, quindi la ***M*** dovrà cancellare tutta la sequenza coincidente con il contenuto di X.

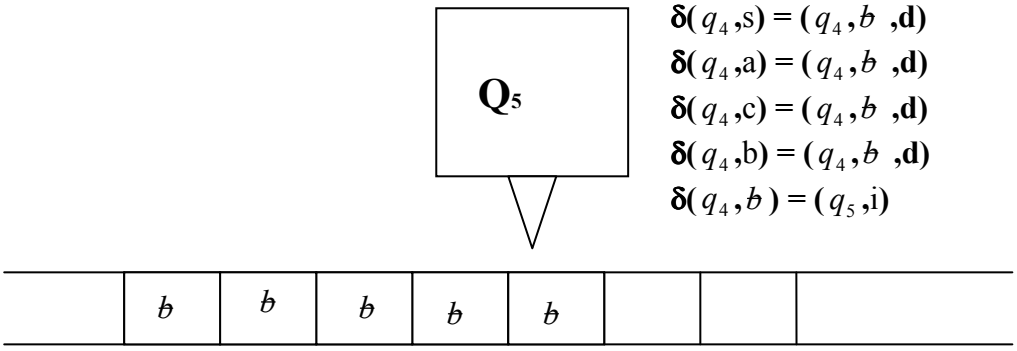
Prima:



Supponiamo di essere nello stato q_4, con testina posizionata sulla prima cella contenente il top dello stack.

Figura 2.20.5 La configurazione finale è q_4 s c b a

Dopo:



La simulazione può essere realizzata, applicando, per esempio, le regole seguenti:

$\delta(q_4, s) = (q_4, b, d)$
$\delta(q_4, a) = (q_4, b, d)$
$\delta(q_4, c) = (q_4, b, d)$
$\delta(q_4, b) = (q_4, b, d)$
$\delta(q_4, b) = (q_5, i)$

Figura 2.20.6 Nastro vuoto

Quindi ***M*** per simulare nil(X), deve posizionare la testina all'estremità della sequenza e scandirla, annullando il contenuto di ogni cella, finché non avrà incontrato la prima cella vuota.Così come è possibile realizzare le simulazioni dei comandi imperativi, è possibile anche la sequenza e i cicli; sarà sufficiente introdurre tanti nastri quante sono le singole componenti da simulare.

Proseguiamo con la dimostrazione introducendo:

- V una nuova variabile;
- $a \in \Sigma$;
- P# che si ottiene a partire da P rimpiazzando ogni comando imperativo "imp" con "imp;push(a,V)".

Il programma P# ha le stesse istruzioni di P:calcolerà la stessa funzione; ed è in grado di contare anche il numero delle istruzioni eseguite, incrementando con una push la dimensione di V.

Definito il programma TIME(P):≡ nil(V); P# e dedotto che μ = n = μ(P), si ha che:

$$\{\overrightarrow{X} = \vec{w}\}TIME(P)\{\vec{V} \models TIME_p(w)\}.$$

Supponendo che inizialmente X contenga la parola w, dopo l'esecuzione di TIME(P) V avrà cardinalità uguale al numero di comandi imp(X) eseguiti da P# che coincidono con quelli di P. Si può applicare il teorema del limite perché l'ipotesi richiesta è verificata, ossia la misura μ(P) = n e, quindi, esiste un limite $b(\vec{w}) \in \varepsilon^{n+2}$ per la funzione calcolata da P. Si ha che:

$$\{\overrightarrow{X} = \vec{w}\}TIME(P)\{\vec{V} \leq b(\vec{w}) \in \varepsilon^{n+2}\}$$

Quindi, esiste una MT che simula P su input $\vec{w}$ in tempo $q_{time}(b(|\vec{w}|)) \cdot b(|\vec{w}|)$, dove $q_{time}(b(|\vec{w}|))$ rappresenta il tempo d'esecuzione impiegato da una MT per simulare un comando imperativo nel caso peggiore, il tutto moltiplicato per $b(|\vec{w}|)$.

Dimostrazione (2)

Sia data la MT $\boldsymbol{M} = \langle \Gamma, b, Q, q_0, F, \delta \rangle$, con la condizione che, su un input w, essa esegue la computazione in tempo $b(|w|)$, per qualsiasi $b \in \varepsilon^{n+2}$. Dimostrare che la funzione f_M calcolata da $\boldsymbol{M}$, può essere calcolata con uno stack-program P su $\Delta := Q \cup \Gamma \cup \{L, N, R\}$ di misura $\mu(P) = n$, dove $\Gamma := \{a_1, ..., a_k\}$. Assumendo che la funzione di transizione δ sia costituita da l mosse $(mossa_1, ..., mossa_n)$ ciascuna composta da: $mossa_i := (q_i, a_i, q_i', a_1', D_i)$, con $D_i = \{L, N, R\}$, dobbiamo costruire il programma P di misura $\mu = n$ che sia in grado di computare f_M, ponendo il risultato nella variabile O, cioè P deve soddisfare:

$$\{\vec{X} = \vec{w}\} P \{O = f_M(w)\}.$$

P avrà la seguente struttura:

P:≡ COMPUTE-TIME-BOUND(Y); con $\mu = n$

INITIALISE (L,Z,R); con $\mu = 0$

Foreach Y[SIMULATE-MOVES]; con $\mu = 0$

OUTPUT(R;O); con $\mu = 0$

e poichè μ(P) = n allora $f_M \in L^n$ = { f | f è calcolabile da P, μ(P)=n }.

Per comodità, si denomina la prima parte di P in questo modo:

INIT { COMPUTE-TIME-BOUND(Y);
INITIALISE (L,Z,R);

La simulazione di ***M*** mediante P prevede che per ogni configurazione α(q,a)β di MT, ottenuta a partire da w dopo m passi (transizioni), cioè $init_M(w) \vdash \alpha(q,a)\beta$, ci sia:

{X = w} INIT; SIMULATE-$MOVES^m$ {L=α, riverse(R)= αβ, Z = q}.

Quindi, dopo aver effettuato l'inserimento nello stack X di w, il primo passo prevede di eseguire INIT. Ricordando che: se b(x) $\in \varepsilon^{n+2}$, allora c'è una costante c tale che b(x) ≤ $E^c_{n+1}(x)$; inoltre, esiste uno stack-program LE[n+1] di misura n tale che { Y = w } LE[n+1] { |Y| = $E_{n+1}(|w|)$ }. Quindi, si può scrivere:

COMPUTE-TIME-BOUND(Y):≡ nil(Y); foreach X[push(a,Y)];

$$\underbrace{LE[n+1];\ldots;LE[n+1]}_{\text{c volte}}$$

Questa sequenza permette di individuare il limite per Y; infatti, dopo aver copiato X in Y, si itera per c volte il programma LE[n+1], in modo che sia soddisfatta:

$\{X = w\}$ COMPUTE-TIME-BOUND(Y)$\{X = w, |Y| = E^{C}_{N+1}(|w|)\}$.

Quindi, μ(COMPUTE-TIME-BOUND) = n; infatti, ciascun LE[n+1] ha μ = n e poiché non ci sono top circle la μ totale resta inalterata.

Si prosegue con INITIALISE(L, Z, R) che corrisponde alla sequenza:

INITIALISE(L, Z, R):≡ nil(L); set(Z, q_0); REVERSE(X; R).

Quindi, lo stack L sarà inizialmente azzerrato, ed in esso, successivamente, disporremo i valori contenuti nelle celle a sinistra della testina. Z sarà inizializzato con lo stato iniziale q_0, e in genere durante l'esecuzione conterrà lo stato corrente. Infine, REVERSE(X; R) è lo stack-program che soddisfa:

$\{X = w\}$ REVERSE(X; R) $\{X = w, R = \text{reverse}(w)\}$

cioè, inverte l'ordine degli elementi di X, quindi il top di X in R sarà il primo elemento ad essere inserito e l'ultimo ad essere estratto.

Passiamo, adesso, ad analizzare SIMULATE-MOVES.

Nota: le istruzioni condizionali if X≡ ε[Q] ∧ if X≡ ε[$\not{Q}$] di misura μ(Q) possono essere così esplicitate:

1) if X≡ ε[Q]:≡ nil(U); push(a,U); foreach X[pop(U)]; if top(U) ≡ a[Q]

2) if X$\not\equiv$ ε[Q]:≡ nil(U); push(a,U); foreach X[pop(U)]; if top(U)≡ ε [Q],

dove la sequenza di istruzioni 1) e 2), rappresentano i passi necessari per controllare se X è vuoto oppure no; soltanto se la condizione è soddisfatta si passa ad eseguire il corpo di Q. In ogni caso, si ricorre allo stack U in cui si dispone una lettera, appartenente all'alfabeto Γ, attraverso push(a,U), e poi si itera l'istruzione pop(U) tante volte quanto vale la cardinalità di X. Se X è vuoto (caso (1)), la pop non è eseguita poiché non c'è iterazione e quindi il controllo finale su U con if top(U) ≡ a[Q] risulta soddisfatto poiché lo stack contiene solo la "a" iniziale. Invece se X non è vuoto l'istruzione pop è iterata un certo numero di volte, e al termine l'istruzione if top(U) ≡ ε [Q] è soddisfatta.

Introduciamo anche altre istruzioni che costituiscono delle sequenze, ovvero:

- **set(U,a)** che sostituisce nil(U); push(a,U)
- **settop(U,a)** che sostituisce pop(U); push(a,U);
- **push(top(L),R)** che sostituisce if top(L) ≡ $a_1[push(a_1, R)]$;...

…;if top(L)≡ $a_k[push(a_k, R)]$.

SIMULATE-MOVES è nella forma $MOVE_1 \ldots MOVE_n$ dove $MOVE_i$ simula $move_i$. Costruiamo tale stack-program, distinguendo tre casi diversi per $MOVE_i$ a seconda se $D_i = \{L, N, R\}$, dove:

(R) indica che la funzione di transizione in una Macchina di Turing ha provocato lo spostamento della testina verso destra;

(L) indica che la funzione di transizione in una Macchina di Turing ha provocato lo spostamento della testina verso sinistra;

(N) indica che la funzione di transizione in una Macchina di Turing non ha provocato alcuno spostamento della testina.

In base a questa distinzione, analizziamo i tre differenti casi:

if top(Z) ≡ q_i [if top(R) ≡ a_i [push($a_i^{'}$, L); set(Z, $q_i^{'}$); pop(R);

if R ≡ ε [push(B, R)]];

Mostriamo la simulazione della MT attraverso un esempio. Supponiamo che la MT sia nella condizione di poter passare da una configurazione alla successiva applicando la i-esima mossa.

Prima:

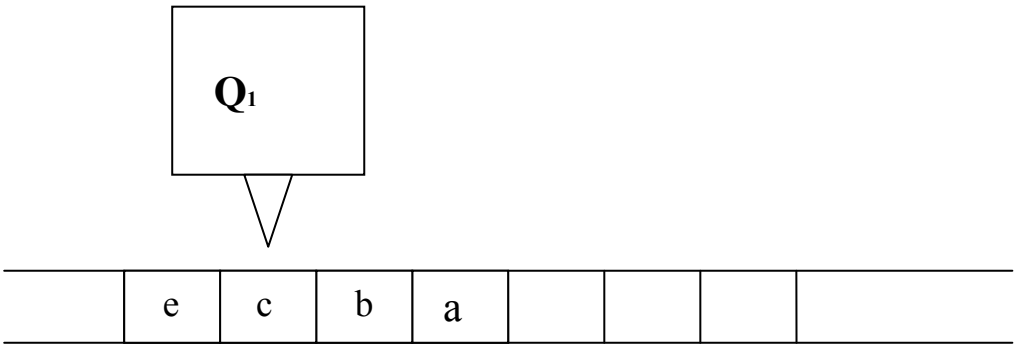


Figura 2.20.7 La configurazione iniziale è e q_1 c b a

La mossa i-esima in MT permette di passare dallo stato q_1, con la testina posizionata sulla cella contenente c, allo stato q_2, sostituendo il simbolo corrente con b, e sostando la testina a destra; cioè:

$$mossa_i := (q_1, c, q_2, b, R)$$

Dopo:

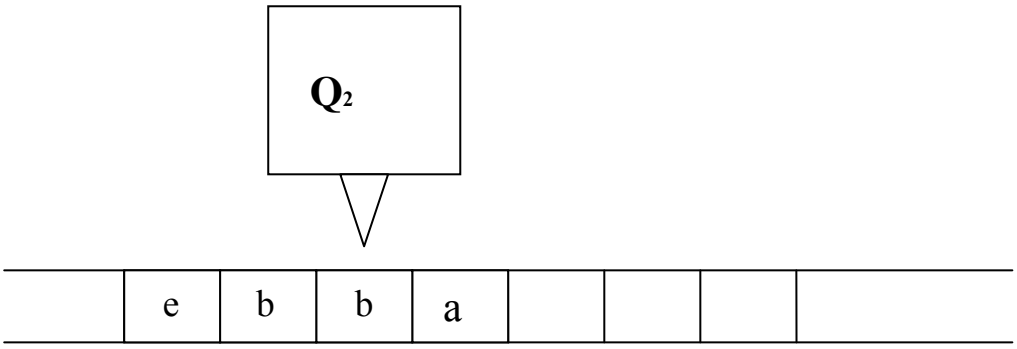
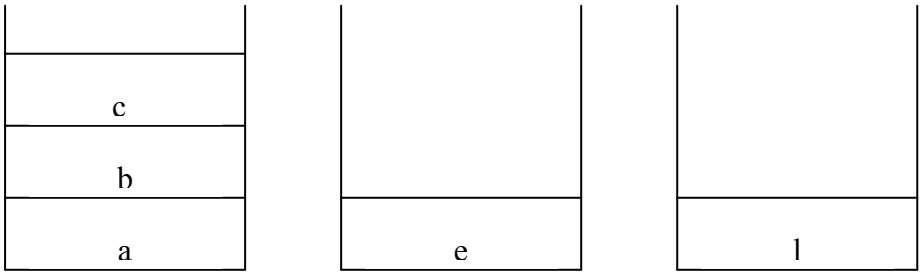


Figura 2.20.8 La configurazione è e b q_2 b a

Per realizzare la simulazione della mossa i-esima si utilizza MOVEi.

È necessario ricostruire le condizioni di partenza relative alla MT.



Stack R

Contiene i valori delle celle a destra, partendo da quella puntata dalla testina

Stack L

Tiene traccia del contenuto delle celle a sx della testina

Stack Z

Contiene lo stato corrente in MT

Figura 2.20.9 Rappresentazione della configurazione e q_1 c b a mediante gli stack

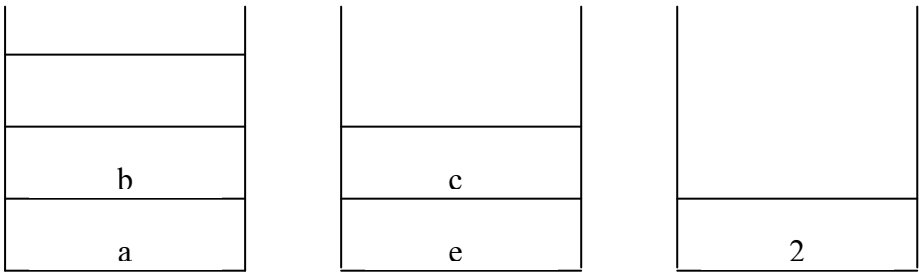
Analizziamo le istruzioni del programma, una dopo l'altra, per comprendere come la simulazione viene realizzata nel primo caso.

Il primo passo da compiere è verificare se "top(Z) ≡ q_1 and top(R) ≡ c"; tale controllo è fondamentale, infatti la MT può utilizzare un certo numero di mosse a cui è associato uno stack-program $MOVE_i$ specifico. Per poter selezionare quello effettivamente richiesto è necessario che le pre-condizioni sugli stack Z ed R siano contemporaneamente soddisfatte. Supponendo che l'esito sia positivo si può passare alla sequenza di istruzioni:

- **push(b, L)** prevede di salvare il nuovo valore "b" nello stack L. Il valore "b" rappresenta la lettera che la funzione di transizione in MT ha sostituito al valore corrente nella cella puntata da q_1. Quindi, poiché al termine della i-esima mossa, la testina avrà subito uno spostamento verso destra rispetto alla cella di partenza (come in figura 2.20.8), è necessario inserire "b" in L, tra i valori a sinistra del corrente, e non in R il cui top è sempre il contenuto della cella corrente puntata dalla testina;

- **set(Z, q_2)** prevede l'aggiornamento dello stack Z con il nuovo stato corrente q_2 da sostituire a q_1;
- **pop(R)** estrae il top dello stack R, ossia "c" per poter passare ad esaminare il prossimo valore.

Dopo l'esecuzione dello stack-program MOVE$_i$ abbiamo:



Stack X
Contiene i valori delle celle a destra, partendo da quella puntata dalla testina

Stack L
Tiene traccia del contenuto delle celle a sinistra della testina

Stack Z
Contiene lo stato corrente in MT

Figura 2.20.10 Rappresentazione della configurazione e b q_2 b a mediante gli stack

Invece, l'ultima istruzione condizionale if R ≡ ε [push(B, R)] controlla se lo stack R è vuoto ed in tal caso inserisce il carattere "B".

Passiamo ad analizzare il secondo caso:

(L) if top(Z) ≡ q_i [if top(R) ≡ a_i [settop(R, $a_i^{'}$); set(Z, $q_i^{'}$);

if L ≡ ε [push(B, R)];

if L≠ε[push(top(L), R); pop(L)]]];

Ripresentiamo la stessa situazione iniziale della figura 2.20.7, con la differenza che la mossa i-esima in MT prevede di passare dallo stato q_1 con testina posizionata sulla cella contenente c, allo stato q_3, sostituendo il simbolo corrente con b, e infine spostando la testina a sinistra:

$$mossa_i := (q_1, c, q_3, b, L)$$

Dopo:

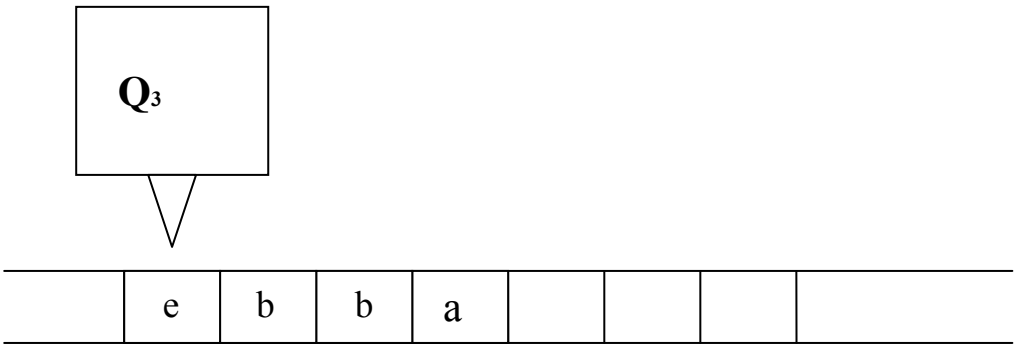


Figura 2.20.11 La configurazione iniziale è q_3 e b b a

Per realizzare la simulazione della mossa i-esima,nel secondo caso, si ricorre sempre allo stack-program $MOVE_i$, che lavora sulle variabili Z, L, R. Supponendo che gli stack siano stati inizializzati, analizziamo le istruzioni del programma per comprendere come raggiungere lo scopo.

Anche in questo caso, il primo passo da compiere è verificare se "top(Z) ≡ q_1 and top(R) ≡ c", e solo se queste condizioni sono contemporaneamente soddisfatte si può passare ad applicare la seguente sequenza di istruzioni:

- **settop(R, b)** propone di simulare la riscrittura del valore "c" corrente nella cella puntata da q_1 con "b", in MT. Ciò viene eseguito effettuando, in ordine, una pop(R) per estrarre il top "c" da R, e poi per l'inserimento di "b" una push(b, R);
- **set(Z, q_3)** prevede l'aggiornamento dello stack Z con il nuovo stato corrente q_3 da sostituire a q_1.

Poiché dobbiamo simulare lo spostamento della testina verso sinistra, lo stack su cui si devono apportare delle modifiche è proprio L, che contiene i valori delle celle immediatamente a sinistra di

quella corrente, il cui top dovrà essere estratto per diventare il top di R (il simbolo su cui in MT è posizionata la testina dopo la i-esima mossa). Prima, però, è necessario effettuare dei controlli su L in modo da poter operare in modo differente a seconda se le celle a sinistra contengono solo blank (cioè se il nastro è vuoto) oppure no.

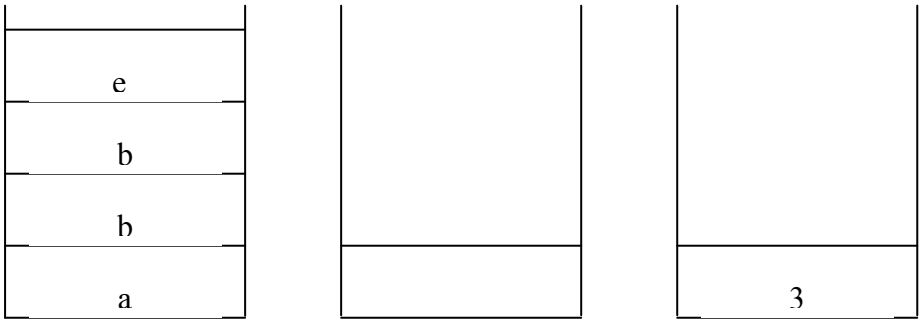
La prima istruzione condizionale che incontriamo è:

if L ≡ ε [push(B, R)], la quale controlla se lo stack R è vuoto ed in tal caso inserisce il carattere "B", altrimenti si passa all'istruzione successiva if L$\not\equiv$ ε [push(top(L), R); pop(L)], cioè se lo stack L non è vuoto allora la condizione ha esito positivo, e ciò indica che a sinistra ci sono celle con valori diversi dal blank sul nastro della MT. Quindi si può eseguire il corpo dell'if, dove la prima istruzione sostituisce una sequenza di istruzioni condizionali che si propongono di controllare se il top di L $\in \Gamma$:

push(top(L),R):=if top(L)$\equiv a_1[push(a_1, R)];\ldots;$if top(L)$\equiv a_k[push(a_k, R)]$

e solo se almeno una condizione è verificata, mediante una push il top di L lo si inserisce in R e lo si estrae da L con pop(L).

Dopo l'esecuzione dello stack-program MOVE$_i$:



Stack X	**Stack L**	**Stack Z**
Contiene i valori delle celle a destra, partendo da quella puntata dalla testina	Tiene traccia del contenuto delle celle a sinistra della testina	Contiene lo stato corrente in MT

Figura 2.20.12 Rappresentazione della configurazione q_3 e b b a mediante gli stack

Infine, consideriamo l'ultimo dei tre casi:

(N) settop(R, a_i'); set(Z, q_i').

Ripresentiamo, ancora una volta, la situazione iniziale della figura 2.20.7, con la differenza che la mossa i-esima in MT prevede di modificare il contenuto della cella corrente "c" in "b" e di portare lo stato da q_1 a q_4, lasciando invariata la posizione della testina:

$$mossa_i := (q_1, c, q_4, b, N)$$

Dopo la mossa$_i$, la MT si presenterà nella seguente situazione:

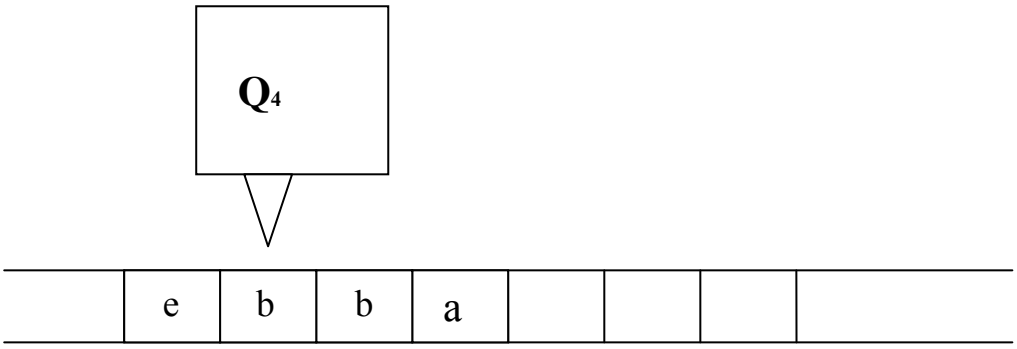


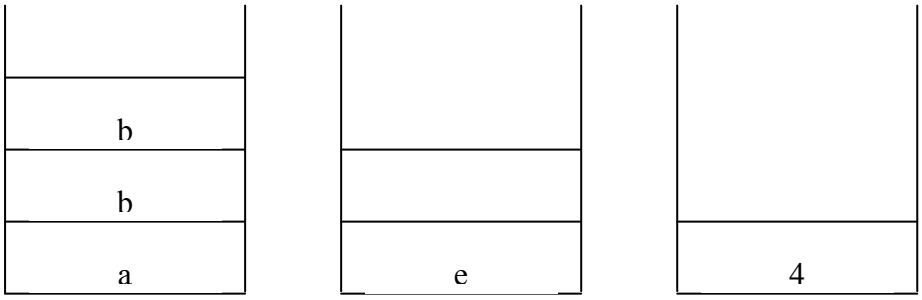
Figura 2.20.13 La configurazione è e q_4 b b a

Per realizzare la simulazione della mossa i-esima,nel terzo caso, si ricorre allo stack-program MOVE$_i$, che lavora sulle variabili Z, R, infatti il contenuto di L resta immutato dato che non c'è spostamento. Supponendo che inizialmente la situazione sia quella illustrata in figura 2.20.9, analizziamo le istruzioni del programma:

- **settop(R, b)** propone di simulare la riscrittura del valore corrente "c" nella cella puntata da q_1 con "b", nella Macchina di Turing. Ciò viene eseguito effettuando in ordine una pop(R), per estrarre il top "c" da R, e poi si passa all'inserimento di "b" mediante una push(b, R);

- **set(Z,** q_4**)** prevede l'aggiornamento dello stack Z con il nuovo stato corrente q_4 da sostituire a q_1.

Dopo l'esecuzione dello stack-program MOVEi:



Stack X	**Stack L**	**Stack Z**
Contiene i valori delle celle a destra, partendo da quella puntata dalla testina	Tiene traccia del contenuto delle celle a sinistra della testina	Contiene lo stato corrente in MT

Figura 2.20.14 Rappresentazione della configurazione e q_4 **b b a mediante gli stack**

Passiamo ad implementare OUTPUT(R; O), che legge dallo stack R il risultato della funzione calcolata e lo dispone nell'apposita variabile O, questo perché lo stack R contiene la parola w $\in \Sigma$ invertita.

OUTPUT(R; O):≡ nil(O), set(Z, a);

foreach R[if top(R) $\in \Delta\backslash\Sigma$ [nil(Z)];

if top(R) $\in \Sigma$ [if top(Z) ≡ a[push(top(R), O)]]]

Valutiamo le varie istruzioni:

- **nil(O)** azzerra lo stack O;
- **set(Z, a)** prevede l'aggiornamento dello stack Z con il nuovo stato corrente della MT;
- **foreach R[if top(R) ∈ Δ\Σ [nil(Z)]** è una sequenza di istruzioni che si apre con un ciclo esterno, che sarà ripetuto un numero di volte pari alla dimensione di R. Il corpo, invece, è costituito da un'istruzione condizionale che controlla se il corrente top di R appartiene a Δ\Σ, in tal caso si può annullare il contenuto di Z.
- Altrimenti si prosegue con:

  if top(R) ∈ Σ [if top(Z) ≡ a[push(top(R), O)]]], dove l'istruzione condizionale più esterna controlla se il top di R è una lettera appartenente a Σ, se tale condizione è verificata si passa al controllo del valore contenuto nel top dello stack Z, mediante il secondo if, il quale se coincide con "a" indica che il risultato della funzione calcolata può essere spostato dallo stack R nella variabile □

CAPITOLO III

1. Un miglioramento della misura μ per gli stack-program

1.1 La misura σ negli stack-program

Ci sono numerosi modi per migliorare la misura μ, poiché è un problema indecidibile se una funzione eseguita da un dato stack-program si trova oppure no in una data classe di complessità. Ricordiamo che, se uno stack Z è modificato in un ciclo controllato da uno stack Y e, poi, Y viene modificato in un ciclo controllato da Z, allora, questo tipo di circolo chiuso prende il nome di "top circle", e quando appare in un ciclo esterno, è prodotta un'esplosione nella complessità computazionale del programma.

Osserviamo, ad esempio, il seguente programma:

$$P := \forall X[\forall Y[push(a,Z)];nil(X);\forall Z[push(a,Y)]],$$

dove, notiamo che, all'interno del loop esterno ∀ X, Y controlla Z (Y→Z), e nel ciclo successivo, Z controlla Y (Z→Y); ovvero, la relazione di controllo ha creato un ciclo chiuso (Y→Z→Y), denominato, appunto "top circle".

Abbiamo introdotto nel capitolo precedente la μ-measure che è un criterio sintattico utile per individuare top circle; infatti, ogni volta che questo appare nel corpo di un ciclo, la μ-measure viene incrementata di uno.

L'obiettivo di questo paragrafo è quello di introdurre e di definire un miglioramento per la misura μ (la σ-measure), partendo dalla seguente considerazione: un programma la cui struttura conduce la μ-measure ad essere uguale ad n vuol dire che contiene n top circle annidati, e questo implica, per il teorema del limite, che esiste una funzione b $\in \varepsilon^{n+2}$ (la n+2-esima classe di Grzegorczyk), dove b rappresenta un limite superiore alla lunghezza del programma.

Supponiamo ora che alcune delle sequenze di pop e push iterate nel programma principale lasciano invariato lo spazio utilizzato complessivamente. Parliamo, cioè, di programmi "not increasing", i quali poiché possono essere iterati senza determinare nessuna crescita nello spazio, allora, l'effettivo spazio utilizzato sarà minore di quello ottenuto con la μ-measure, e potrà essere valutato dalla misura alternativa σ.

Possiamo, quindi, distinguere i cicli in due tipi:

- increasing, determinano un aumento delle dimensioni degli stack coinvolti nel ciclo stesso;
- not increasing, lasciano invariata la dimensione complessiva degli stack.

Presentiamo gli stack-program P_1 e P_2, il primo contenente un top circle increasing e il secondo una un top circle not increasing.

Valutiamo il primo programma P_1:

P_1 : ≡ nil(Y); push(a,Y); nil(Z); push(a,Z) ;

∀X[nil(Z); ∀Y[push(a,Z) ; push(a,Z)] ;

nil(Y); ∀Z[push(a,Y)]]

Supposto |X| = 3, osserviamo che prima del top circle, la dimensione iniziale degli stack coinvolti nel ciclo sarà: |Y| = 1 e |Z| = 1; mentre, alla fine, il risultato ottenuto sarà: |Y| = 8 e |Z| = 8. P_1 contiene, ovviamente, un top circle increasing, poiché quest'ultimo determina un aumento delle dimensioni degli stack coinvolti nel ciclo stesso.

Passiamo, adesso, ad analizzare il secondo programma P_2:

P_2 : ≡ nil(Y); push(a,Y); nil(Z); push(a,Z) ;

∀X[nil(Z); ∀Y[push(a,Z)] ;

nil(Y); ∀Z[push(a,Y)]]

Supposto |X| = 3, notiamo che, prima del ciclo esterno ∀ X, la situazione iniziale sarà: |Y| = 1 e |Z| = 1. Possiamo affermare che, il top circle presente in P_2 è not increasing, poiché lascia invariata la dimensione degli stack; infatti, il risultato ottenuto sarà sempre: |Y| = 1 e |Z| = 1.

In definitiva, possiamo affermare che mentre μ cresce ogni volta che un top circle appare nel corpo di un ciclo, σ aumenta soltanto per i top circle increasing non considerando, di conseguenza, i cicli not increasing. Osserviamo che la μ-measure cresce anche se

annidiamo delle istruzioni o dei cicli che non modificano lo spazio complessivo e che non comportano una crescita nella complessità della funzione calcolata. In altre parole, la nuova misura σ non prende in considerazione tutte le situazioni in cui le operazioni sono eseguite, e il loro bilancio complessivo è negativo.

Nel resto del capitolo, indicheremo con "imp(Y)" un comando imperativo, quale ad esempio pop(Y), push(a,Y) oppure nil(Y); mentre, con "mod($\overline{X}$)" indichiamo un modificatore, cioè una sequenza di comandi imperativi che operano sulle variabili che appaiono in $\overline{X}$.

Introduciamo, adesso, una definizione leggermente modificata di ciclo che risulta essere più adeguata per la nuova misura σ.

Definizione 3.1 *Sia Q una sequenza di sottoprogrammi* $Q_1;...;Q_l$. *C' è un ciclo in Q se esiste una sequenza di variabili* $Z_1;Z_2...;Z_l$ *e una permutazione* π *di* $\{1,...,l\}$ *tale che* $Z_1 \xrightarrow{Q_{\pi(1)}} Z_2 \xrightarrow{Q_{\pi(2)}} ...Z_l \xrightarrow{Q_{\pi(l)}} Z_1$. *Sappiamo che i sottoprogrammi* $Q_1;...;Q_l$ *e le variabili* $Z_1;...;Z_l$ *sono coinvolti nel ciclo.*

Ricordiamo che una permutazione è una funzione biiettiva p: X → X che rappresenta l'operazione con cui si scambia l'ordine di successione di due o più elementi di una serie ordinata; ovvero, è un modo di combinare n elementi distinti scambiandoli di posizione. Ad esempio, se Q:= $Q_1;Q_2;Q_3$ e abbiamo le variabili $Z_1;Z_2;Z_3$ e una permutazione π di {1,2,3}, allora, anche se cambiamo l'ordine di successione delle variabili, otterremo sempre un ciclo; cioè, avremo:

$$Z_1 \xrightarrow{Q_1} Z_2 \xrightarrow{Q_2} Z_3 \xrightarrow{Q_3} Z_1$$

$$Z_2 \xrightarrow{Q_2} Z_3 \xrightarrow{Q_3} Z_1 \xrightarrow{Q_1} Z_2$$

$$Z_3 \xrightarrow{Q_{31}} Z_1 \xrightarrow{Q_1} Z_2 \xrightarrow{Q_2} Z_3$$

Per ragioni di semplicità, considereremo π(i) = i ottenendo il caso $Z_1 \xrightarrow{Q_1} Z_2 \xrightarrow{Q_2} \ldots Z_l \xrightarrow{Q_l} Z_1$.

Definizione 3.2 *Sia P uno stack-program e Y una variabile. La σ-measure di P rispetto alla variabile Y (indicata con* $\underline{\sigma_Y(P)}$*) è definita induttivamente nel modo seguente (con sg(z)=1 se z ≥ 1, sg(z)=0 altrimenti):*

- $\sigma_Y(\mathrm{mod}(\overline{X})) := sg(\Sigma\hat{\sigma}_Y(imp(Y)))$, $\forall imp(Y) \in mod(\overline{X})$, *dove*

- $\hat{\sigma}_Y(push(a,Y)) := 1;$
- $\hat{\sigma}_Y(pop(Y)) := -1;$
- $\hat{\sigma}_Y(nil(Y)) := -\infty;$
- $\hat{\sigma}_Y(imp(X)) := 0,$ *con* $Y \neq X;$

- σ_Y *(if top* $Z \equiv a[P]$*):=* $\sigma_Y(P)$*;*
- $\sigma_Y(P_1;P_2) := \max(\sigma_Y(P_1),\sigma_Y(P_2)),$ *con* $P_1;P_2$ *che non è un modificatore;*
- σ_Y *(foreach X[Q]):=* $\sigma_Y(Q)+1$*, se esiste un ciclo in Q e un sottoprogramma* Q_i *tale che*
  - *Y e* Q_i *sono coinvolti nel ciclo;*
  - $\sigma_Y(Q) = \sigma_Y(Q_i)$ *;*
  - *il ciclo è increasing;*

  altrimenti σ_Y *(foreach X[Q]):=* $\sigma_Y(Q)$ *se il ciclo è not increasing, cioè, se indicato con* $Q_1;...;Q_l$ *la sequenza di sottoprogrammi e con* $Z_1;...;Z_l$ *le variabili coinvolte nel ciclo, abbiamo che* $\sigma_{Z_i}(Q_j) = 0,$ $\forall$ *i:=1…l e j:=1…l. Se*

le precedenti condizioni non sono verificate, possiamo affermare che il ciclo è increasing.

Notiamo che la σ_Y-measure di un modificatore ($\sigma_Y(\mathrm{mod}(\overline{X})) := sg(\Sigma\hat{\sigma}_Y(imp(Y)))$) è uguale ad 1 soltanto quando, in assenza di comandi nil, il numero complessivo di push su Y è maggiore del numero di pop sulla stessa variabile Y; cioè, si ha soltanto quando si produce una crescita nella lunghezza di Y. Vediamo un semplice esempio di questa situazione:

P:≡ push(a,Y); ∀ X[push(a,Y); push(a,Y)]

Supponiamo che, all'inizio X contenga la parola "abc", cioè |X| = 3. Prima dell'esecuzione dello stack-program P, avremo |Y| = 0; mentre, al termine di P, la dimensione dello stack Y sarà pari a 7.

Nel caso di ciclo increasing, abbiamo che $\sigma_Y(Q) = \sigma_Y(Q_i)$,e questo vuol dire che c'è un'esplosione nella complessità di Y in $\sigma_Y(Q_i)$, che comporta, di conseguenza, una crescita per Q.

Inoltre, analizziamo tre differenti casi in cui si può verificare un ciclo not increasing, cioè, quando si ha σ_Y *(foreach X[Q]):=* σ_Y *(Q)* con $\sigma_{Z_i}(Q_j) = 0$:

1) Non ci sono cicli in cui Y è coinvolto;di conseguenza, la dimensione dello stack Y resta invariata.
2) Y è coinvolto, insieme ad un sottoprogramma Q_i, in un ciclo in Q, ma accade che $\sigma_Y(Q_i)$ è inferiore di $\sigma_Y(Q)$. Questo vuol dire che c'è un'esplosione nella complessità di Y in $\sigma_Y(Q_i)$ ma questa crescita è in ogni caso limitata dalla complessità di Y in un differente sottoprogramma di Q;
3) Si suppone che Y sia coinvolto in qualche ciclo in Q, ma ognuno di questi è un ciclo not increasing; quindi, in accordo alla definizione data precedentemente, ogni variabile Z_i, coinvolta in ogni ciclo, non produce una crescita nella complessità dei sottoprogrammi Q_j coinvolti nello stesso ciclo ($\sigma_{Z_i}(Q_j) = 0$). Da ciò si deduce che lo spazio utilizzato durante l'esecuzione del ciclo esterno foreach X[Q] è fondamentalmente lo stesso utilizzato da Q. Questo non è un fatto sorprendente

perché si può liberamente iterare un programma not increasing senza provocare una crescita dannosa.

In tutti e tre i casi la σ-measure deve restare invariata;mentre, aumenta soltanto quando vi è un top circle e quando, simultaneamente, almeno una delle variabili coinvolte nel top circle causa una crescita nella complessità spaziale del sottoprogramma relativo (cioè, se esiste p tale che $\sigma_{Z_p}(Q_p) > 0$).

Possiamo, a questo punto, introdurre il concetto di σ-measure che permette di calcolare la complessità computazionale degli stack-program, in modo molto più efficiente rispetto alla μ-measure.

Sia dato il programma P:

1. Se P è costituito solamente da comandi imperativi,non esistono, ovviamente, controlli tra le variabili e, quindi, non ci sono circoli. Ricordiamo che, un modificatore "mod($\overline{X}$)" è una sequenza di comandi imperativi che operano sulle variabili che appaiono in $\overline{X}$; allora:

$$\tilde{\sigma}(P) = \tilde{\sigma}(\text{mod}(\overline{X})) := 0;$$

2. Supponiamo che P sia costituito da una sequenza di sottoprogrammmi $Q_1;...;Q_l$;distinguiamo due casi:

   Non esiste nessuna sequenza di controlli tra le variabili dei sottoprogrammi tali da creare un top circle; quindi:

   $$\tilde{\sigma}(P) = \tilde{\sigma}(P_1; P_2) := \max(\sigma_Y(P_1; P_2));$$

   Esiste una relazione di controlli tra le variabili di tutti i sottoprogrammi;si possono verificare due situazioni:

Il circolo creatosi è un top circle increasing, che determina un aumento della complessità computazionale del programma; infatti:

$$\tilde{\sigma}(\text{foreach } X[Q]) := \max(\sigma_Y(\text{foreach } X[Q]))$$

$$= \max(\sigma_Y(Q)+1);$$

Altrimenti, si parlerà di top circle not increasing, che non provoca nessuna crescita nello spazio del programma; quindi:

$$\tilde{\sigma}(\text{foreach } X[Q]) := \max(\sigma_Y(\text{foreach } X[Q]))$$

$$= \max(\sigma_Y(Q)).$$

La differenza tra gli ultimi due casi analizzati, sta nell'influenza che essi hanno sulla complessità degli stack-program; infatti, in P, si

verificherà un incremento nella complessità computazionale soltanto se sarà presente un top circle increasing.

Introduciamo la definizione formale della misura σ rispetto all'intero insieme di variabili che appaiono in uno stack-program.

Definizione 3.3 *Sia P uno stack-program. La σ-measure di P (indicata con σ(P)) è definita nel seguente modo:*

- *Se P è una sequenza di comandi imperativi:*

  $\tilde{\sigma}(P) = \tilde{\sigma}(\mathrm{mod}(\overline{X})) := 0$*;*

- *Se P è un'istruzione condizionale:*

  $\tilde{\sigma}(P) = \tilde{\sigma}$ *(if top Z ≡ a[Q]):= max(*σ_Y *(if top Z ≡ a[Q]))*

  =max(σ_Y *(Q)),*

  per tutte le Y che appaiono in P;

- *Se P è una sequenza di sottoprogrammi:*

  $\tilde{\sigma}(P) = \tilde{\sigma}(P_1; P_2) := \max(\sigma_Y(P_1; P_2)) = \max(\sigma_Y(P_1); \sigma_Y(P_2))$,

  per tutte le Y che appaiono in P, con $P_1; P_2$ *che non è un modificatore;*

- *Se P è un ciclo foreach X[Q]:*

  $\tilde{\sigma}(P) = \tilde{\sigma}$ *(foreach X[Q]):=max(*σ_Y *(foreach X[Q]))*

=max((σ_Y (Q)+1), se è increasing

=max((σ_Y (Q)),se è not increasing,

per tutte le Y che appaiono in P;

- $\sigma(P) := \tilde{\sigma}(P) \dot{-} 1$, *con l'usuale operazione "predecessore".*

Osserviamo che, approssimativamente, usiamo $\hat{\sigma}_Y$ per individuare tutti i modificatori increasing, per una data variabile Y (questo può essere fatto ponendo $\hat{\sigma}_Y$ uguale ad 1). Ricordiamo che, un modificatore è uguale ad 1 soltanto quando, in assenza di comandi nil, il numero complessivo di push su Y è maggiore del numero di pop sempre sullo stesso stack Y. Tuttavia, una volta individuati, non dobbiamo considerarli nella valutazione della σ-measure; questo è il motivo per cui nella definizione precedente si utilizza "$\dot{-}1$".

Notiamo che σ(P) < μ(P), per ogni stack-program P.

1.2 La procedura di Riduzione $\rightsquigarrow$

Definizione 3.4 *Due stack-program P e Q sono equivalenti sullo spazio se* $\{\overline{X} = \vec{w}\}P\{|\overline{X}| = m\}$ *implica che* $\{\overline{X} = \vec{w}\}Q\{|\overline{X}| = O(m)\}$. *Questo è denotato con* $P \approx_S Q$.

Un esempio di $P \approx_S Q$ è:

$\{X_1 = 5\}P\{X_1 = 9\}$

$\{X_1 = 5\}Q\{X_1 = O(9)\}$

Deduciamo che, Q non ha una dimensione superiore a quella di P, a meno di una costante moltiplicativa prefissata.

Definizione di $\rightsquigarrow$: *Sia A uno stack-program tale che $\mu(A)=n+1$ e $\sigma(A)=m$, con $m<n+1$ ($\sigma(A) < \mu(A)$); il programma $\rightsquigarrow A$ è ottenuto nel seguente modo:*

- *Se A = foreach X[R]:*
  - *con $\mu(R)= \sigma(R) = n$, e indicando con $C_1,...,C_l$ i top circle in R, e con $A_{i1},...,A_{ip}$ le variabili coinvolte in C_i, $\forall$ i=1,...,l, abbiamo che $\rightsquigarrow A$ è il risultato del*

cambiamento di ogni imp(A_{ij}) in un nop(A_{ij}), ovvero in un'operazione non imperativa;

- *con $\mu(R) > \sigma(R)$, allora $\rightsquigarrow A$=foreach X[$\rightsquigarrow R$];*

- *Se $A = A_1; A_2$ e :*
  - *se $\max(\mu(A_1), \mu(A_2)) = \mu(A_1)$, abbiamo che $\rightsquigarrow A = \rightsquigarrow A_1; A_2$;*
  - *se $\max(\mu(A_1), \mu(A_2)) = \mu(A_2)$, abbiamo che $\rightsquigarrow A = A_1; \rightsquigarrow A_2$;*
  - *infine, se $\mu(A_1) = \mu(A_2)$, abbiamo che $A = \rightsquigarrow A_1; \rightsquigarrow A_2$;*

- *Se A = if top(X) ≡ a[R], abbiamo che $\rightsquigarrow A$=if top(X) ≡ a[$\rightsquigarrow R$].*

Se il programma A ha μ(A)=n+1 e σ(A)=m, con m<n+1(σ(A)< μ(A)); applicando $\rightsquigarrow$ ad A, avremo che μ($\rightsquigarrow$ A) = σ ($\rightsquigarrow$ A) = m.

Lemma 3.5 *Dato uno stack-program P, con $\mu(P) = n+1$ e $\sigma(P) = n$, esiste uno stack-program $\rightsquigarrow P$ tale che con $\mu(\rightsquigarrow P) = n$, $\sigma(\rightsquigarrow P) = n$, e $P \approx_S \rightsquigarrow P$.*

Dimostrazione: per induzione su n.

PASSO BASE (per n=0): Sia $\mu(P) = 1$ e $\sigma(P) = 0$. Nel caso principale, P è nella forma "foreach X[Q]", con un ciclo not increasing in Q. Applicando $\rightsquigarrow$ a P, otteniamo un programma P′ la cui σ-measure è ancora uguale a 0, e la cui μ-measure è ridotta a 0, poiché $\rightsquigarrow$ ha interrotto il ciclo in P che portava μ da 0 a 1 (infatti, in P′ non ci sono più push sulle variabili coinvolte nel ciclo). Notiamo che P può diminuire la lunghezza, ossia la dimensione, degli stack coinvolti nel ciclo, mentre P′ non esegue alcuna operazione nello stesso ciclo, ma può aumentare le sue variabili soltanto attraverso un fattore lineare; cioè, se $\{\overline{X} = \vec{w}\}P\{|\overline{X}| = m\}$, allora $\{\overline{X} = \vec{w}\}P'\{|\overline{X}| = cm\}$, con c costante dipendente dalla struttura di P, ottenendo, $P \approx_S P'$.

PASSO INDUTTIVO (per n+1): Sia $\mu(P) = n+2$ e $\sigma(P) = n+1$. P sia nella forma "foreach X[Q]", e sia C un top circle che appare in Q, con $\mu(Q) = n+1$. Abbiamo due situazioni differenti:

- $\sigma(Q) = n+1$;
- $\sigma(Q) = n$.

Analizziamo il primo caso ($\mu(Q) = n+1$ e $\sigma(Q) = n+1$).

C è un ciclo not-increasing , poiché è stato individuato da μ, ma non da σ. Applicando la procedura di riduzione $\rightsquigarrow$ a P, otteniamo un programma P′ tale che $\mu(P') = n+1$, $\sigma(P') = n+1$, e $P \approx_S P'$. Notiamo che, la σ-measure resta sempre pari ad n+1, mentre la μ-measure è stata ridotta a n+1.Passiamo ad analizzare il secondo caso ($\mu(Q) = n+1$ e $\sigma(Q) = n$).

C è un ciclo increasing, individuato sia da μ che da σ. Per l'ipotesi induttiva, abbiamo che esiste un programma Q′ tale che $\mu(Q') = n$ e $\sigma(Q') = n$, e $Q \approx_S Q'$. Partendo da P, costruiamo un nuovo programma P′ = foreach X[Q′]. Abbiamo che $\mu(P') = \mu(Q') + 1 = n+1$, $\sigma(P') = \sigma(Q') + 1 = n + 1$, e $P \approx_S P'$.

I casi in cui P è sequenza di sottopragrammi ($P = P_1; P_2; ...; P_k$) o un istruzione condizionale (P = if top(X) ≡ a[P]) possono essere provati in modo analogo alla dimostrazione fatta per P = foreach X[Q]. □

1.3 Il Teorema del Limite con la misura σ

Teorema 3.6 *Per ogni funzione f calcolata da uno stack-program P di misura μ(P) = n e σ(P) = m, esiste una funzione* $b \in \varepsilon^{m+2}$ *tale che* $| f(\vec{w}) | \leq b | (\vec{w}) |$.

Dimostrazione:

Sia K = μ(P) – σ(P) (k = n – m). Applichiamo k volte il lemma precedente 3.5, ottenendo, così, una sequenza P := $P_0; P_1; ...; P_k$ di stack-program tale che ∀ i < k:

$$\mu(P_{i+1}) = \mu(P) - i\,,\ \sigma(P_i) = \sigma(P_{i+1})\,,\ \text{e}\ P_i \approx_S P_{i+1}\,.$$

Per il teorema del limite 2.16 di Kristiansen e Niggl (dimostrato nel capitolo precedente), possiamo affermare che dato lo stack-program P_k con $\sigma(P_k) = m$, esiste una funzione $b \in \varepsilon^{\sigma(P_k)+2} = \varepsilon^{m+2}$ che rappresenta il limite alla lunghezza di P_k. Allo stesso modo, possiamo notare che questo risultato vale anche per lo stack-program P, poiché è derivato dalla transitività di $\approx_S$ ($P_0 \approx_S P_1 \approx_S P_2 \approx_S ... P_K$). □

Abbiamo, come possiamo notare, migliorato la misura degli stack-program poiché σ < μ.

Possiamo affermare, che gli stack-program con σ-measure pari ad n possono essere simulati da una macchina di Turing con complessità temporale in ε^{n+2}. Allo stesso modo, possiamo dire che una Macchina di Turing con complessità temporale in ε^{n+2} può essere simulata da uno stack-program di con σ-measure pari ad n.

BIBLIOGRAFIA

[1] S. J. Bellantoni, S. Cook, *A new recursion-theoretic characterization of the polytime function.* Computational Complexity 2(1992)97-110.

[2] S. J. Bellantoni, *Predicative recursion and computational complexity*, PhD thesis, Toronto, 1992.

[3] S. J. Bellantoni and K. H. Niggl, *Ranking primitive recursion: the low Grzegorczyk classes revisited*, to appear on SIAM J. of Computing.

[4] A. Cobham, *The intrinsic computational difficulty of functions*, Y. Bar-Hillel (ed), Proceedings of the International Conference on Logic, Methodology, and Philosophy of Science, 24-30, North-Holland, Amsterdam, 1962.

[5] L. Colson, D. Fredholm, *System T, call-by-value and the minimum problem*. Theoretical computer science 206 (1998), 301-315.

[6] E. Covino, G. Pani, *Recursive programming languages for complexity classes*, 16th Annual IEEE Symposium on Logic In Computer Science (LICs 2001), June 16-19, 2001, Boston.

[7] E. Covino, G. Pani, *An implicit recursive language for the polynomial time-space complexity classes*, Journal of Universal Computer Science, vol. 8, no. 1(2002).

[8] E. Covino, G. Pani, *Time-space computational complexity of imperative programming language*, 17th Annual IEEE Symposium on Logic In Computer Science (LICs 2002), July 22-25, 2002, Copenhagen.

[9] E. Covino, G. Pani, *Space complexity analysis for stack programs*, Fifth International Workshop in Implicit Computational Complexity (ICC'03), Ottawa.

[10] E. Covino, G. Pani, *A refinement of the μ-measure for stack programs*, Electronic Notes in Theoretical Computer Science 90(1) 37-44(2003).

[11] A. Grzegorczyk, *Some classes of recursion functions*, Rozprawy Mate. IV, Warsaw, 1953.

[12] M. Hofmann, *The strength of non-size-increasing computations*. Principles of Programming Languages, POPL'02, Portland, Oregon, January 16-18th, 2002.

[13] M. Hofmann, *Programming languages captuning complexity classes*. SIGACT News Logic Column 9.

[14] N. Jones, *Program analysis for implicit computational complexity*. Third International Workshop on Implicity Computational Complexity (ICC'01), Aarhus.

[15] N. Jones, *Logspace and Ptime characterized by programming languages*. Theoretical Computer Science 228 (1999) 151-174.

[16] Lars Kristiansen, *New recursion-theoretic characterizations of well-known complexity classes*. Fourth International Workshop on Implicit Computational Complexity (ICC'02), Copenhagen.

[17] Lars Kristiansen and K. H. NIggl, *On the computational complexity of imperative programming languages*. Theoretical computer science, to appear.

[18] Lars Kristiansen and K. H. NIggl, *The gardland measure and computational complexity of imperative programs*. Fifth International Workshop in Implicit Computational Complexity (ICC'03), Ottawa.

[19] D.Leivant, *Ramified recurrence and computational complexity I: word recurrence and polytime*, in Clote and J. Remmel (eds), Feasible Mathematics II (Birkhauser,1994), 320-343.

[20] D.Leivant and J. Y. Marion, *Ramified recurrence and computational complexity II: substitution and polyspace*, in J. Tiuryn and L. Pocholsky (eds), Computer Science Logic, LNCS 933 (1995) 486-500.

[21] D.Leivant, *A foundational delineation of computational feasibility*, Proc. Of the 6th Annual IEEE Symposium on Logic In Computer Science, (IEEE Computer Society Press, 1991), 2-18.

[22] D.Leivant, *Stratified functional programs and computational complexity*, in Conference Records of the 20th Annual ACM Symposium on Principles of Programming Languages, New York, 1993, 325-333.

[23] Karl Heinz Niggl, *Control structures in programs and computational complexity*. Fourth Implicit Computational Complexity Workshop (ICC'02), Copenhagen.

[24] E. Nelson, *Predicative arithmetic*, Princeton University Press, Princeton, 1986.

[25] I. Oitavem, *New recursive characterization of the elementary functions and the functions computable in polynomial space*, Revista Matematica de la Universidad Complutense de Madrid, 10.1 (1997) 109-125.

[26] R. W. Ritchie, *Classes of predictable computable functions*, Transactions of A.M.S., 106, 1963.

[27] H. E. Rose, *Subrecursion: functions and hierarchies*. Oxford University Press, Oxford, 1984.

[28] H. Simmons, *The realm of primitive recursion*, Arch. Math. Logic, 27 (1988) 177-188.