INFORMATICA TEORICA

Luca Cappelletti
Prof. Carlo Merenghetti

6 CFU

2019
Informatica Magistrale
Università degli studi di Milano
Italia
2 febbraio 2020

Indice

Parte I

Calcolabilità

1

Nozioni di base

1.1 Nozioni su funzioni

Definizione 1.1 | Funzione

Dati due insiemi A e B, una **funzione** f di dominio A e codominio B è una legge che ad ogni elemento $a \in A$ associa un elemento $b \in B$.

Definizione 1.2 | Funzione iniettiva

Una funzione f si dice **iniettiva** se e solo se:

$$x_1 = x_2 \Rightarrow x_1 = x_2$$

Una funzione **iniettiva** è una funzione che associa a elementi distinti del dominio elementi distinti del codominio.

Definizione 1.3 | Immagine di funzione

Si dice **immagine** di f l'insieme:

$$\mathrm{Im}_f \equiv \{y \mid \exists x \in A(y = x)\}$$

Definizione 1.4 | Funzione suriettiva

Una funzione si dice **suriettiva** se e solo se la sua immagine coincide con il codominio:

$$\forall b \in B \, \exists a \in A : \; a = b$$

Definizione 1.5 | Funzione biettiva

Una **corrispondenza biunivoca** o **funzione biettiva** è una funzione $f : A \longrightarrow B$ tale che:

$$\forall y \in B \quad \exists! x \in A : y = x$$

Una funzione è **biettiva** se e solo se è sia *iniettiva* che *suriettiva*.

Definizione 1.6 | Funzione inversa

Data una funzione biettiva $f : A \longrightarrow B$ si dice **inversa** di f la funzione $f^{-1} : B \Rightarrow A$ definita da $x = f^{-1}(x)$ se e solo se $y = x$

Definizione 1.7 | Composizione di funzioni

Date due funzioni $f : A \longrightarrow B$, $g : B \longrightarrow C$ in cui il codominio della prima coincide con il codominio della seconda diciamo **composizione**:

$$g \circ f : A \longrightarrow C$$

la funzione definita tramite la legge:

$$(f \circ g)(x) = x$$

Definizione 1.8 | Funzione identità

La funzione $I_A : A \longrightarrow A$ definita da $\forall a (I_A(a) = a)$ si chiama funzione **identità** su A.

Se $f : A \longrightarrow B$ allora $I_B \circ f = f$ o $f \circ I_A = f$. Inoltre, se $f : A \longrightarrow B$ è una corrispondenza biunivoca, allora $f \circ f^{-1} = I_B$ e $f^{-1} \circ f = I_A$.

Definizione 1.9 | Funzione definita e indefinita

Se all'elemento $a \in A$ viene associato un elemento di B, diremo che f è **definita** su a, altrimenti verrà detta **indefinita**.

Definizione 1.10 | Indefinito

Il simbolo utilizzato per indicare **indefinito** è:

$$\perp$$

Definizione 1.11 | Dominio di definizione

Data una funzione $f : A \longrightarrow B$, si dice **Dominio** di definizione:

$$A = \{x \mid x \in A, x \text{ definita}\}$$

Definizione 1.12 | Funzione parziale

Una funzione parziale $f : A \longrightarrow B$ può essere pensata come una funzione:

$$\overline{f} : A \longrightarrow B \cup \{\perp\}$$

dove $\perp$ è un simbolo (distinto) di indefinito:

$$\overline{f}(x) = \begin{cases} x & \text{se definita} \\ \perp & \text{se indefinita} \end{cases}$$

1.2 Nozioni su insiemi

Definizione 1.13 | Coprodotto di insiemi o Unione Disgiunta

Dati due insiemi A e B diciamo **unione disgiunta** di A e B l'insieme $A \sqcup B$ unione di due coppie isomorfe e disgiunte di A e B, cioè:

$$A \sqcup B = \{(a,1) \mid a \in A\} \cup \{(b,2) \mid b \in B\}$$

Le due iniezioni $i : A \longrightarrow A \sqcup B$ e $j : B \longrightarrow A \sqcup B$ sono definite da $i(a) = (a,1)$ e $j(b) = (b,1)$ rispettivamente.

Definizione 1.14 | Prodotto Cartesiano

Dati n insiemi, non vuoti, $A_1, \ldots, A_n$, il prodotto cartesiano:

$$A_1 \times \ldots \times A_n = \{\langle a_1, \ldots, a_n \rangle\}$$

è l'insieme delle n-uple ordinate in cui l'i-esimo elemento è preso dall'i-esimo insieme.

Definizione 1.15 | Insieme delle funzioni

Siano A e B due insiemi non vuoti di elementi. Si definisce insieme delle funzioni da A a B l'insieme:

$$A^B = \{f : A \longrightarrow B \cup \{\bot\}\}$$

delle funzioni (totali o parziali) che hanno A come dominio e B come codominio.

Definizione 1.16 | Insiemi isomorfi

Dati due insiemi A e B, essi sono **isomorfi**, scrivendo $A \simeq B$, se esiste una corrispondenza biunivoca:

$$f : A \longrightarrow B$$

Definizione 1.17 | Cardinalità

Chiameremo la classe quoziente classe della cardinalità. In particolare cardinalità del numerabile è la classe di equivalenza contenente:

$$\mathbb{N} = \{0, 1, 2, \ldots, n, \ldots\}$$

Teorema 1.1 | Teorema di Cantor

L'insieme $\mathbb{N}^{\mathbb{N}}$ delle funzioni da $f : \mathbb{N} \longrightarrow \mathbb{N}$ non è numerabile:

$$\mathbb{N} \not\simeq \mathbb{N}^{\mathbb{N}}$$

In altre parole, $\mathbb{N}$ non è **isomorfo** a $\mathbb{N}^{\mathbb{N}}$: non è possibile enumerare gli elementi all'interno dell'insieme $\mathbb{N}^{\mathbb{N}}$.

Dimostrazione 1.1 | Teorema di Cantor

Per dimostrare che $\mathbb{N}^{\mathbb{N}}$ non è numerabile dovremo provare che non è possibile elencare tutti gli elementi di $\mathbb{N}^{\mathbb{N}}$.

La dimostrazione si sviluppa per assurdo: se $\mathbb{N}^{\mathbb{N}}$ fosse numerabile, dovremmo riuscire ad elencare tutti i suoi elementi costruendo una tabella in cui le righe corrispondono a tutte e sole le funzioni di $\mathbb{N}^{\mathbb{N}}$ e le colonne corrispondono ai numeri naturali:

	0	1	2	…	n
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	…	$f_0(n)$
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	…	$f_1(n)$
⋮	⋮	⋮	⋮	⋮	⋮
f_n	$f_n(0)$	$f_n(1)$	$f_n(2)$	…	$f_n(n)$

La tabella così costruita associa ad ogni $f : \mathbb{N} \longrightarrow \mathbb{N}$ di $\mathbb{N}^{\mathbb{N}}$ e ad ogni intero $n \in \mathbb{N}$ il valore in n di f. Si prenda allora la funzione $g : \mathbb{N} \longrightarrow \mathbb{N}$ definita come:

$$g(n) = f_n(n) + 1$$

È facile verificare che g non coincide con nessuna delle f_i, elencate nella tabella. Infatti, se g fosse f_i per qualche intero i, si avrebbe l'assurdo:

$$f_i(i) = f_i(i) + 1$$

Pertanto $\mathbb{N}^{\mathbb{N}}$ non è numerabile.

Osservazione 1.1 | Osservazione sul teorema di Cantor

Siccome in un sistema digitale i programmi devono essere specificati con una quantità finita di informazione, ogni dato e quindi ogni programma può essere descritto da un intero:

$$\text{Dati} \simeq \mathbb{N}, \quad \text{Programmi} \simeq \mathbb{N}$$

L'insieme delle funzioni computabili da un dato sistema, descritto da:

$$C : \text{Dati} \times \text{Programmi} \longrightarrow \text{Dati} \cup \{\bot\}$$

è allora il set delle funzioni computabili *FC* è equivalente a:

$$FC \equiv \{f \mid f : \mathbb{N} \longrightarrow \mathbb{N} \cup \{\bot\}, \exists P \in \text{Programmi}\,(x = C(x,P))\}$$

Allora:

$$FC \simeq \text{Programmi} \simeq \mathbb{N} \not\simeq (\mathbb{N} \cup \{\bot\})^{\mathbb{N}}$$

Esistono quindi funzioni non computabili, e il problema di caratterizzare la classe di quelle computabili è ben posto.

Teorema 1.2 | Funzioni coppia di Cantor

L'insieme dei dati $\mathbb{N}\times\mathbb{N}$ è **isomorfo** all'insieme dei numeri naturali $\mathbb{N}$ cioè coppie ordinate di interi possono essere poste in corrispondenza biunivoca con gli interi utilizzando la funzione coppia di Cantor:

$$\langle , \rangle : \mathbb{N}\times\mathbb{N} \longrightarrow \mathbb{N}^{+}$$

Dimostrazione 1.2 | Funzioni coppia di Cantor

È sufficiente considerare la seguente tabella:

x \ y	0	1	2	3	4	…
0	1	3	6	10	15	…
1	2	5	9	14	…	…
2	4	8	13	…	…	…
3	7	12	…	…	…	…
4	11	…	…	…	…	…
⋮	…	…	…	…	…	…

È chiaro come, estendendo tale legge, a ogni coppia (x,y) corrisponda un preciso intero positivo $n=\langle x,y\rangle$ e viceversa. Non è difficile ottenere esplicitamente la funzione $\langle x,y\rangle$ come composizione di operazioni aritmetiche, infatti:

Lungo l'asse x è:

$$\langle x,0\rangle = 1+1+2+\ldots+z = 1+\frac{z\cdot(z+1)}{2}$$

Lungo la diagonale è:

$$\langle z-y,y\rangle = \langle z,0\rangle + y = 1+\frac{z\cdot(z+1)}{2}+y$$

Posto $x=z-y$ otteniamo:

$$\langle x,y\rangle = 1+\frac{(x+y)(x+y+1)}{2}+y$$

Corollario 1.1 | Funzioni proiezione

Data la funzione coppia $\langle , \rangle : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}^+$ definita da:

$$\langle x, y \rangle = 1 + \frac{(x+y)(x+y+1)}{2} + y$$

vogliamo costruire le due **funzioni proiezione**:

$$\sin : \mathbb{N}^+ \longrightarrow \mathbb{N} \text{ e } \mathrm{des} : \mathbb{N}^+ \longrightarrow \mathbb{N}$$

tali che per ogni intero $n \geqslant 1$ valga $\langle \sin(n), \mathrm{des}(n) \rangle = n$.

Per come è costruita la funzione coppia, per ogni intero $n \geqslant 1$ esiste un intero $z \geqslant 0$ per cui valgono le seguenti uguaglianze:

$$n = \langle z - \mathrm{des}(n), \mathrm{des}(n) \rangle = 1 + \frac{z \cdot (z+1)}{2} + \mathrm{des}(n)$$

Riprendendo la tabella dalla dimostrazione delle funzioni coppia, notiamo che z corrisponde all'ascissa del primo elemento della diagonale su cui si trova n. In altre parole, z è l'unico intero tale che:

$$\langle z, 0 \rangle \leqslant n < \langle z+1, 0 \rangle$$

da cui:

$$1 + \frac{z \cdot (z+1)}{2} \leqslant n < 1 + \frac{(z+1)(z+2)}{2}$$

Risolvendo per z otteniamo:

$$\sqrt{2n - \frac{7}{4}} - \frac{3}{2} < z \leqslant \sqrt{2n - \frac{7}{4}} - \frac{1}{2}$$

e quindi:

$$z = \left\lfloor \sqrt{2n - \frac{7}{4}} - \frac{1}{2} \right\rfloor$$

Dimostreremo ora che z può essere calcolato mediante la formula:

$$z = \mathrm{round}\left(\sqrt{2n}\right) - 1 = \left\lfloor \sqrt{2n - \frac{7}{4}} - \frac{1}{2} \right\rfloor$$

Supponendo che la relazione sia falsa, si ottengono 3 casi possibili:

1. Esiste un intero k tale che:
$$\left\lfloor \sqrt{2n - \frac{7}{4}} + \frac{1}{2} \right\rfloor \geqslant k+1 \quad \wedge \quad \mathrm{round}\left(\sqrt{2n}\right) \leqslant k$$
Questo implica $\sqrt{2n - \frac{7}{4}} \geqslant k + \frac{1}{2} \quad \wedge \quad \sqrt{2n} < k + \frac{1}{2}$, da cui l'assurdo: $\sqrt{2n} < \sqrt{2n - \frac{7}{4}}$.
2. Esiste un intero k tale che:
$$\left\lfloor \sqrt{2n - \frac{7}{4}} + \frac{1}{2} \right\rfloor \leqslant k \quad \wedge \quad \mathrm{round}\left(\sqrt{2n}\right) \geqslant k+1$$
Questo implica $\sqrt{2n - \frac{7}{4}} < k + \frac{1}{2} \quad \wedge \quad \sqrt{2n} \geqslant k + \frac{1}{2}$. Elevando al quadrato otteniamo:
$$2n < k^2 + k + 2 \quad \wedge \quad 2n \geqslant k^2 + k + \frac{1}{4} \quad \Rightarrow \quad \frac{k^2+k}{2} + \frac{1}{8} \leqslant n < \frac{k^2+k}{2} + 1$$
Siccome $\frac{k^2+k}{2}$ è un intero, è immediato concludere che nessun intero n può soddisfare la relazione e pertanto anche in questo caso abbiamo ottenuto un assurdo.

Le identità sono pertanto dimostrate e sostituendo z otteniamo le formule per il calcolo delle proiezioni:

$$\sin(n) = \mathrm{round}\left(\sqrt{2n}\right) \frac{\mathrm{round}\left(\sqrt{2n}\right) + 1}{2} - n \qquad \mathrm{des}(n) = n - \mathrm{round}\left(\sqrt{2n}\right) \frac{\mathrm{round}\left(\sqrt{2n}\right) + 1}{2}$$

2

Linguaggi di programmazione

Definizione 2.1 | Linguaggio di programmazione (aspetto sintattico)

Un *linguaggio di programmazione* da un punto di vista **sintattico** è un insieme (generalmente infinito) di programmi, dove con "programma" si indica una particolare sequenza finita di simboli di un dato alfabeto.

Definizione 2.2 | Linguaggio di programmazione (aspetto semantico)

Un *linguaggio di programmazione* da un punto di vista **semantico** è un insieme (generalmente infinito) di programmi, dove con "programma" si indica un insieme di regole che trasformano il dato di ingresso nell'eventuale dato di uscita: il significato di un programma è la funzione parziale realizzata.

Definizione 2.3 | Significato di RAM

Con RAM si intende **Random Access Machines**, cioè macchine ad accesso casuale, si tratta di macchine a registri.

Definizione 2.4 | Linguaggio RAM

Si tratta di un meccanismo di calcolo che presenta alcune affinità con quelli reali, ed è usato appunto per le macchine RAM.

Definizione 2.5 | Sintassi del linguaggio RAM ridotto

Un programma RAM (ridotto) è una sequenza finita di istruzioni appartenenti a uno dei seguenti tre tipi:

1. $R_k \leftarrow R_k + 1$
2. $R_k \leftarrow R_k - 1$
3. If $R_k = 0$ then goto n

dove k ed n sono interi non negativi.

Definizione 2.6 | Grammatica delle istruzioni nel linguaggio RAM ridotto

Le istruzioni del linguaggio RAM ridotto hanno il formato seguente:

$$\langle \text{istruzione} \rangle \rightarrow \begin{cases} R_k \leftarrow R_k + 1 \\ R_k \leftarrow R_k - 1 \\ \text{If } R_k = 0 \text{ then goto } n \end{cases}$$

Definizione 2.7 | Grammatica dei programmi nel linguaggio RAM ridotto

I programmi nel linguaggio RAM ridotto hanno il formato seguente:

$$\langle \text{programma} \rangle \rightarrow \begin{cases} \langle \text{istruzione} \rangle \\ \langle \text{istruzione} \rangle ; \langle \text{programma} \rangle \end{cases}$$

Le istruzioni possono essere scritte in sequenza, separandole con il simbolo ";" o alternativamente a capo.

Indichiamo con **Programmi** l'insieme dei programmi per RAM:

$$\begin{aligned} \mathbb{P} &= \text{Funzioni}(\text{RAM}) \\ &= \{f \mid f : \text{Dati} \longrightarrow \text{Dati} \cup \{\perp\}, \exists P \in \text{Programmi}: \ f = \varphi_P\} \end{aligned}$$

Osservazione 2.1 | Registri di una macchina RAM

Una macchina RAM possiede **infiniti** registri: $L, R_0, R_1, \ldots, R_j, \ldots$ ognuno capace di contenere un intero, con L utilizzato come **contatore delle istruzioni** o **program counter**.

Osservazione 2.2 | Tipo dei dati di una macchina RAM

Una macchina RAM utilizza dati di tipo **intero**.

Osservazione 2.3 | Inizializzazione di una macchina RAM

La macchina RAM viene inizializzata ponendo 1 in L e il dato di ingresso x in R_1. Tutti gli altri registri sono azzerati.

Osservazione 2.4 | Esecuzione di una macchina RAM

L'esecuzione viene effettuata dalla **Unità Centrale**, che accede all'informazione contenuta nei registri con *accesso casuale*. Le istruzioni vengono eseguite in sequenza, incrementato L di 1 a meno che l'istruzione di salto condizionato non alteri la sequenzialità.

Osservazione 2.5 | Terminazione di una macchina RAM

Il programma di una macchina RAM termina quando il contenuto di L è 0.

Osservazione 2.6 | Posizione del dato di uscita di una macchina RAM

Il dato di uscita di un programma di una macchina RAM y è il contenuto di R_0.

2.1 Semantica di una macchina RAM

Definizione 2.8 | Stato di una macchina RAM

Con **stato** si intende una funzione $S : \{L, R_0, R_1, R_2, \ldots\} \longrightarrow N$, l'intero $S(R_j)$ è detto contenuto di R_j nello stato S. Intuitivamente, lo stato è una *fotografia* dei contenuti dei registri in un certo istante. Indichiamo con **Stati** l'insieme degli stati.

Definizione 2.9 | Stato finale di una macchina RAM

Lo **stato finale** è uno stato S in cui $S(L) = 0$.

Definizione 2.10 | Dato

Con **Dato** si intende un qualsiasi intero non negativo. Indichiamo con *Dati* l'insieme dei dati $\{0, 1, 2, \ldots, n \ldots\}$.

Definizione 2.11 | Inizializzazione di una macchina RAM

Con **inizializzazione** è la funzione in: Dati $\rightarrow$ Stati definita come segue: $x \mapsto S_{in}$ dove:

$$S_{in}(L) = 1, \quad S_{in}(S_j) = \begin{cases} x & \text{se } j = 1 \\ 0 & \text{altrimenti} \end{cases}$$

Definizione 2.12 | Numero di istruzioni di un programma

Dato un programma P indichiamo il numero di istruzioni del programma con $|P|$.

Definizione 2.13 | Sequenza di computazione

Con **sequenza di computazione** del programma P sul dato x è una successione $\{S_0, S_1, \ldots, S_n\}$ di stati tali che:

1. $S_0 = \text{in}(x)$
2. $S_{e+1} = \delta(S_e, P) \quad (0 \leqslant e < n)$

Osservazione 2.7 | A cosa serve la sintassi?

La sintassi ci ha permesso di definire il linguaggio di programmazione **Programmi**.

Osservazione 2.8 | A cosa serve la semantica?

La semantica ci permette di definire il sistema di programmazione $\{\varphi_P\}$. Essa può essere vista come una funzione del tipo:

$$\text{Semantica} : \text{Programmi} \longrightarrow (\text{Dati} \cup \{\perp\})^{\text{Dati}}$$

L'immagine della funzione è l'insieme delle funzioni che ammetto programmi in grado di computarle.

Definizione 2.14 | Sequenza di computazione accettata

Con **sequenza di computazione accettata** dal programma P sul dato x è una sequenza di computazione $\langle S_0, \ldots, S_n \rangle$ dove S_n è uno stato finale $(S_n(L)) = 0$.

Definizione 2.15 | Stato prossimo

Con lo **stato prossimo** è la funzione (parziale) (detta anche *funzione di transizione di stato*):

$$\delta : \text{Stati} \times \text{Programmi} \longrightarrow Stati$$

dove $S' = \delta(S, P)$ è definito dalla seguente procedura:

Caso $S(L) = 0$ allora S' è indefinito, cioè $S' = \perp$.

Caso $S(L) > |P|$ allora $S'(L) = 0$, $S'(R_j) = S(R_j)$ (con $i \geqslant 0$) gli altri registri vengono non modificati.

Caso $0 < S(L) \leqslant |P|$ allora seleziona l'istruzione $S(L)$ in P diciamo *istr*:

ADD se *istr* è del tipo $R_k \leftarrow R_k + 1$:

$$S'(L) = S(L) + 1$$
$$S'(R_j) = \begin{cases} S(R_j) + 1 & \text{se } j = k \\ S(R_j) & \text{altrimenti} \end{cases}$$

SUB se *istr* è del tipo $R_k \leftarrow R_k - 1$:

$$S'(L) = S(L) + 1$$
$$S'(R_j) = \begin{cases} S(R_j) - 1 & \text{se } j = k \\ S(R_j) & \text{altrimenti} \end{cases}$$

IF se *istr* è del tipo If $R_k = 0$ then goto m:

$$S'(L) = \begin{cases} m & \text{se } S(R_k) = 0 \\ S(L) + 1 & \text{altrimenti} \end{cases}$$
$$S'(R_j) = S(R_j)$$

Definizione 2.16 | Funzione computata

Con **funzione computata** del programma P è la funzione:

$$\varphi_P : \text{Dati} \longrightarrow \text{Dati} \cup \{\perp\}$$

è la composizione delle funzioni (inizializzazione, transizione di stati iterata, estrazione del risultato) definita da:

$$\varphi_P(x) = \begin{cases} y & \text{Se esiste una sequenza di computazione } \langle S_0, \ldots, S_n \rangle \text{ accettata da } P \text{ sul dato } x \text{ ed inoltre } y = S_n(R_0) \\ \perp & \text{altrimenti} \end{cases}$$

La funzione φ_P è **parziale**, essendo possibili sequenze di computazioni infinite. Vedremo che tale fenomeno è inevitabile.

2.2 Linguaggio While

2.2.1 Sintassi del linguaggio While

Definizione 2.17 | Comando di assegnamento in sintassi While

Un **comando di assegnamento** nel linguaggio While è del tipo:

$$x_k = 0, \quad x_k = x_j + 1, \quad x_k = x_j - 1 \qquad (0 \leqslant k, j \leqslant 20)$$

dove x_k e x_j sono dette variabili. Si va a limitare il numero dei registri.

Definizione 2.18 | Comando While

Un **comando while** è un commento del tipo:

$$\text{while } x_k \neq 0 \text{ do } C \qquad (0 \leqslant k \leqslant 20)$$

con C un comando arbitrario.

Definizione 2.19 | Comando composto in sintassi While

Un **comando composto** nel linguaggio While è del tipo:

$$\text{begin } C_1; C_2; \ldots; C_m \text{ end} \qquad (0 < m)$$

essendo $C_1, \ldots, C_m$ comandi arbitrari.

Definizione 2.20 | Programma While

Un **programma while** è un comando composto. Indichiamo con **W-Programmi** l'insieme dei programmi while:

$$\begin{aligned}\mathbb{W} &= \text{Funzioni}(\text{while}) \\ &= \{g \mid g : \text{Dati} \longrightarrow \text{Dati} \cup \{\perp\}, \exists P \in \text{W-Programmi } (g = \psi_P)\}\end{aligned}$$

2.2.2 Grammatica del linguaggio While

Definizione 2.21 | Grammatica di programma in linguaggio While

La grammatica di un **programma** in linguaggio While è:

$$\langle \text{programma} \rangle \to \langle \text{comando composto} \rangle$$

Definizione 2.22 | Grammatica di comando in linguaggio While

La grammatica di un **comando** in linguaggio While è:

$$\langle \text{com.} \rangle \to \langle \text{assegnamento} \rangle / \langle \text{com. while} \rangle / \langle \text{com. composto} \rangle$$

Definizione 2.23 | Grammatica di assegnamento in linguaggio While

La grammatica di un **assegnamento** in linguaggio While è:

$$\langle \text{assegn.} \rangle \to x_k = 0 / x_k = x_j + 1 / x_k = x_j - 1 \qquad (0 \leqslant k, j \leqslant 20)$$

Definizione 2.24 | Grammatica di comando while in linguaggio While

La grammatica di un **comando while** in linguaggio While è:

$$\langle \text{com. while} \rangle \to \text{while } x_k \neq 0 \text{ do } \langle \text{com.} \rangle \qquad (0 \leqslant k \leqslant 20)$$

Definizione 2.25 | Grammatica di sequenza di comandi in linguaggio While

La grammatica di una **sequenza di comandi** in linguaggio While è:

$$\langle \text{sequenza di com.} \rangle \to \langle \text{com.} \rangle / \langle \text{com.} \rangle \langle \text{sequenza di com.} \rangle$$

Definizione 2.26 | Grammatica di comando composto in linguaggio While

La grammatica di una **sequenza di comandi** in linguaggio While è:

$$\langle \text{com. composto} \rangle \to \text{begin } \langle \text{sequenza di com.} \rangle \text{ end}$$

Osservazione 2.9 | Come dimostrare che una proprietà P valga per tutti i programmi

Per poter dimostrare che una proprietà P vale per tutti i programmi basterà dimostrare due proprietà:

1. La proprietà P vale per tutti i comandi di assegnamento.
2. Se la proprietà P vale per C, allora vale per:

$$C' \equiv \text{while } x_k \neq 0 \text{ do } C$$

Se P vale per $C_1, \ldots, C_m$, allora vale per:

$$C' \equiv \text{begin } C_1; \ldots; C_m \text{ end}$$

2.2.3 Semantica operazionale

Definizione 2.27 | Stato nel linguaggio While

Nel linguaggio while, uno **stato** è una funzione:

$$\underline{x} : \{x_0, x_1, \ldots, x_{20}\} \longrightarrow \mathbb{N}$$

Indichiamo con $w-\text{Stati}$ l'insieme degli stati N^{21}.

Osservazione 2.10 | Differenze tra registri nel linguaggio RAM e While

A differenza dell'analoga nozione per il linguaggio RAM, non abbiamo necessità di introdurre il registro L, in quanto il nostro linguaggio while non ha istruzioni di salto. La limitazione a soli 21 registri di memoria è per semplicità e, come vedremo, non limita la capacità computazionale.

Definizione 2.28 | Dato nel linguaggio While

Nel linguaggio while, un **dato** è un qualsiasi intero non negativo.

Definizione 2.29 | Inizializzazione nel linguaggio while

Con **inizializzazione** è la funzione:

$$w-\text{in} : \text{Dati} \longrightarrow w-\text{Stati}$$

dove:

$$w-\text{in}(x) = (0, x_1, 0, 0, \ldots, 0)$$

Definizione 2.30 | Semantica nel linguaggio While

Nel linguaggio while la **semantica** del programma P è la funzione:

$$\psi_P : \text{Dati} \longrightarrow \text{Dati} \cup \{\perp\}$$

dove: $\psi_P(x) = \text{Componente di indice } 0 \text{ in } [\![P]\!](w-\text{in}(x))$.

Diremo anche che ψ_P è la funzione calcolata dal programma P, e $\{\psi_P\}$ il sistema di programmazione del linguaggio while.

Definizione 2.31 | Semantica nel linguaggio while

La **semantica** dei comandi è una funzione parziale, detta funzione di transizione di stato:

$$[\![\ ]\!]() : \text{Comandi} \times w-\text{Stati} \longrightarrow w-\text{Stati}$$

Sia C un comando, $\underline{x}$ uno stato: definiamo $\underline{y} = [\![C]\!](\underline{x})$ (intuitivamente, il risultato del comando C sullo stato $\underline{x}$) per ricorrenza.

Caso $C \equiv x_k = 0$: $y_j = \begin{cases} 0 & \text{Se } k = j \\ x_j & \text{Altrimenti} \end{cases}$

Caso $C \equiv x_k = x_i + 1$: $y_j = \begin{cases} x_i + 1 & \text{Se } k = j \\ x_j & \text{Altrimenti} \end{cases}$

Caso $C \equiv x_k = x_i - 1$: $y_j = \begin{cases} x_i - 1 & \text{Se } k = j \\ x_j & \text{Altrimenti} \end{cases}$

Caso $C \equiv$ **begin** $C_1; \ldots; C_m$ **end:** $[\![C]\!](\underline{x}) = [\![C_m]\!] \circ [\![C_{m-1}]\!] \circ \ldots \circ [\![C_1]\!](\underline{x})$, dove $\circ$ è la composizione fra funzione. Si va a invertire l'indice perché avviene la composizione di funzioni.

Caso $C \equiv$ **while** $x_k \neq 0$ **do** C_1**:**

$$[\![C]\!](\underline{x}) = \begin{cases} [\![C_1]\!]^{\mu_e(\text{componente } k\text{-esima in } [\![C_1]\!]^e(\underline{x}) = 0)}(\underline{x}) \\ \circ \perp \end{cases}$$

Questo comando introduce l'indefinito perché $x_k = 0$ può non essere verificata. In particolare la notazione significa:

$$[\![C_1]\!]^e = [\![C_1]\!] \circ [\![C_1]\!] \circ \ldots \circ [\![C_1]\!] \quad (e \text{ volte})$$

Con μ_e intendiamo il primo intero non negativo tale che la k-esima componente della funzione $[\![C_1]\!]^e$ valutata sull'argomento $\underline{x}$ risulta 0. Se questo non succede mai, conveniamo di porre $[\![C]\!](\underline{x}) = \perp$ (indefinito) convenendo inoltre che $[\![C]\!] \circ \perp = \perp \circ [\![C]\!] = \perp$.

Siamo ora in grado di associare ad ogni comando C una funzione parziale $[\![C]\!] : w-\text{Stati} \longrightarrow w-\text{Stati}$: essa esegue tutte le istruzioni in un blocco unico. Non c'è una sequenza di stati data dalla funzione δ.

Poiché un programma P è in particolare un comando (comando composto), associamo a P la funzione parziale:

$$[\![P]\!] : w-\text{Stati} \longrightarrow w-\text{Stati}$$

2.3 Compilatore: da W-Programmi a Programmi RAM

Definizione 2.32 | Traduzione

Siano L_1 e L_2 due linguaggi di programmazione, con rispettivi sistemi di programmazione $\{\psi_{P_1}\}$ e $\{\varphi_{P_2}\}$.

Una **traduzione** è una funzione $T : L_1 \longrightarrow L_2$ tale che:

1. T è una funzione totale calcolabile in PASCAL: questa condizione dice che T deve essere effettivamente calcolabile.
2. T mantiene la semantica inalterata, cioè $\forall P \in L_1$: $\psi_P = \varphi_{T(P)}$

Definizione 2.33 | Traduzione guidata della sintassi

Se il linguaggio è definito induttivamente dalla sintassi, è spesso possibile descrivere la traduzione T in termini ricorsivi sulla costruzione del linguaggio: si parla in tal caso di **traduzione guidata della sintassi**.

Osservazione 2.11 | A cose serve una traduzione?

Chiamato *Funzioni*(L_i) l'insieme delle funzioni computabili in L_i, è chiaro che se esiste una traduzione $T : L_1 \longrightarrow L_2$ allora:

$$\text{Funzioni}\,(L_1) \subseteq \text{Funzioni}\,(L_2)$$

la costruzione di una traduzione (programma compilatore) è allora uno strumento per il nostro problema di confronto tra classi di funzioni.

Teorema 2.1 | Correttezza del compilatore

Procediamo a costruire una traduzione Conf : W – Programmi $\longrightarrow$ Programmi. Sia C un comando qualsiasi, allora:

Caso $C \equiv x_k = 0$**:** $\text{Comp}(C) = \begin{cases} \text{LOOP}: \begin{cases} R_k \leftarrow R_k - 1 \\ \text{IF } R_k = 0 \text{ THEN GOTO EXIT} \\ \text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \end{cases} & \text{Il registro } R_{21} \text{ è sempre a zero, pertanto la condizione risulta sempre vera.} \\ \text{EXIT}: \ R_k \leftarrow R_k - 1 & \text{Si tratta di un'istruzione qualsiasi che non ha nessuna funzione.} \end{cases}$

Caso $C \equiv x_k = x_j \pm 1$ **con** $(k \neq j)$**:** $\text{Comp}(C) = \begin{cases} \underbrace{\text{LOOP}}_{\text{Azzera } R_k}: \begin{cases} R_k \leftarrow R_k - 1 \\ \text{IF } R_k = 0 \text{ THEN GOTO EXIT } 1 \\ \text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \end{cases} \\ \underbrace{\text{EXIT } 1}_{\text{Copio } R_j \text{ in } R_{22}}: \begin{cases} \text{IF } R_j = 0 \text{ THEN GOTO EXIT } 2 \\ R_j \leftarrow R_j - 1 \\ R_{22} \leftarrow R_{22} + 1 \\ \text{IF } R_{21} = 0 \text{ THEN GOTO EXIT } 1 \end{cases} \\ \underbrace{\text{EXIT } 2}_{\substack{\text{Copio } R_{22} \text{ che vale } R_j \text{ in} \\ R_k \text{ e ancora in } R_j}}: \begin{cases} \text{IF } R_{22} = 0 \text{ THEN GOTO EXIT } 3 \\ R_{22} \leftarrow R_{22} - 1 \\ R_j \leftarrow R_j + 1 \\ R_k \leftarrow R_k + 1 \\ \text{IF } R_{21} = 0 \text{ THEN GOTO EXIT } 2 \end{cases} \\ \text{EXIT } 3: \ R_k \leftarrow R_k \pm 1 \quad \text{alla fine in } R_k \text{ devo avere } R_j \pm 1 \end{cases}$

Caso $C \equiv$ **begin** $C_1; \ldots; C_m$ **end:** $\text{Comp}(C) = \begin{cases} \text{Comp}(C_1) \\ \vdots \\ \text{Comp}(C_m) \end{cases}$

Caso $C \equiv$ **while** $x_k \neq 0$ **do** C_1**:** $\text{Comp}(C) = \begin{cases} \text{LOOP}: \begin{cases} r\text{IF } R_k = 0 \text{ THEN GOTO EXIT} \\ \text{Comp}(C_1) \\ \text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \end{cases} \\ \text{EXIT}: \ R_{22} \leftarrow R_{22} - 1 & \text{l'istruzione serve come "riempitivo" ma non ha nessuna funzione.} \end{cases}$

Comp è una funzione che ad programma in linguaggio while associa un programma in linguaggio RAM. *Comp è una traduzione.*

Dimostrazione 2.1 | Correttezza del compilatore

Verifichiamo che **Comp** rispetta le proprietà di una *traduzione*:

1. Poiché è definita per ricorsione sulla definizione induttiva di **W-Programmi**, *Comp* è una funzione totale, essa è inoltre facilmente programmabile in PASCAL.

2. Dobbiamo ora mostrare che **Comp** mantiene la semantica, cioè $\forall P \in$ W-Programmi $\quad (\psi_P = \varphi_{\text{Comp}(P)})$.

   Cominciamo con l'osservare che se $\overline{C} = \text{Comp}(C)$, allora $\overline{C}$ contiene al più le variabili $R_0, R_1, R_2, \ldots, R_{20}, R_{21}, R_{22}$ (C sia un qualsiasi comando).

   Con una immediata induzione, si osserva che se i registri R_{21} e R_{22} sono azzerati prima della esecuzione di $\overline{C} = \text{Comp}(C)$, al termine dell'esecuzione il contenuto di R_{21} e R_{22} sarà nuovamente 0.

   Associamo ora ad ogni programma $\overline{C} = \text{Comp}(C)$ una funzione $\delta_{\overline{C}} : \mathbb{N}^{21} \longrightarrow \mathbb{N}^{21} \cup \{\perp\}$ che al vettore $(x_0, x_1, \ldots, x_{20})$ associa il vettore $(y_0, y_1, \ldots, y_{20})$ con la seguente regola: se x_k $(0 \leqslant k \leqslant 20)$ è il contenuto del registro R_k prima della esecuzione di C e i registri R_{21} e R_{22} sono azzerati, allora y_k $(0 \leqslant k \leqslant 20)$ è il contenuto di R_k dopo la esecuzione di $\overline{C}$ (nel caso in cui la computazione termini).

   Poiché w-Stati $= \mathbb{N}^{21}$, vale che:

   $$\delta_{\overline{C}} : \text{w-Stati} \longrightarrow \text{w-Stati} \cup \{\perp\}$$

   Dimostriamo ora, utilizzando lo schema di induzione prodotto dalla definizione induttiva dell'insieme dei comandi, che vale:

   $$[\![C]\!] = \delta_{\text{Comp}(C)}$$

   (a) Sia A un comando di assegnamento. La verifica che: $[\![A]\!] = \delta_{\text{Comp}(A)}$ è immediata.

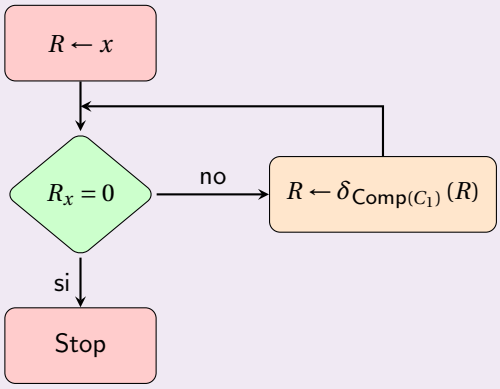
   (b) Sia $C \equiv$ begin $C_1; \ldots; C_m$ end : valgono:

   $$\begin{aligned} [\![C]\!] &\equiv [\![C_m]\!] \circ [\![C_{m-1}]\!] \circ \ldots \circ [\![C_1]\!] \\ &= \delta_{C_m} \circ \delta_{C_{m-1}} \circ \ldots \circ \delta_{C_1} \\ &= \underbrace{\delta_{\text{Comp}(C_m)} \circ \ldots \circ \delta_{\text{Comp}(C_1)}}_{\text{Ipotesi d'induzione}} \end{aligned}$$

   $$\begin{aligned} \delta_{\text{Comp}(C)} &= \delta \begin{cases} \text{Comp}(C_1) \\ \vdots \\ \text{Comp}(C_m) \end{cases} \\ &= \delta_{\text{Comp}(C_m)} \circ \ldots \circ \delta_{\text{Comp}(C_1)} \end{aligned}$$

   e notando che i registri R_{21} e R_{22} sono uguali a zero: $[\![C]\!] = \delta_{\text{Comp}(C)}$

   (c) Sia $C \equiv$ while $x_k \neq 0$ do C_1, $\delta_{\text{Comp}(C)}(\underline{x})$ è ottenuto dal flow-chart interpretato:

   Quindi $\delta_{\text{Comp}(C)}(\underline{x}) = \delta^e_{\text{Comp}(C_1)}(\underline{x})$ dove e è il primo intero per cui la k-esima componente di $\delta^e_{\text{Comp}(\underline{x})}(\underline{x}) = 0$.

   Poiché, per l'ipotesi di induzione, $\delta_{\text{Comp}(C_1)} = [\![C_1]\!]$, vale allora:

   $$\delta_{\text{Comp}(C)}(\underline{x}) = [\![C_1]\!]^e(\underline{x})$$

   dove e è il primo intero per cui la k-esima componente è 0. Poiché inoltre, per definizione della semantica del comando while, è $[\![C]\!](\underline{x}) = [\![C_1]\!]^e(\underline{x})$ concludiamo che per ogni comando C vale:

   $$[\![C]\!] = \delta_{\text{Comp}(C)}$$

   Sia P un programma while: esso è un particolare comando, quindi $[\![P]\!] = \delta_{\text{Comp}(P)}$. Allora concludiamo che:

   $$\begin{aligned} \psi_P(x) &= O-\text{esima componente di } [\![P]\!](w - \text{in } x) \\ &= O-\text{esima componente di } \delta_{\text{Comp}(P)}(\text{in } x) \\ &= \varphi_{\text{Comp}(P)}(x) \end{aligned}$$

Teorema 2.2 | Conseguenza di correttezza del compilatore

Una immediata conseguenza del teorema di **correttezza del compilatore** è che ogni funzione computabile in linguaggio while è anche computabile in linguaggio RAM (ridotto), e pertanto vale quindi che:

$$\text{Funzioni}(\text{While}) \subseteq \text{Funzioni}(\text{RAM})$$

2.4 Macroistruzioni

Teorema 2.3 | Esistenza di macroistruzioni

Esistono **macroistruzioni**, comandi che fanno riferimento a set di comandi noti, del tipo:

$$x_i = \text{Operazione}(x_j, x_s) \quad \text{oppure} \quad x_i = \text{Operazione}(x_j) \qquad (i, j, s \leqslant 10)$$

per il seguente elenco di operazioni:

Somma: $(x + y)$

Prodotto: $(x \cdot y)$

Sottrazione fra interi positivi: $(x - y)$

Divisione intera: $(x \div y)$

Modulo: $(\langle x \rangle_y)$

Coppia di Cantor: $(\langle x, y \rangle)$

Proiezione destra di Cantor: $(\text{des } x)$

Proiezione sinistra di Cantor: $(\text{sin } x)$

Proiezione sulla k-esima componente:

$$\text{Pro}(k, x) = \text{se } k \leqslant l \text{ allora } x_k$$

Incrementa la k-esima componente:

$$\text{inc}(k, x) = (x_1, \ldots, x_k + 1, \ldots, x_e)$$

Decrementa la k-esima componente:

$$\text{dec}(k, x) = (x_1, \ldots, x_k - 1, \ldots, x_e)$$

Determina la lunghezza della sequenza:

$$\text{lung}(x) = l$$

Associa al numero l il numero che rappresenta una sequenza di l zeri:

$$\text{ze}(l) = (0, 0, \ldots, 0)$$

2.5 Aritmetizzazione di programmi

Definizione 2.34 | Aritmetizzazione di programmi

L'**aritmetizzazione** di Programmi è una funzione:

$$\text{Cod} : \text{Programmi} \longrightarrow \mathbb{N}^+$$

tale che:

1. Cod è una corrispondenza biunivoca.
2. Cod può essere computata con un programma PASCAL.

Esempio 2.1 | Aritmetizzazione di Programmi

Utilizzando le funzioni $\langle x, y \rangle$ (funzione coppia di Cantor), $\langle x \rangle_3$ (resto della divisione tra x e 3) e $x \div 3$ (quoto della divisione tra x e 3):

1. Sia **Istruzioni** l'insieme delle istruzioni. Sia $\text{Ar} : \text{Istruzioni} \longrightarrow \mathbb{N}$ definito da:

   Caso istr $= R_k \leftarrow R_k + 1$**:** $\text{Ar}(\text{istr}) = 3 \cdot k$ (multipli di 3)

   Caso istr $= R_k \leftarrow R_k - 1$**:** $\text{Ar}(\text{istr}) = 3 \cdot k + 1$ (multipli di 3 più uno)

   Caso istr $=$ **IF** $R_k = 0$ **THEN GOTO** m**:** $\text{Ar}(\text{istr}) = 3 \cdot \langle k, m \rangle - 1$ (multipli di 3 meno uno)

   Ar è una corrispondenza biunivoca.

2. La funzione $\text{Cod} : \text{Programmi} \longrightarrow \mathbb{N}^+$ è realizzata come segue: se $\text{Cod}(P) = \text{istr}_1; \ldots; \text{istr}_s$ allora:

   $$\text{Cod}(P) = \langle \text{Ar}(\text{istr}_1), \langle \ldots \langle \text{Ar}(\text{istr}_s), 0 \rangle \ldots \rangle \rangle$$

   Cod è allora una aritmetizzazione.

Definizione 2.35 | Numero di Gödel

Se $j = \text{Cod}(P)$, diciamo che j è il numero di Gödel di P, o che P è lo j-esimo programma nella numerazione di Gödel.

Problema 2.1 |

Determinare il primo j tale per cui:

$$\forall x : \varphi_j(x) = \perp$$

Soluzione 2.1 | Soluzione del problema 2.5

Bisogna identificare il programma corrispondente al j più piccolo che sia indefinito: una possibile soluzione è il programma seguente:

$$0 : \text{IF } R_0 = 0 \text{ THEN GOTO } 0$$

Il registro R_0 (scelto perché è il registro con l'indice più piccolo) viene inizializzato a 0, per cui la condizione $R_0 = 0$ è sempre vera, per cui il GOTO procede a tornare all'istruzione 0-esima.

Questo programma viene codificato a $j = 3 \cdot \langle 0, 0 \rangle - 1 = 3 \cdot 1 - 1 = 2$: per essere sicuri che questo sia il j più piccolo strettamente positivo che porta a un $\perp$, controlliamo se per $j = 1$ la soluzione è definita:

$$0 : R_0 \leftarrow R_0 + 1$$

Il programma così definito non da esito $\perp$, per cui possiamo confermare che $j = 2$ è il più piccolo valore corrispondente a un programma con esito non definito.

Problema 2.2 |

Determinare infiniti j_k tale per cui:

$$\forall x : \varphi_{j_k}(x) = 0$$

Soluzione 2.2 | Soluzione del problema 2.5

Utilizzando il parametro k per generare infiniti programmi, basta non modificare il registro R_0 e non generare deadlock per poter avere esito 0, quindi $\forall k \in \mathbb{N}^+$ possiamo realizzare:

$$0 : R_k \leftarrow R_k + 1$$

Quindi qualsiasi j multiplo di 3 corrisponde a un programma che ha come valore, per ogni ingresso, 0.

Problema 2.3 |

La funzione $f(x) = \varphi_x(1)$ è "intuitivamente" calcolabile?

Soluzione 2.3 | Soluzione del problema 2.5

Per mostrare che la funzione non è calcolabile è sufficiente identificare un x tale per cui il programma corrispondente, con input 1, non è calcolabile: abbiamo già identificato un tale programma nella soluzione del problema 2.5: pertanto la funzione non è calcolabile.

2.6 Programma interprete e funzione universale

Definizione 2.36 | Interprete

Costruiamo *int* utilizziamo macroistruzioni del linguaggio while: per l'esistenza di Comp : W-Programmi $\longrightarrow$ Programmi concluderemo che *int* è esprimibile in linguaggio RAM.

Il nostro obbiettivo è di simulare con un programma while l'esecuzione di una RAM con programma P_j, di numero di Gödel j, sul dato x. Poiché il programma P_j non contiene esplicitamente la variabile R_k, se $k \geqslant 1$, tali variabili permarranno azzerate nell'esecuzione del programma. Potremmo pertanto definire, relativamente all'esecuzione di P_j, lo stato attraverso il contenuto dei registri L ed R_k $(0 \leqslant k \leqslant j+2)$.

In realtà basterebbero i registri sino a R_5 ma ci allarghiamo un poco.

Le variabili del programma *int* contengono:

x_0 **contiene** $(x_0, x_1, \ldots, x_{j+1}, x_{j+2})$ cioè lo stato di memoria.

x_1 **contiene** l, dove l è interpretato come contenuto corrente del registro L.

x_2 **contiene** il dato x

x_3 **contiene** il codice j del programma P_j

x_4 **contiene** il codice dell'istruzione corrente da eseguire sulla macchina da simulare.

Mette il dato x e il codice di programma j rispettivamente nei registri x_2 ed x_3	$\begin{cases} x_2 = \sin(x_1) \\ x_3 = \mathrm{des}(x_1) \end{cases}$	Proiettori
Il contenuto di x_0 è posto essere $(0, x, 0, 0, \ldots, 0)$	$\begin{cases} x_0 = \underbrace{Z_e(x_3)}_{\text{Determina il numero di zero che vanno dopo } x\text{: sono 20}} \\ x_0 = \langle 0, \langle x_3, x_0 \rangle \rangle \end{cases}$	Questa coppia di istruzioni produce la sequenza $(0, x, 0, \ldots, 0)$
Simula l'inizializzazione di L	$\{x_1 = 1$	Ora è a tutti gli effetti una macchina RAM

Finché non si arriva alla condizione di terminazione $(L = 0 \iff x_1 = 0)$ si desume dal contenuto di x_1 l'istruzione corrente da simulare, se esiste (cioè se $x_1 \leqslant \mathrm{lung}(j)$). In tal caso se ne mette il codice in x_4, si decodifica e si simula la istruzione, dei due effetti di cambiamento di stato di memoria e di L.

Il codice del procedimento è illustrato nella figura 2.2.

Una volta raggiunta la condizione di terminazione si lascia in x_0 il contenuto finale di R_0 nella macchina simulata:

$$x_0 = \sin(x_0)$$

Poiché il programma utilizza esplicitamente 9 variabili e le macroistruzioni possono essere sostituite con istruzioni utilizzando al più 5 ulteriori variabili, concludiamo che esiste un programma int_1 scritto in linguaggio while che utilizza 14 variabili e tale che:

$$\psi_{int_1}(\langle x, y \rangle) = \varphi_y(x)$$

Detto int_2 il programma RAM tale che $int_2 = \mathrm{Comp}(int_1)$, vale che:

1. int_2 contiene al più 16 variabile R_k
2. $\varphi_{int_2}(\langle x, y \rangle) = \varphi_{\mathrm{Comp}(int_1)}(\langle x, y \rangle) = \varphi_y(x)$

while $x_1 \neq 0$ **do**

if $x_1 > \mathbf{lung}(j)$ then $x_1 = 0$

else

$x_4 = \mathrm{Pro}(x_1, x_3)$

if $\langle x_4 \rangle_3 = 0$ **then** $\begin{cases} x_5 = x_4 \div 3 \\ x_0 = \mathrm{inc}(x_5, x_0) \\ x_1 = x_1 + 1 \end{cases}$

if $\langle x_4 \rangle_3 = 1$ **then** $\begin{cases} x_5 = x_4 \div 3 \\ x_0 = \mathrm{dec}(x_5, x_0) \\ x_1 = x_1 + 1 \end{cases}$

if $\langle x_4 \rangle_3 = 2$ **then** $\begin{cases} x_5 = x_4 \div 3 + 1 \\ x_6 = \mathrm{des}(x_5) \\ x_7 = \sin(x_5) \\ x_8 = \mathrm{Pro}(x_7, x_6) \\ \text{if } x_8 = 0 \text{ then } x_1 = x_0 \text{ else } x_1 = x_1 + 1 \end{cases}$

Figura 2.2: Processo di simulazione

Teorema 2.4 | Funzione universale

Facendo riferimento all'interprete int_2 definito precedentemente, dato il sistema di programmazione RAM $\{\varphi_j\}$, esiste un indice $u = \text{Cod}(int_2)$ tale che:

$$\forall x \forall y \quad \varphi_u(\langle x, y \rangle) = \varphi_y(x)$$

La funzione parziale φ_u è detta **funzione universale** per $\{\varphi_j\}$.

2.7 Eliminazione del GOTO

Teorema 2.5 | Da programma RAM a While

Per ogni programma RAM P esiste un programma while W contenente al più 14 variabili x_k tale che:

$$\forall x \quad \varphi_P(x) = \varphi_W(x)$$

Lo stesso risultato può essere espresso come:

$$\text{Funzioni(RAM)} \subseteq \text{Funzioni(while)}$$

Dimostrazione 2.2 | Da programma RAM a While

Sia $j = \text{Cod}(P)$ il numero di Gödel di P. Consideriamo il seguente programma while con macroistruzioni:

$$\text{begin } x_0 = j; x_1 = \langle x_1, x_0 \rangle; x_0 = 0; x_0 = \varphi_{int_1}(x_1) \text{ end}$$

dove:

1. La macroistruzione $x_0 = j$ sta per: begin $\underbrace{x_0 = x_0 + 1; x_0 = x_0 + 1; \ldots; x_0 = x_0 + 1}_{j \text{ volte}}$ end
2. La macroistruzione $x_0 = \varphi_{int_1}(x_1)$ è realizzata dal programma int_1.

Siccome int_1 usa 14 variabili (compresa x_0) e su x_0 non ci sono effetti collaterali, il precedente programma abbrevia un programma while W che utilizza 14 variabili. Inoltre:

$$\psi_W(x) = \psi_{int_1}(\langle x, j \rangle) = \varphi_j(x) = \varphi_P(x)$$

3

Le funzioni ricorsive parziali

Definizione 3.1 | Composizione

Se $h(x_1,\dots,x_m)$ è una funzione a m argomenti e $g_1(\underline{x}),\dots,g_m(\underline{x})$ sono funzioni a n argomenti, la composizione di h e $g_1,\dots,g_m$ è la funzione f in n argomenti definita:

$$f(\underline{x}) = h(g_1(\underline{x}), g_2(\underline{x}),\dots,g_m(\underline{x}))$$

Definizione 3.2 | Ricorsione primitiva

Se $h(y,z,\underline{x})$ è una funzione a $n+2$ argomenti e $g(\underline{x})$ una funzione a n argomenti, la ricorsione primitiva applicata a h e g determina una funzione $\mathrm{RP}(h \mid g) = f(y,\underline{x})$ in $n+1$ argomenti definita da:

$$\mathrm{RP}(y,\underline{x}) = \begin{cases} g(\underline{x}) & \text{se } y = 0 \\ h(\mathrm{RP}(y-1,\underline{x}), y-1, \underline{x}) & \text{se } y > 0 \end{cases}$$

Definizione 3.3 | Minimalizzazione

L'operazione di minimalizzazione applicato su una funzione ad $n+1$ argomenti $\underline{x}, y$ restituisce se esiste il più piccolo valore dell'$n+1$-esimo elemento che annulla il valore della funzione.

$$\min(f)(\underline{x}) = \begin{cases} y & \text{se } y = \operatorname{argmin}_y f(\underline{x}, y) = 0 \\ \perp & \text{altrimenti} \end{cases}$$

Essa è una funzione parziale.

Definizione 3.4 | Funzioni base

Le funzioni base sono:

Annullamento: $O^n(x_1, x_2,\dots,x_n) = 0$

Successore: $S(x) = x+1$

Proiezione: $\mathrm{Pro}_k^n(x_1,\dots,x_k,\dots,x_n) = x_k$

Definizione 3.5 | Funzioni ricorsive parziali

La classe delle **funzioni ricorsive parziali** P_f è la più piccola classe contenente le funzioni base e chiusa rispetto a *composizione*, *ricorsione primitiva* e *minimalizzazione*.

Indicheremo con P_f^n la classe delle funzioni ricorsive parziali in n argomenti.

Osservazione 3.1 | Come dimostrare che una proprietà vale per tutte le funzioni ricorsive parziali

Data una proprietà P, per dimostrare che essa vale per tutte le *funzioni ricorsive parziali* P_f, è sufficiente seguire il seguente schema di induzione:

1. P vale per tutte le **funzioni base**.
2. (a) Se P vale per $h, g_1,\dots,g_m$ allora vale anche per la **composizione**.
   (b) Se P vale per h e g allora vale anche per la funzione definita per **ricorsione primitiva**.
   (c) Se P vale per g, allora vale anche per la funzione definita per **minimalizzazione**.

Definizione 3.6 | Funzioni ricorsive totali

Chiamiamo T_f la classe di funzioni totali $f \in P_f$, chiamandola classe delle **funzioni ricorsive totali**.

Indicheremo con T_f^n la classe delle funzioni ricorsive totali in n argomenti.

3.1 Da funzioni ricorsive parziali a programmi while

Vogliamo determinare la relazione tra l'insieme delle funzioni realizzabili in linguaggio while e delle funzioni appartenenti alla classe delle P_f. Per poter realizzare funzioni while a n argomenti procediamo ad introdurre dei vettori.

Definizione 3.7 | Vettore di dimensione n a componenti intere

Una possibile rappresentazione dei vettori è:

$$[x_1] = x_1$$

$$\underbrace{[x_1, x_2] = \langle x_1, x_2 \rangle - 1}$$

dove $[,]$ realizza un isomorfismo $[,]:\mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}$

$$\underbrace{[x_1, x_2, x_3] = [x_1, [x_2, x_3]]}$$

dove $[,,]$ realizza un isomorfismo $[,,]:\mathbb{N}\times\mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}$

$\dots$

Definiamo le funzioni computabili da programmi while a n argomenti come il sistema $\{\psi_P^n\}$, dove $\psi_P^n = \psi_P([x_1, \dots, x_n])$. In sostanza, l'indice n in alto funziona come una dichiarazione di tipo delle variabile in ingresso.

Teorema 3.1 | Funzioni ricorsive parziali e programmi while

Ogni funzione $f \in P_f$ è computabile da programmi while.

Dimostrazione 3.1 | Funzioni ricorsive parziali e programmi while

Per quanto riguarda le funzioni base, esse sono computabili da programmi while con una immediata verifica.

Procediamo quindi verificare i rimanenti tre punti:

Composizione: Sia $h(x_1, \dots, x_m) = \psi_W^m$ per un programma while W, e siano $g_k(\underline{x}) = \psi_{B_k}^n$ per programmi while B_k, con $1 \leqslant k \leqslant m$. Il seguente programma calcola allora la composizione $h(g_1, \dots, g_m)$:

begin

$x_0 = \psi_{B_m}(x_1)$;

$x_0 = [x_0, \psi_{B_{m-1}}(x_1)]$;

$\vdots$

$x_0 = [x_0, \psi_{B_m}(x_1)]$;

$x_0 = \psi_W(x_0)$;

end

Ricorsione primitiva: Sia $h(y, z, \underline{x}) = \psi_W^{n+2}$ per un programma while W e sia $g(\underline{x}) = \psi_B^n$ per un programma while B. Allora la funzione ottenuta per ricorsione primitiva da h e g è calcolata dal programma:

begin

$x_4 = \mathrm{Pro}_1(x_1)$;

$x_1 = \mathrm{des}(x_1)$;

$x_0 = \psi_B(x_1)$;

while $x_3 \neq x_4$ do:

$x_0 = \psi_W([x_0, x_3, x_1])$

$x_3 = x_3 + 1$

end

Si noti che il programma è iterativo.

Minimalizzazione: Supponiamo che esista un programma W che calcola g, cioè $g = \psi_W^{n+1}$. Allora il seguente programma while calcola la funzione $f(\underline{x})$ ottenuta applicando a g l'operatore di minimalizzazione:

begin

$x_2 = \psi_W([0, x_1])$;

while $x_3 \neq 0$ do:

$x_0 = x_0 + 1$

$x_2 = \psi_W([x_0, x_1])$

end

3.2 Da programmi while a funzioni ricorsive parziali

Poniamoci ora il problema opposto: sono le funzioni computate da programmi while ricorsive parziali?

Per semplificare la notazione, ricordiamo che ad ogni comando C è associata una funzione parziale $[\![C]\!]:\mathbb{N}^{21}\longrightarrow\mathbb{N}^{21}$. D'altro lato, usando le funzioni coppia $[,]$, abbiamo mostrato che c'è un isomorfismo $\mathbb{N}^{21}\simeq\mathbb{N}$. Potremo allora interpretare la semantica $[\![C]\!]$ come una funzione parziale $f_C:\mathbb{N}\longrightarrow\mathbb{N}$, definita come segue:

1. All'ingresso x associamo il vettore a 21 componenti rappresentato da x.
2. Trasformiamo tale vettore in $[y_1,\ldots,y_{21}]$ con la funzione $[\![C]\!]$.
3. Otteniamo l'uscita $y=f_C(x)$ interpretando tale vettore come numero.

Teorema 3.2 | Da programmi while a funzioni ricorsive parziali

Se C è un comando del linguaggio while, allora $f_C\in P_f$.

Dimostrazione 3.2 | Da programmi while a funzioni ricorsive parziali

Se P è un programma, $\psi_P(x)=\text{Pro}\big(1,f_P([0,x,0,\ldots,0])\big)$. Se dimostriamo che f_P è ricorsiva parziale, allora anche ψ_P lo è, in quanto ottenuta per composizione di funzioni ricorsive parziali.

1. Se A è un assegnamento, allora $f_A\in P_f$ con verifica immediata.
2. (a) Sia $C\equiv$ begin $C_1,\ldots,C_m$ end un comando e $f_{C_k}\in P_f$, con $1\leqslant k\leqslant m$. Allora $f_C=f_{C_m}\circ\ldots\circ f_{C_1}\in P_f$, in quanto ottenuto per composizione di funzioni di P_f.

   (b) Sia $C\equiv$ while $x_k\neq 0$ do C_1 e sia $f_{C_1}\in P_f$. Ricordando la semantica:

$$f_C(x)=f_{C_1}^{\mu e}(x)^{\text{Pro}\left(k,f_{C_1}^e(x)\right)=0}$$

   Consideriamo la funzione $S\big(y,x\big)=f_{C_1}^y(x)$. Dall'identità:

$$f_{C_1}^y=f_{C_1}^1\circ f_{C_1}^{y-1}\ \text{ con }\ y>0$$

   segue che:

$$S\big(y,x\big)=\begin{cases}x & \text{se } y=0\\ f_{C_1}^1\big(S\big(y-1,x\big)\big) & \text{altrimenti}\end{cases}$$

   Sappiamo che $S\big(y,x\big)\in P_f^2$ in quanto definita per ricorsione primitiva di funzioni in P_f. Allora $\text{Pro}\big(k,S\big(y,x\big)\big)\in P_f$, perché composizione di funzioni di P_f, e $e(x)=\mu y\big(\text{Pro}\big(k,S\big(y,x\big)\big)\big)\in P_f$, in quanto definita per minimalizzazione di una funzione in P_f.

   $S(e(x),x)\in P_f$ perché definita come composizione di funzione in P_f, e quindi $f_C(x)\in P_f$, poiché $f_C(x)=S(e(x),x)$.

Teorema 3.3 | Equivalenza tra insiemi di funzioni

$$\text{Funzioni (RAM)}=\text{Funzioni}\big(\text{while}\big)=\text{Funzioni (PASCAL)}=P_f$$

Tutti i formalismi che abbiamo presentato permettono il calcolo della stessa classe di funzioni: le **funzioni ricorsive parziali**.

Teorema 3.4 | Tesi di Church

La classe delle funzioni **intuitivamente calcolabili** (calcolabili in modo automatico) è la classe delle **funzioni ricorsive parziali.**

4

Esistenza di problemi non decidibili

Definizione 4.1 | Problema

Descriviamo un problema come una coppia:

$$\langle \langle \text{istanza} \rangle, \langle \text{questione} \rangle \rangle$$

dove $\langle \text{istanza} \rangle$ è una variabile su un dato dominio D, $\langle \text{questione} \rangle$ un predicato su D.

Definizione 4.2 | Problema decidibile

Diremo che un problema M è **decidibile** se esiste una funzione ricorsiva totale $\Phi : D \longrightarrow \{0,1\}$ per cui:

$$\varphi(x) = \begin{cases} 1 & \text{se la condizione } p \text{ è verificata su } x \\ 0 & \text{altrimenti} \end{cases}$$

Richiediamo cioè che la funzione caratteristica dell'estensione di p sia ricorsiva totale.

Definizione 4.3 | Problema dell'arresto

Poniamoci il problema di decidere l'arresto dell'esecuzione del programma P sul dato x:

Problema: Arresto per P, (AP)

Istanza: Un intero x

Questione: È $\varphi_P(x) \neq \perp$?

È la sequenza di esecuzione del programma P sul dato x finita?

Teorema 4.1 | Problema dell'arresto

(AP) non è decidibile.

Dimostrazione 4.1 | Problema dell'arresto

Supponiamo per assurdo che (AP) sia decidibile.

Esiste allora una funzione ricorsiva totale Φ_{AP} tale che:

$$\Phi_{AP}(x) = \begin{cases} 0 & \text{se } \varphi_P(x) = \perp \\ 1 & \text{altrimenti} \end{cases}$$

Allora anche la seguente funzione è ricorsiva totale:

$$\psi(x) = \begin{cases} 0 & \text{se } \Phi_{AP}(x) = 0 \\ 1 + \varphi_P(x) & \text{altrimenti} \end{cases}$$

Esisterebbe quindi un programma RAM, con un numero di Gödel e, che computa ψ, cioè: $\forall x \quad \varphi_e(x) = \psi(x)$.

Vale allora la seguente catena:

$$\begin{aligned} \psi(e) &= \underbrace{\varphi_e(e) = \varphi_P(e)}_{e = P \text{ è un possibile assegnamento}} \\ &= \begin{cases} 0 & \text{se } \Phi_{AP}(e) = 0 \\ 1 + \varphi_P(e) \end{cases} \end{aligned}$$

$\Phi_{AP}(e)$ non può essere 1, altrimenti sarebbe $\varphi_P(e) = 1 + \varphi_P(e)$.

Poiché Φ_{AP} è totale e prende valori in $\{0,1\}$, deve essere $\Phi_{AP}(e) = 0$.

Se $\Phi_{AP}(e) = 0$, dalla catena precedente segue che $\varphi_P(e) = 0$, mentre dalla definizione segue che $\varphi_P(e) = \perp$, da cui l'assurdo.

4.1 Funzione del linguaggio di programmazione per il linguaggio RAM

Definizione 4.4 | Funzione del linguaggio di programmazione

La legge che ad ogni P e ad ogni $\overline{y}$ associa $P_{\overline{y}}$ viene chiamata S_1^1 **funzione del linguaggio di programmazione** ed è una funzione ricorsiva totale.

Teorema 4.2 | S_1^1 Funzione del linguaggio RAM

Dato il sistema di programmazione RAM $\{\varphi_i\}$, esiste una funzione ricorsiva totale $S_1^1(i,y)$ tale che:

$$\forall i \forall y \forall x \quad \varphi_{S_1^1(i,y)}(x) = \varphi_i\left(\langle x,y\rangle\right)$$

Significa che dato un programma P che lavora su due variabili, è sempre possibile generare automaticamente un programma P' che lavora su una variabile fissando l'altra.

Dimostrazione 4.2 | S_1^1 Funzione del linguaggio RAM

Sia P_i il programma di numero di Gödel i, che computa la funzione φ_i. Fissato un $y \in \mathbb{N}$, sia $\overline{P}_{i,y}$ il segmento programma RAM, scritto usando macroistruzioni:

$$\overline{P}_{i,y} = \underbrace{R_0 \leftarrow R_0+1; \ldots; R_0 \leftarrow R_0+1}_{y \text{ volte}}; R_1 \leftarrow \langle R_1, R_0\rangle; R_0 \leftarrow 0; P_i$$

Si osserva immediatamente che la funzione computata da $\overline{P}_{i,y}$ su ingresso x è $\varphi_i\left(\langle x,y\rangle\right)$. Basta allora porre:

$$S_1^1(i,y) = \mathrm{Cod}\left(\overline{P}_{i,y}\right)$$

Non è quindi difficile fornire un algoritmo che, dato i e y, dà in uscita il valore $S_1^1(i,y)$.

S_1^1 è allora una funzione ricorsiva totale.

Teorema 4.3 | S_n^m Funzione del linguaggio RAM

Dato il sistema di programmazione RAM $\{\varphi_i\}$ esiste una funzione ricorsiva totale a $n+1$ variabili $S_n^m(i,y)$ tale che:

$$\forall i \forall \underline{\boldsymbol{y}} \forall \underline{\boldsymbol{x}} \quad \varphi_{S_n^m(i,y)}\left(\underline{\boldsymbol{x}}\right) = \varphi_i\left(\underline{\boldsymbol{x}}, \underline{\boldsymbol{y}}\right)$$

Questo vale anche per altri linguaggi di programmazione.

5

Sistema di programmazione accettabile

Osservazione 5.1 | Come costruire un sistema di programmazione accettabile

Identificheremo con $\mathbb{N}$ il linguaggio di programmazione. I requisiti che richiederemo sono:

1. Un sistema di programmazione deve contenere programmi per tutte le funzioni ricorsive parziali di un argomento $\left(P_f^1\right)$.
2. Deve esistere un programma universale u che, ricevuta in ingresso la coppia $\langle x, y\rangle$, simula il programma y sul dato x.
3. Deve essere possibile scambiare argomenti e parametri in modo automatico, per il teorema S_n^m.

Definizione 5.1 | Sistema di programmazione accettabile

Un sistema di programmazione è detto **accettabile** se rispetta le seguenti 3 proprietà:

1. È concorde con la tesi di Church: permette di calcolare esattamente P_f, la classe delle funzioni ricorsive parziali.
2. Ammette interprete universale.
3. Ammette la funzione del linguaggio di programmazione descritta nel teorema S_1^1.

Definizione 5.2 | Traduzione

Dati due sistemi di programmazione $\{\varphi_i\}$ e $\{\psi_i\}$, una **traduzione** t da $\{\varphi_i\}$ a $\{\psi_i\}$ è una funzione $t : \mathbb{N} \longrightarrow \mathbb{N}$ tale che:

1. t è ricorsiva totale
2. $\forall i \in \mathbb{N} : \psi_{t(i)} = \varphi_i$

Definizione 5.3 | Quines

Si tratta di una classe di programmi che stampano sé stessi. Un sistema di programmazione accettabile ammette sempre un quine.

Teorema 5.1 | Teorema di ricorsione

Sia $t : \mathbb{N} \longrightarrow \mathbb{N}$ una funzione ricorsiva totale, e sia $\{\varphi_j\}$ un sistema di programmazione. Esiste allora un "programma" $n \in \mathbb{N}$ tale che: $\varphi_n = \varphi_{t(n)}$.

In altre parole, fissato un sistema di programmazione accettabile, per qualsiasi funzione ricorsiva totale t esiste un programma la cui semantica non cambia sotto t.

Dimostrazione 5.1 | Teorema di ricorsione

Per dimostrare il teorema di ricorsione, si prenda la particolare funzione $\varphi_{\varphi_i(i)}(x)$ e si applichi ad essa per due volte la definizione di interprete universale:

$$\begin{aligned}\varphi_{\varphi_i(i)}(x) &= \varphi_{\varphi_u(i,i)}(x)\\ &= \varphi_u\left(x, \varphi_u(i,i)\right)\\ &= x, i\end{aligned}$$

La funzione ottenuta è una composizione di φ_u su sè stessa ed è quindi una **funzione ricorsiva parziale**. Siccome $\{\varphi_i\}$ è un sistema di programmazione accettabile, esisterà un particolare programma $e \in \mathbb{N}$ tale che $\varphi_e(x,i) = f(x,i)$ e allora per il teorema S_1^1 possiamo scrivere:

$$\varphi_e(x,i) = \varphi_{S_1^1(e,i)}(x) = \varphi_{\varphi_i(i)}(x)$$

Per costruzione, si ha quindi che $t\left(S_1^1(e,i)\right) \in T_f$ poiché essa risulta ottenuta come composizione di funzioni ricorsive totali, e pertanto sarà calcolata in particolare da un programma m, cioè:

$$t\left(S_1^1(e,i)\right) = \varphi_m(i)$$

Allora si dimostra che $n = S_1^1(e,m)$ mantiene la semantica sotto t ricavando:

$$\begin{aligned}\varphi_n(x) &= \varphi_{S_1^1(e,m)}(x)\\ &= \varphi_e(x,m)\\ &= \varphi_{\varphi_m(m)}(x)\end{aligned}$$

Ed inoltre:

$$\begin{aligned}\varphi_{t(n)}(x) &= \varphi_{t\left(S_1^1(e,m)\right)}(x)\\ &= \varphi_{\varphi_m(m)}(x)\end{aligned}$$

da cui la tesi.

6

Insiemi ricorsivi e ricorsivamente numerabili

Definizione 6.1 | Funzione caratteristica

La **funzione caratteristica** $X_A : \mathbb{N} \longrightarrow \{0,1\}$ di un insieme $A \subseteq \mathbb{N}$ è la funzione:

$$X_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{altrimenti} \end{cases}$$

Definizione 6.2 | Insieme ricorsivo

Intuitivamente un sottoinsieme $A \subseteq \mathbb{N}$ è ricorsivo se e solo se esiste un procedimento finito che permette di decidere, per ogni $x \in \mathbb{N}$, se $x \in A$ oppure $x \notin A$.

Un insieme A si dice **ricorsivo** se e solo se la sua funzione caratteristica X_A è **ricorsiva totale**.

Definizione 6.3 | Relazione ricorsiva

Data una relazione $R \subseteq \mathbb{N} \times \mathbb{N}$, una relazione R sarà detta ricorsiva se la sua funzione caratteristica $X_R(x,y)$ è ricorsiva totale.

Osservazione 6.1 | Insieme ricorsivo e problema decidibile

La nozione di insieme ricorsivo coincide con quella di problema decidibile.

Definizione 6.4 | Insieme ricorsivamente numerabile

Intuitivamente, diremo che un insieme A è ricorsivamente numerabile se esiste un procedimento finito che permette di elencare gli elementi di A "uno dietro l'altro".

Più formalmente, un insieme $A \subseteq \mathbb{N}$ è detto **ricorsivamente numerabile** se o $A = \emptyset$ o A è l'immagine di una funzione ricorsiva totale $f : \mathbb{N} \longrightarrow \mathbb{N}$.

Definizione 6.5 | Insieme parzialmente decidibile

Un insieme ricorsivamente numerabile è detto anche **parzialmente decidibile**, perché, per decidere se un elemento $y \in A$, basta enumerare fino fino al valore x per cui $y = f(x)$. Naturalmente, tale metodo è parziale perché in generale non è possibile decidere se $y \notin A$.

Teorema 6.1 | Teorema: Insieme ricorsivamente numerabile

Sia $A \subseteq \mathbb{N}$. Allora le seguenti tre affermazioni sono equivalenti:

1. A è ricorsivamente numerabile.
2. A è il dominio di quale funzione ricorsiva parziale $f \in P_f$
3. Esiste una relazione ricorsiva $R \subseteq \mathbb{N} \times \mathbb{N}$ per cui:

$$A = \{x \mid \exists y : (x,y) \in R\}$$

Dimostrazione 6.1 | Teorema: Insieme ricorsivamente numerabile

Per provare la correttezza del teorema, si procederà mostrando l'esistenza di una relazione di implicazione circolare tra le tre affermazioni.

Intanto è banale osservare che $(1) \Rightarrow (2)$**:** è sempre possibile costruire un programma (che quindi appartiene al set delle funzioni ricorsive parziali) se l'insieme A è ricorsivamente numerabile.

Per mostrare che $(2) \Rightarrow (3)$, si prenda $A = \text{Dom}_f$ per qualche $f \in P_f$. Allora, per la **tesi di Church**, esisterà in particolare un programma P che calcola f: $\varphi_P(x) = x$.

Consideriamo quindi la relazione binaria, che sappiamo essere ricorsiva:

$$R_P = \{(x, y) \mid P(x) \neq \perp \text{ in } y \text{ passi}\}$$

Allora A sarà l'insieme degli elementi per i quali $P(x) \neq \perp$ termina in un numero finito di passi, cioè:

$$A = \{x \mid \exists y : (x, y) \in R_P\}$$

Infatti, preso $x \in \text{Dom}_f$, si ha che $P(x) \neq \perp$ e quindi esisterà un valore $\overline{y}$ per cui $(x, \overline{y}) \in R_P$ e di conseguenza $x \in A$ e viceversa: quindi $A = \text{Dom}_f$.

Infine mostriamo che $(3) \Rightarrow (1)$, cioè che preso $A = \{x \mid \exists y : (x, y) \in R\}$, per qualche relazione binaria R ricorsiva, allora $A = \text{Im}_f$, per qualche funzione $f \in T_f$.

Se $A = \emptyset$ allora sarebbe ricorsivamente numerabile per definizione. Viceversa, fissiamo un $a \in A$ e definiamo:

$$f(n) = \begin{cases} \sin(n) & \text{se } (\sin(n), \text{des}(n)) \in R \\ a & \text{altrimenti} \end{cases}$$

Si noti che $f \in T_f$ poiché R è ricorsiva per ipotesi e pertanto è sempre possibile decidere l'appartenenza di $(\sin(n), \text{des}(n))$. Inoltre l'insieme $A = \text{Im}_f$. Infatti $\forall x \in \text{Im}_f \cup \{a\} \Rightarrow x \in A$ per definizione.

Viceversa, se $x \in A$ allora $\exists y : (x, y) \in R$ e conseguentemente per $n = \langle x, y \rangle$ si ha che $f(n) = x$ e quindi $x \in \text{Im}_f$, che conclude la dimostrazione.

Teorema 6.2 | Insieme ricorsivo e ricorsivamente numerabile

Sia $A \subseteq \mathbb{N}$ un insieme ricorsivo. Allora A è ricorsivamente numerabile.

Teorema 6.3 | Insieme ricorsivo e suo complemento

Sia $B \subseteq \mathbb{N}$ un insieme ricorsivamente numerabile e sia B^C il suo complemento. Se anche B^C è ricorsivamente numerabile, allora B è ricorsivo.

7

Il teorema di Rice

Definizione 7.1 | Insieme che rispetta le funzioni

Dato un sistema di programmazione accettabile (φ_i), un insieme $I \subseteq \mathbb{N}$ è detto **rispettare le funzioni** se e solo se:

$$\forall i \in I \forall j : \varphi_j = \varphi_i \Leftrightarrow j \in I$$

Detto in altre parole, I rispetta le funzioni se, contenendo un programma i, allora contiene tutti i programmi che calcolano la funzione φ_i.

Teorema 7.1 | Teorema di Rice

Sia $I \subseteq \mathbb{N}$ un insieme tale che **rispetti le funzioni**. Allora I è **ricorsivo** se e solo se $I = \emptyset$ o $I = \mathbb{N}$.

Detto in altri termini, il teorema stabilisce che un insieme che rispetta le funzioni è ricorsivo solamente in casi banali: se I è ricorsivo, o tutto $\mathbb{N}$ è dentro all'insieme I o niente è dentro e $I = \emptyset$.

Dimostrazione 7.1 | Teorema di Rice

Per dimostrare il teorema si procede per assurdo: sia I un insieme ricorsivo e si supponga che $I \neq \emptyset \neq \mathbb{N}$. Allora esisteranno due elementi a e b tali che $a \in I$ e $b \notin I$. Dunque è possibile definire una traduzione:

$$t(n) = \begin{cases} a & \text{se } n \notin I \\ b & \text{se } n \in I \end{cases}$$

Se I è ricorsivo, allora l'appartenenza ad I può essere nettamente decisa e quindi la funzione t è ricorsiva totale. Sotto questa ipotesi trova quindi applicazione il **teorema di ricorsione**, in base al quale esiste certamente un $\overline{n}$ per il quale $\varphi_{\overline{n}} = \varphi_{t(\overline{n})}$.

Dunque, se $\overline{n} \in I$ si ha che $\varphi_{t(\overline{n})} = \varphi_b$ ma $b \notin I$. Viceversa per $\overline{n} \notin I$ otteniamo $\varphi_{t(\overline{n})} = \varphi_a$ con $a \in I$.

Quindi I non può essere ricorsivo.

Osservazione 7.1 | Conseguenza del teorema di Rice

Una delle conseguenze del teorema di Rice è che non risulta possibile dimostrare in modo automatico la correttezza dei programmi: cioè, dato un programma i, non esiste un programma T che su input i fornisce in output il valore 1 se $\varphi_i = f$ e 0 se $\varphi_i \neq f$, dove f è una specifica funzione in base alla quale si vuole testare la correttezza del programma.

Infatti, decidere la correttezza di i equivarrebbe a decidere l'appartenenza all'insieme:

$$A = \{i \mid \varphi_i = f\}$$

Tuttavia il teorema di Rice impone che tale insieme non sia ricorsivo: infatti A rispetta le funzioni per costruzione ed inoltre A non è l'insieme vuoto (poiché almeno $f = \varphi_i \in A$) ed in più esiste sempre almeno un programma j per cui $\varphi_j \neq f$. Pertanto A non è decidibile, il che equivale a dire che T non esiste.

Problema 7.1 | Esercizio riassuntivo

Dato un sistema di programmazione accettabile $\{\varphi_i\}$:

1. Dimostrare l'esistenza di un programma n che verifichi la seguente equazione:

$$\varphi_n(x) = \varphi_x\left(\langle n, \varphi_x(1)\rangle\right)$$

2. Successivamente, chiamato $\overline{n}$ una soluzione della equazione, si definisce l'insieme:

$$A = \{x \mid \varphi_{\overline{n}}(x) \neq \perp\}$$

Si dica a quale classe appartengono A e A^C

Soluzione 7.1 | Esercizio riassuntivo

Partiamo dal primo punto ed iniziamo con l'osservare la parte destra dell'equazione, applicando l'interprete universale:

$$\begin{aligned}\varphi_x\left(\langle n, \varphi_x(1)\rangle\right) &= \underbrace{\varphi_x\left(\langle n, \varphi_u(x,1)\rangle\right)}_{\text{Sostituisco } \varphi_x(i) \text{ con la funzione universale}} \\ &= \underbrace{\varphi_u\left(x, \langle n, \varphi_u(x,1)\rangle\right)}_{\text{Sostituisco } \varphi_x(\ldots) \text{ con la funzione universale}} \\ &= \underbrace{f(n,x)}_{\text{Diventa quindi una funzione in } n \text{ ed } x}\end{aligned}$$

dove l'ultimo passaggio indica che il risultato ottenuto è una funzione solamente di n ed x. Inoltre, $f(n,x) \in P_f$ poiché è composizione di funzioni ricorsive parziali (φ_u) e totali (la funzione coppia di Cantor). Quindi esisterà in particolare un programma α tale che $\varphi_\alpha(n,x) = f(n,x)$. Allora si ha:

$$\varphi_\alpha(n,x) = \varphi_{S_1^1(\alpha,n)}(x)$$

Con $S_1^1(\alpha,n) \in T_f$ per ipotesi.

Si noti che $S_1^1(\alpha,n)$ è funzione solamente di n poiché α è un programma fissato.

Allora possiamo applicare il teorema di ricorsione, in base al quale esiste un $\overline{n}$ tale che:

$$\begin{aligned}\varphi_{\overline{n}}(x) &= \underbrace{\varphi_{S_1^1(\alpha,\overline{n})}(x)}_{\text{Per teorema di ricorsione}} \\ &= \varphi_\alpha\left(\overline{n}, x\right) \\ &= \varphi_u\left(x, \langle \overline{n}, \varphi_u(x,1)\rangle\right) \\ &= \varphi_x\left(\langle \overline{n}, \varphi_x(1)\rangle\right)\end{aligned}$$

Ottenendo quindi l'equazione del primo punto.

Passiamo ora al secondo punto:

Ora osserviamo che A, così come è definito, non coinvolge gli indici delle funzioni φ_i, pertanto lo riscriviamo utilizzando l'equazione sopra descritta (e di cui $\overline{n}$ è soluzione). Procediamo quindi scrivendo:

$$A = \{x \mid \varphi_x\left(\langle \overline{n}, \varphi_x(1)\rangle\right) \neq \perp\}$$

Ora, A rispetta le funzioni per costruzione (essendo costruito come l'insieme su cui la funzione è definita), ed inoltre è immediato verificare che non si tratta né dell'insieme vuoto né di $\mathbb{N}$ (poiché risulta possibile sia trovare almeno un $\overline{n}$ per cui $\exists x : \varphi_{\overline{n}} = \perp$, basta immaginare ad una funzione che con qualsiasi ingresso va in deadlock e quindi l'insieme non può essere $\mathbb{N}$, sia un $\overline{n}$ tale per cui $\exists x : \varphi_{\overline{n}} \neq \perp$, basti immaginare una funzione che ritorna sempre un valore costante e quindi l'insieme non può essere $\emptyset$.). Applicando il teorema di Rice deduciamo che esso non è certamente ricorsivo, e nemmeno A^C lo è, dato che la classe degli insiemi ricorsivi è chiusa per complemento.

Volendo controllare se A è ricorsivamente numerabile, possiamo sfruttare la definizione di insieme ricorsivamente numerabile come dominio di una funzione $f \in P_f$. Dovremo quindi mostrare che esiste un programma in particolare che, su input x, termina se $x \in A$, altrimenti va in loop.

In questo caso però risulta molto più comodo mostrare un programma che termina quando $x \notin A$, cioè quando $\varphi_n(x) = \perp$, mentre va in loop in caso contrario.

Tale programma è facilmente implementato richiamando l'interprete universale $\varphi_u(n,x)$, e dimostra che A^C è ricorsivamente numerabile.

Ne segue che A non può essere nemmeno ricorsivamente numerabile, poiché se lo fosse si avrebbe che sia A che A^C sarebbero ricorsivi, cosa che sappiamo non essere.

Parte II

Computabilità

8

Macchine di Turing deterministiche

Definizione 8.1 | Macchina di Turing

Una **macchina di Turing deterministica** ad un nastro è una 6-tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, \{q_s, q_n\})$ in cui:

1. Q è un insieme **finito di stati**
2. Σ è l'**alfabeto di input**
3. Γ è l'**alfabeto di lavoro**. Vale che $\Sigma \subset \Gamma$. Esiste in particolare, un simbolo speciale, **blank**, $B \in \Gamma \setminus \Sigma$.
4. $\delta : (Q \setminus \{q_s, q_n\}) \times \Gamma \longrightarrow Q \times (\Gamma \setminus \{\text{blank}\}) \times \{-1, 1\}$ è la **funzione di transizione**
5. $q_0 \in Q$ è lo **stato iniziale**
6. $\{q_s, q_n\} \subseteq Q$ è l'insieme degli **stati finali**. In particolare, q_s è detto **accettante**, mentre q_n è detto **non accettante**.

Definizione 8.2 | Computazione

Una sequenza di *mosse* è detta **computazione di M su input x**. Tale computazione può essere finita o infinita, secondo che si giunga o meno ad uno dei due stati finali q_s e q_n.

Definizione 8.3 | Computazione accettante

La computazione di M su x è **accettante** se termina nello stato q_s. In tal caso, diciamo che M **accetta** x.

Se invece la computazione termina nello stato q_n oppure non termina, diciamo che M **non accetta** x.

Definizione 8.4 | Configurazione

Una **configurazione** di M è una descrizione, ad ogni dato istante:

1. dello stato assunto dal controllo
2. dalla posizione della testina sul nastro
3. dal contenuto del nastro

Diciamo che M si trova nella configurazione $\langle q, k, w \rangle$ se:

1. Il controllo è nello stato q
2. La testina è posizionata sulla k-esima cella del nastro
3. La porzione non blank del nastro è rappresentata dalla stringa $w \in \Gamma^*$

Una configurazione può essere denotata da uqv, intendendo cioè che:

1. M si trova nello stato q
2. Il contenuto non blank del nastro di lavoro è la stringa $uv \in \Gamma^*$
3. La testina sta scandendo il primo carattere di v

Definizione 8.5 | Linguaggio accettato

Il linguaggio L_M **accettato o riconosciuto** da M è definito come l'**insieme delle stringhe in Σ^* accettate da M**. Si tratta dell'insieme:

$$L_M = \{x \in \Sigma^* \mid M \text{ accetta } x\}$$

Definizione 8.6 | Configurazione iniziale

La **configurazione iniziale** di M su input x è data da $c_0 = \langle q_0, 1, x \rangle$ o equivalentemente $c_0 = q_0 x$.

Definizione 8.7 | Configurazione accettante

Data una configurazione c, denotiamo con $q(c)$ lo stato contenuto in c. c è detta **accettante** (non accettante) se $q(c) = q_s$ $(q(c) = q_n)$.

Definizione 8.8 | Configurazione successiva

Date due configurazioni c' e c'', con $q(x') \in \{q_s, q_n\}$, diciamo che c'' è la configurazione successiva a c' (e scriviamo $c'c''$) se M da c' a c'' in una mossa.

Definizione 8.9 | Configurazione d'arresto

Le configurazioni accettanti e non accettanti non hanno successori e sono, quindi, da ritenersi **configurazioni d'arresto**.

Osservazione 8.1 | Perché sono deterministiche?

L'aggettivo deterministico esprime il fatto che per ogni configurazione di M esiste al più una configurazione successiva univocamente determinata dalla funzione di transizione di M.

8.1 Macchina di Turing e Problemi di Decisione

Teorema 8.1 | Insiemi ricorsivamente numerabili e macchine di Turing

Sia $A \subseteq \mathbb{N}$. Allora A è ricorsivamente numerabile se e solo se esiste una macchina di Turing deterministica M tale che $L_M = L_A$

La classe degli insiemi ricorsivamente numerabili coincide con la classe degli insiemi riconosciuti da macchine di Turing deterministiche.

In altre parole, una macchina di Turing deterministica può essere vista come una **procedura** che accetta le stringhe appartenenti ad un insieme ricorsivamente numerabile.

Teorema 8.2 | Insiemi ricorsivi e macchine di Turing

Sia $A \subseteq \mathbb{N}$. Allora A è ricorsivo se e solo se esiste un algoritmo deterministico per L_A

Vale a dire che classe degli insiemi ricorsivi coincide con la classe degli insiemi riconosciuti da algoritmi deterministici.

8.2 Macchine di Turing per il calcolo di funzioni

Definizione 8.10 | Macchina di Turing che calcola funzione parziale

Oltre che come riconoscitori di linguaggi, le macchine di Turing deterministiche possono essere viste come dispositivi per il calcolo di funzioni.

Diciamo che la macchina di Turing deterministica M calcola la funzione parziale $f : \Sigma^* \longrightarrow \Gamma^*$ se:

1. Su ogni stringa $x \in \Sigma^*$ su cui f è definita, la computazione di M su x si arresta in una configurazione avente $f(x) \in \Gamma^*$ sul nastro di lavoro.
2. Su ogni stringa $x \in \Sigma^*$ su cui f non è definita, la computazione di M su x non si arresta.

Osservazione 8.2 | Le macchine di Turing sono un sistema di programmazione accettabile

È ben noto che le macchine di Turing deterministiche costituiscono un **sistema di programmazione accettabile**. Infatti:

1. La classe delle funzioni calcolate da macchine di Turing deterministiche coincide con quella delle funzioni **ricorsive parziali**.
2. Esiste una macchina di Turing deterministica **universale**, in grado, cioè, di simulare ogni altra macchina di Turing deterministica.
3. Vale il **Teorema** S_n^m

Pertanto in tale contesto vale la Tesi di Church per le macchine di Turing.

Teorema 8.3 | Tesi di Church per le macchine di Turing

La classe delle funzioni intuitivamente calcolabili coincide con la classe delle funzioni calcolate da macchine di Turing deterministiche.

8.3 Tempo di calcolo

Definizione 8.11 | Complessità del caso pessimo nel tempo

Sia M una macchina di Turing deterministica. La complessità in tempo di M su input di lunghezza n è data dalla funzione $t_M : \mathbb{N} \longrightarrow \mathbb{N}$ tale che:

$$t_M(n) = \max\{T_M(x) \mid x \in \Sigma^n\}$$

Definizione 8.12 | DTIME

Si indica con $\mathrm{DTIME}(n)$ la classe dei linguaggi accettati da macchine di Turing deterministiche in tempo $O(n)$.

Definizione 8.13 | Classe dei linguaggi accettati da macchine di Turing

Per ogni funzione $t : \mathbb{N} \longrightarrow \mathbb{N}$, $\mathrm{DTIME}(t(n))$ è la classe dei linguaggi accettati da macchine di Turing deterministiche in tempo $t(n)$.

Proposizione 8.1 |

Per ogni funzione $t : \mathbb{N} \longrightarrow \mathbb{N}$ tale che $\lim_{n\to+\infty} \frac{t(n)}{n} = +\infty$ e per ogni costante $c > 0$, vale:

$$\mathrm{DTIME}(c \cdot t(n)) = \mathrm{DTIME}(t(n))$$

Teorema 8.4 | Tesi di Church Estesa

Un problema di decisione è effettivamente risolubile se ammette un algoritmo deterministico polinomiale in tempo.

Teorema 8.5 | Complessità di macchine di Turing a uno e k nastri

Se un linguaggio L è riconoscibile in tempo $t(n)$ da una macchina di Turing deterministica a $k > 1$ nastri, allora L è riconoscibile da una macchina di Turing deterministica a un nastro in tempo $O\left(t^2(n)\right)$

Dimostrazione 8.1 | Complessità di macchine di Turing a uno e k nastri

Sia M una macchina di Turing deterministica a k nastri che riconosce L in tempo $t(n)$ e sia Γ l'alfabeto di lavoro di M.

Possiamo definire un nuovo alfabeto di lavoro $\Gamma' = (\Gamma \times \{0,1\})^k$. Così, in maniera del tutto ovvia, una stringa $x \in \Gamma'^*$ può descrivere il contenuto delle porzioni non blank dei k nastri di M e la posizione delle corrispondenti testine.

Infatti, se $x_j = ((a_1, l_1), \ldots, (a_k, l_k))$ è la j-esima lettera di x, allora ogni a_i è il contenuto della j-esima cella nel nastro i-esimo, con $1 \leqslant i \leqslant k$, mentre $l_i = 1$ se e solo se su tale cella è posizionata la relativa testina.

Definiamo ora una nuova macchina di Turing deterministica M' ad un solo nastro, avente Γ' come alfabeto di lavoro, che simula M. La computazione di M' su un generico input è suddivisa in fasi, una per ogni mossa della macchina M.

In ogni fase, M' scorre la porzione non blank del suo nastro dalla prima all'ultima posizione e riporta quindi la testina sulla prima cella. **Durante questa doppia passata**, la macchina aggiorna opportunamente il contenuto del nastro in modo da simulare un passo di M: nella prima scansione si aggiornano le componenti corrispondenti agli spostamenti delle testine di M verso destra, mentre nella seconda si aggiornano quelli relativi agli spostamenti verso sinistra.

Inoltre, al termine di ogni fase, M' ha memorizzato nello stato il simbolo letto da ciascuna testina di M in modo da determinare la mossa da simulare nella fase successiva, cioè i nuovi simboli da stampare e gli spostamenti delle testine.

Poiché M in $t(n)$ passi può visitare al più $t(n)+1$ celle su ogni nastro, ogni fase di M' su input di dimensione n può essere eseguita in al più $2 \cdot t(n) + 1$ passi. Ne segue che il numero totale di mosse compiute da M' è al più uguale a $2 \cdot t^2(n) + t(n)$.

8.4 Spazio di lavoro

Definizione 8.14 | Spazio di lavoro

Lo **spazio** è inteso come quantità di memoria occupata durante la computazione.

Definizione 8.15 | Complessità nello spazio

Data la macchina di Turing deterministica $M = \langle Q, \Sigma, \Gamma, \delta, q_0, \{q_s, q_n\}\rangle$ ed una stringa $x \in \Sigma^*$, sia $S(x)$ il numero di celle diverse dal nastro visitate durante la computazione di M su x. Poniamo $S(x) = +\infty$ se M visita infinite celle. La **complessità nello spazio nel caso pessimo** di M è la funzione $s : \mathbb{N} \longrightarrow \mathbb{N}$ definita come:

$$s(n) = \max\{S(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

Diciamo che il linguaggio $L \subseteq \Sigma^*$ è riconosciuto in spazio $f(n)$ se esiste una macchina di Turing deterministica M tale che:

1. $L = L_M$
2. M ha complessità in spazio $s(n) \leqslant f(n)$

La complessità in spazio di L è la minima complessità in spazio per una macchina di Turing deterministica che accetta L.

Definizione 8.16 | DSPACE

Si definisce $\mathrm{DSPACE}(f(n))$ la classe dei linguaggi riconosciuti da macchine di Turing deterministiche (con nastri di input e di lavoro separati) in spazio $O(n)$.

Proposizione 8.2 |

Per ogni costante $c > 0$ vale:

$$\mathrm{SPACE}(c \cdot s(n)) = \mathrm{DSPACE}(s(n))$$

Teorema 8.6 | DTIME e DSPACE

Per ogni funzione $f : \mathbb{N} \longrightarrow \mathbb{N}$, vale:

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$$

Dimostrazione 8.2 | DTIME e DSPACE

È sufficiente osservare che una macchina di Turing deterministica che lavora in tempo $f(n)$ può visitare al più $f(n)$ celle diverse sul nastro di lavoro, una nuova cella per ogni mossa.

9

La classe NP

Definizione 9.1 | SODD

Il problema è costituito da:

Istanza: una formula $\varphi(x_1, \ldots, x_n)$

Questione: φ è soddisfacibile? Esiste un assegnamento $\underline{\boldsymbol{\alpha}} \in \{0,1\}^n$ tale che $\varphi(\underline{\boldsymbol{\alpha}}) = 1$?

Osserviamo che:

1. Se φ è soddisfacibile, allora esiste un assegnamento $\underline{\boldsymbol{\alpha}}$ che dimostra la soddisfacibile, cioè tale che $\varphi(\underline{\boldsymbol{\alpha}}) = 1$. La dimensione $|\underline{\boldsymbol{\alpha}}|$ del "testimone" $\underline{\boldsymbol{\alpha}}$ è al più la dimensione $|\varphi|$ della formula. Inoltre verificare $\varphi(\underline{\boldsymbol{\alpha}}) = 1$ dato un $\underline{\boldsymbol{\alpha}}$ è fattibile in tempo polinomiale $|\varphi|^{O(1)}$.
2. Se φ non è soddisfacibile, allora nessun assegnamento soddisfa φ.

Definizione 9.2 | Problema NP

Un linguaggio $L \subseteq \Sigma^*$ è nella classe NP se esiste una relazione $V(x,y)$ calcolabile in tempo polinomiale ed un polinomio $p(n)$ per cui:

$$L = \{x \mid \exists y : (p(|x|)) \wedge (V(x,y) = 1)\}$$

Vale cioè che:

1. Se $x \in L$ allora esiste una "breve" dimostrazione di appartenenza y, cioè $|y| = p(|x|)$ e $V(x,y) = 1$. Inoltre il calcolo $V(x,y)$ può essere fatto in tempo polinomiale.
2. Se $x \notin L$ allora non esiste alcuna dimostrazione di lunghezza $p(|x|)$, cioè se $|y| = p(|x|)$ allora $V(x,y) = 0$

9.1 Classificazione dei problemi: concetto di riduzione

Definizione 9.3 | Riduzione polinomiale multi-uno

Dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$ diremo che L_1 è **polinomialmente riducibile multi-uno** a L_2 se esiste una funzione $f : \Sigma^* \longrightarrow \Sigma^*$ tale che:

1. f è calcolabile in tempo polinomiale.
2. $x \in L_1$ se e solo se $f(x) \in L_2$

Scriveremo in tal caso $L_1 <_p L_2$. Se ulteriormente f è calcolabile in spazio logaritmico, diremo che L_1 è **riducibile multi-uno in spazio logaritmico** a L_2 scrivendo $L_1 <_l L_2$.

Teorema 9.1 |

Dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$, allora:

1. se $L_1 <_p L_2$ e $L_2 \in \mathsf{P}$ allora $L_1 \in \mathsf{P}$
2. se $L_1 <_l L_2$ e $L_2 \in \mathsf{NC}$ allora $L_1 \in \mathsf{NC}$

Dove NC è la classe dei problemi efficientemente risolubili con algoritmi paralleli.

9.2 Problemi P-completi e NP-Completi

Definizione 9.4 | P-Completo

Un problema A è detto **P-completo** se:

1. $A \in \mathrm{P}$
2. $\forall X \in \mathrm{P}\,(X <_l A)$

Definizione 9.5 | NP-Completo

Un problema B è detto **NP-completo** se:

1. $B \in \mathrm{NP}$
2. $\forall X \in \mathrm{NP}\,(X <_p B)$

Teorema 9.2 |

Dati due problemi A e B, allora:

1. Se A è P-completo e $A \in NC$, allora $NC = P$.
2. Se B è NP-completo e $B \in P$, allora $P = NP$

Parte III

Note in preparazione all'esame

10

Lista dei teoremi

10.1 Computabilità

Teorema di Cantor L'insieme $\mathbb{N}^{\mathbb{N}}$ non è numerabile.

Funzioni coppia di Cantor Si tratta del teorema che illustra l'esistenza di una funzione tra $\mathbb{N}$ ed $\mathbb{N} \times \mathbb{N}$, che sono le funzioni coppia.

Correttezza del compilatore Viene costruita una traduzione tra linguaggio while e RAM e viene mostrato che questa è valida per ogni programma.

Conseguenza di correttezza del compilatore Mostra che il set delle funzioni realizzabili in linguaggio while è compatibile anche con il linguaggio RAM.

Interprete e funzione universale Introduce il concetto di codifica di un programma, di interprete, e di funzione che simula un formalismo usando le istruzioni di una codifica: la funzione universale.

Da programma RAM a While Il teorema mostra che è possibile costruire, per ogni programma RAM, un programma While che svolge la stessa funzione.

Funzioni ricorsive parziali e programmi while Mostra che l'insieme delle funzioni ricorsive parziali è computabile da programmi while.

Da programmi while a funzioni ricorsive parziali Mostra che l'insieme dei comandi while coincide con una funzione ricorsiva parziale.

Equivalenza tra insiemi di funzioni Mostra che tutti gli insiemi di funzioni sono riconducibili alle funzioni ricorsive parziali.

Tesi di Church La classe delle funzioni intuitivamente calcolabili è la classe delle funzioni ricorsive parziali.

Problema dell'arresto Illustra come il problema dell'arresto non sia decidibile.

Teorema S_1^1 Significa che dato un programma P che lavora su due variabili, è sempre possibile generare automaticamente un programma P' che lavora su una variabile fissando l'altra.

Teorema di Ricorsione Fissato un sistema di programmazione accettabile, per qualsiasi funzione ricorsiva totale t esiste un programma la cui semantica non cambia sotto t.

Insieme ricorsivamente enumerabile Un insieme ricorsivamente enumerabile è il dominio di una funzione ricorsiva parziale ed è definibile tramite una relazione ricorsiva.

Teorema di Rice Il teorema stabilisce che un insieme che rispetta le funzioni è ricorsivo solamente in casi banali: non risulta possibile dimostrare in modo automatico la correttezza dei programmi.

10.2 Computabilità

Definizione di una macchina di Turing As title.

Insiemi ricorsivamente numerabili e macchine di Turing

Insiemi ricorsivi e macchine di Turing

Tesi di Church per le macchine di Turing La classe delle funzioni intuitivamente calcolabili coincide con la classe delle funzioni calcolate da macchine di Turing deterministiche.

Tesi di Church Estesa Un problema di decisione è effettivamente risolubile se ammette un algoritmo deterministico polinomiale in tempo.

Complessità di macchine di Turing a uno e k nastri Illustra la relazione tra la complessità delle macchine di Turing a uno e quelle a k nastri.