

INFORMATICA TEORICA

giosumarin

March 2021

Contents

1	Lezione 1	2
1.1	Definizione di funzione	2
1.2	Funzione iniettiva, suriettiva e biettiva	3
2	Lezione 2	4
3	Lezione 3	6
3.1	\mathbb{R} non è numerabile	6
3.2	Cosa è calcolabile?	7
4	Lezione 4	8
4.1	$Dati \sim \mathbb{N}$	8
4.1.1	$DATI \simeq \mathbb{N}$	10
5	Lezione 5	10
5.1	Ingredienti della definizione formale di semantica	11
6	Lezione 6	13
6.1	$PROG \sim \mathbb{N}$ su programmi <i>RAM</i>	13
6.2	Come aritmetizzare?	13
6.3	Sistema di calcolo <i>WHILE</i>	14
6.3.1	Dimostrazioni induttive su alberi bianri	14
7	Lezione 7	15
7.1	Esecuzione su una macchina <i>WHILE</i> (intuitivamente)	15
7.2	Definizione formale di semantica di un programma <i>WHILE</i>	16
7.3	Potenza computazionale sistema <i>WHILE</i>	17
7.3.1	Relazione tra $F(RAM)$ e $F(WHILE)$?	17
7.3.2	Confronto tra due sistemi di calcolo	17
7.4	Concetto di traduttore	17
8	Lezione 8	18
8.1	Costruzione induttiva di <i>comp</i>	18
8.2	$F(RAM) \subseteq F(WHILE)$	19

9	Lezione 9	20
9.1	Interprete WHILE di programmi RAM	20
9.1.1	Variabili	20
9.1.2	Codice interprete in macrowhile	20
9.1.3	Conseguenze	20
9.1.4	Riflessioni concetto di calcolabilità	22
10	Lezione 10	22
10.1	Chiusura di insiemi rispetto ad operazioni	22
10.1.1	Chiusura	22
10.1.2	Chiusura di un'insieme	23
10.1.3	Chiusura di un insieme rispetto a un insieme di operazioni	24
10.1.4	Verso una definizione teorica di calcolabilità	24
11	Lezione 11	25
11.1	<i>ELEM</i> : nucleo delle funzioni calcolabili	25
11.2	Operatore di composizione di funzioni	25
11.2.1	Ampliamo <i>ELEM</i> chiudendo rispetto a <i>COMP</i>	25
11.3	Operatore di ricorsione primitiva	25
11.3.1	Ampliamo $ELEM^{COMP}$ chiudendo rispetto a <i>RP</i>	26
11.4	<i>RICPRIM</i> vs <i>WHILE</i>	26
11.4.1	Road map	26
11.4.2	Dimostrazione induttiva	27
11.4.3	Considerazioni su <i>RICPRIM</i>	28
12	Lezione 12	28
12.1	Operatore di minimalizzazione di funzioni	28
12.2	La classe <i>P</i> delle funzioni ricorsive parziali	29
12.2.1	Teorema $P \subseteq F(WHILE)$	29
12.2.2	Teorema $F(WHILE) \subseteq P$	30
12.2.3	$F(WHILE) \subseteq P$ dimostrando che $f_c \in P$	31
12.3	Tesi di Church-Turing	31
13	Lezione 13	32
13.1	Proprietà auspicabili per un sistema di programmazione	32
13.2	Sistemi di programmazione accettabili (spa)	33
13.2.1	Esistenza di compilatori tra spa	34

1 Lezione 1

1.1 Definizione di funzione

Funzione: una legge/regola che ci dice come associare un elemento di A a uno di B .

Definizione globale: $f : A \rightarrow B$: chiamiamo A il dominio della funzione e B il codominio.

Definizione locale: $a \rightarrow^f b$ oppure $f(a) = b$ con b immagine di a rispetto a f e a controimmagine di b rispetto a f .

$f : \mathbb{N} \rightarrow \mathbb{N}$ con $\mathbb{N} = \{0, 1, 2, 3, 4, \dots\}$ e con $\mathbb{N}^+ = \{1, 2, 3, 4, \dots\}$

Globale: $f(n) = \lfloor \sqrt{n} \rfloor$; Locale: $f(5) = \lfloor \sqrt{5} \rfloor$ In una funzione, per definizione, un valore del dominio può portare a uno solo valore di codominio.

1.2 Funzione iniettiva, suriettiva e biettiva

Funzione Iniettiva

$f : A \rightarrow B$ è **iniettiva** sse $\forall a_1, a_2 \in A \Rightarrow f(a_1) \neq f(a_2)$

ovvero non ci sono confluenze verso un punto del codominio.

Funzione Suriettiva

$f : A \rightarrow B$ è **suriettiva** sse $\forall b \in B, \exists a \in A : f(a) = b$

Definiamo con Im_f l'insieme delle immagini. Quindi

$$\{Im_f = b \in B : \exists a \in A. f(a) = b\} = \{f(a), a \in A\}$$

Possiamo quindi dire che in generale $Im_f \subseteq B$ ed è suriettiva sse $Im_f = B$, ovvero quando il grafico della funzione copre tutto l'asse y .

Funzione Biettiva Una funzione si dice biettiva quando è sia iniettiva che suriettiva, ovvero

$$\forall b \in B \exists! a \in A : f(a) = b$$

dove con $\exists!$ indichiamo "esiste unico".

Composizione di funzioni Nota: non è commutativo

$$f : A \rightarrow B$$

$$g : B \rightarrow C$$

$$f \text{ composto } g: g \cdot f : A \rightarrow C$$

$$\text{definita come } g \cdot f(a) = g(f(a))$$

Funzione Inversa

$$f : A \rightarrow B \text{ biettiva}$$

$$f^{-1} : B \rightarrow A \text{ t.c. } f^{-1}(b) = a \leftrightarrow f(a) = b$$

Definiamo

$$i_A : A \rightarrow A \text{ con } i_A(a) = a$$

che ci permette di dare una definizione ulteriore di funzione inversa combinando la funzione identità e la composizione

$$f^{-1} \cdot f = i_A \wedge f \cdot f^{-1} = i_B$$

2 Lezione 2

$f(a) \downarrow$: f **definita** $\forall a \in A$ si dice che f è **totale**

$f(a) \uparrow$: **non definita per ogni** $a \in A$.

$f : A \rightarrow B$ è parziale se qualche elemento di A associa un elemento di AB , infatti:

$$Dom_f = \{a \in A : f(a) \downarrow\} \subseteq A$$

$$Dom_f \subsetneq A \Rightarrow f \text{ **parziale (incluso stretto)**}$$

$$Dom_f = A \Rightarrow f \text{ **totale**}$$

Totalizzare

$$f : A \rightarrow B \text{ **parziale**} \Rightarrow \tilde{f} : A \rightarrow B \cup \{\perp\} \text{ **totale**},$$

$$\text{Indichiamo } B \cup \{\perp\} \rightarrow B_\perp$$

$$\tilde{f} = \begin{cases} f(a) & \text{se } a \in Dom_f \\ \perp & \text{altrimenti} \end{cases}$$

Prodotto Cartesiano

$$A \times b = \{(a, b) : a \in A \wedge b \in B\}$$

Nota: \times **non commutativa** $A \times B \neq B \times A$

Proiettore -iesimo $\pi_i : A_1 \times \dots \times A_n \rightarrow A_i$

$$\pi_i(a_1, \dots, a_n) = a_i$$

Indichiamo A per n volte come $A \times \dots \times A = A^n$

Insieme di funzioni

$$B^A = \{f : A \rightarrow B\} = \text{insieme delle funzioni da } A \text{ a } B$$

$$B_\perp^A = \{f : A \rightarrow B\} = \text{insieme delle funzioni parziali da } A \text{ a } B$$

Funzione di valutazione Dati A, B e B_\perp^A si definisce funzione di valutazione

$$w : B_\perp^A \times A \rightarrow B \text{ con } w(f, a) = f(a)$$

Fissando a eseguo un benchmark di funzioni, fissando f creo i punti del grafico di f .

Sistema di calcolo C Abbiamo $P \in \text{PROG}$ che è una sequenza di regole che trasforma un dato di input in un dato di output $\Rightarrow P \in \text{DATI}_\perp^{\text{DATI}}$ è una funzione (in un linguaggio).

$$C : \text{DATI}_\perp^{\text{DATI}} \times \text{DATI} \rightarrow \text{DATI}_\perp$$

dove $C(P, x)$ è la funzione calcolata da P

P è un oggetto semantico/rappresentazione, se faccio girare ho una funzione.

Potenza computazionale di C

$$F(C) = \{C(P, _) : P \in \text{PROG}\} \subseteq \text{DATI}_{\perp}^{\text{DATI}}$$

$$F(C) = \text{DATI}_{\perp}^{\text{DATI}} \Rightarrow \text{informatica può tutto}$$

$$F(C) \subsetneq \text{DATI}_{\perp}^{\text{DATI}} \Rightarrow \text{esistono compiti non automatizzabili}$$

Cardinalità Indichiamo con $|A|$ il numero di elementi di A . Ha senso però solo su insiemi infiniti. Infatti $|\mathbb{N}| = \aleph_0 = |\mathbb{R}|$ risultano equinumerosi, che me ne faccio? In realtà, l'infinito di \mathbb{N} è meno fitto di quello di \mathbb{R} .

Relazione Relazione binaria su $A : R \subseteq A^2$. Elementi $a, b \in A$ sono nella relazione R sse $(a, b) \in R$ che si può anche indicare con aRb .

Relazione di equivalenza sse:

- Riflessiva, $\forall a : aRa$
- Simmetrica, $\forall a, b : aRb = bRa$
- Transitiva, $aRb \wedge bRc \rightarrow aRc$

Relazioni di equivalenza e partizioni $A : R \subseteq A^2$ induce partizione su $A \Rightarrow A_1, A_2, \dots \subseteq A$ t.c.

- $A_i \neq \emptyset$;
- $i \neq j \Rightarrow A_i \cap A_j = \emptyset$;
- $\cup_{i \in I} A_i = A$.

Data $a \in A$ la sua classe di equivalenza è $[a]_R = \{b \in A : aRb\}$.

Si dimostra che:

- Non esistono classi di equivalenza vuote (per riflessività ho almeno dentro me stesso);
- dati $a, b \in A \Rightarrow [a]_R \cap [b]_R = \emptyset$ o $[a]_R = [b]_R$
- $\cup_{a \in A} [a]_R = A$

L'insieme delle classi di equivalenza spezzetta A . L'insieme A visto come partizioni è detto quoziente di A rispetto a R e si indica con A/R .

Cardinalità di insiemi Sia U la classe di tutti gli insiemi. Definisco $\sim \subseteq U^2$ come $A \sim B$ sse esiste biezionone tra A e B (associazione 1 a 1 tra elementi di A e B).

Proprietà di \sim :

- riflessiva (uso funzione identità di A (i_A));

- simmetrica: se $A \sim B$ allora $B \sim A$ con la funzione inversa (con biezione esiste per forza);
- transitiva: composizione di biettiva è biettiva.

Se $A \sim B$ i due insiemi sono equinumerosi. Un insieme si dice numerabile sse $A \sim \mathbb{N}$.

3 Lezione 3

Definiamo un insieme non numerabile un insieme a cardinalità infinita ma non "listabili esaustivamente" come \mathbb{N} , sono più fitti e se provo a listare mi perdo qualche elemento.

3.1 \mathbb{R} non è numerabile

Proviamo a dimostrare che non c'è biezione tra \mathbb{N} e \mathbb{R} :

1. dimostro che $\mathbb{R} \sim [0, 1]$, ovvero che $[0, 1]$ è fitto come \mathbb{R} ;
2. dimostro che $\mathbb{N} \not\sim [0, 1]$
3. $\mathbb{N} \not\sim [0, 1] \Rightarrow \mathbb{N} \not\sim \mathbb{R}$

$\mathbb{R} \sim [0, 1]$

- scelgo un punto su $[0, 1]$
- proietto sulla semicirconferenza centrata in $\frac{1}{2}$
- traccio linea tra $\frac{1}{2}$ e il punto proiettato

La funzione è iniettiva in quanto ogni punto crea un punto diverso (cambia l'angolo); è anche suriettiva tramite l'operazione inversa. Possiamo quindi dire che $\mathbb{R} \sim [0, 1]$.

$\mathbb{N} \not\sim [0, 1]$ Dimostrazione per assurdo: $\mathbb{N} \sim [0, 1]$, quindi $[0, 1]$ è listabile.

$$\begin{array}{cccccc}
 0. & \underline{a_{11}} & a_{12} & a_{13} & a_{14} & \dots \\
 0. & a_{21} & \underline{a_{22}} & a_{23} & a_{24} & \dots \\
 0. & a_{31} & a_{32} & \underline{a_{33}} & a_{34} & \dots \\
 0. & a_{41} & a_{42} & a_{43} & \underline{a_{44}} & \dots \\
 \dots & \dots & \dots & \dots & \dots & \dots
 \end{array}$$

1 posso sciverso come $0.\overline{9}$. Costruiamo ora $0, c_1, c_2, \dots, c_i, \dots$

$$c_i = \begin{cases} a_{ii} + 1 & \text{se } a_{ii} < 9 \\ a_{ii} - 1 & \text{se } a_{ii} = 9 \end{cases}$$

c non è nessuno della lista perchè differisce per la i -esima componente, differisce dal primo perchè $c_1 \neq a_{11}$, dal secondo perchè $c_2 \neq a_2$ e così via. Possiamo quindi dire che $\mathbb{N} \not\sim [0, 1]$.

Quindi $\mathbb{N} \not\sim \mathbb{R}$, di conseguenza \mathbb{R} non è numerabile ed è un'insieme continuo: tutti gli insiemi equinumerosi a \mathbb{R} si dicono insiemi continui.

Insieme delle parti di \mathbb{N} $P(\mathbb{N} =$ sottoinsiemi di $\mathbb{N} \not\sim$ dimostrato per diagonalizzazione. Creo elenco di sottoinsiemi e trovo un sottoinsieme di \mathbb{N} che non c'è nell'elenco.

$$\begin{aligned}\mathbb{N} &\Rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6\ \dots} \\ A &\Rightarrow \mathbf{1\ 1\ 0\ 1\ 1\ 0\ \dots} \\ \text{dove } 1 &\Rightarrow \in A \text{ e } 0 \Rightarrow \notin A\end{aligned}$$

Per assurdo $P(\mathbb{N} \sim \mathbb{N} \Rightarrow$ listo esaustivamente

$$\begin{array}{ccccccccc} \underline{b_{01}} & b_{11} & b_{21} & b_{31} & \dots & & & & \\ \underline{b_{02}} & \underline{b_{12}} & b_{22} & b_{32} & \dots & & & & \\ b_{03} & \underline{b_{13}} & \underline{b_{23}} & b_{33} & \dots & & & & \\ \dots & \dots & \dots & \dots & \dots & & & & \end{array}$$

Considero ora il sottoinsieme di \mathbb{N} rappresentato dal vettore $\overline{b_{01}b_{12}b_{23}}\dots$ dove overline rappresenta il negato. Questo vettore è un sottoinsieme di $P(\mathbb{N})$ che non appartiene a \mathbb{N} .

$\mathbb{N}^{\mathbb{N}}$ Insieme non numerabile $\mathbb{N}^{\mathbb{N}} = \{f : \mathbb{N} \rightarrow \mathbb{N}\}$.

Anche in questo caso procedo per diagonalizzazione per ipotesi assurda. Metto sulle colonne i valori di N e sulle righe le funzioni.

$$\begin{array}{ccccccccc} \underline{f_0(0)} & f_0(1) & f_0(2) & f_0(3) & \dots & & & & \\ \underline{f_1(0)} & \underline{f_1(1)} & f_1(2) & f_1(3) & \dots & & & & \\ f_2(0) & \underline{f_2(1)} & \underline{f_2(2)} & f_2(3) & \dots & & & & \\ \dots & \dots & \dots & \dots & \dots & & & & \end{array}$$

Definisco $\phi : \mathbb{N} \rightarrow \mathbb{N}$ con $\phi(n) = f_n(n) + 1$. $\phi \in \mathbb{N}^{\mathbb{N}}$ e dovrebbe stare nella lista esaustiva ma non c'è quindi è un insieme continuo come l'insieme delle parti di \mathbb{N} .

3.2 Cosa è calcolabile?

Considerazioni ragionevoli:

- $PROG \sim \mathbb{N}$, considero la digitalizzazione di un programma, è un numero espresso in binario
- $DATI \sim \mathbb{N}$, come sopra

Quindi $F(C) \sim PROG \sim \mathbb{N} \not\sim \mathbb{N}_{\perp}^{\mathbb{N}} \sim DATI_{\perp}^{DATI}$. Esistono funzioni non calcolabili, pochi programmi e tante funzioni.

Table 1: Rappresentazione delle coppie di Cantor

		y			
		0	1	2	3
x	0	1	3	6	10
	1	2	5	9	
	2	4	8		
	3	7			

4 Lezione 4

4.1 *Dati* $\sim \mathbb{N}$

Forniamo una legge che:

- associ biunivocamente dati a numeri e viceversa;
- consente di operare direttamente per operare sui corrispettivi dati; che ci consenta di dire, senza perdita di generalizzazione, che i nostri programmi lavorano sui numeri.

Per fare ciò, passiamo attraverso un risultato matematico sulla cardinalità di insiemi. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+ \Rightarrow \mathbb{N} \times \mathbb{N} \sim \mathbb{N}$, da cui si può ottenere $\mathbb{Q} \sim \mathbb{N}$ considerando che possiamo vedere le frazioni $\in \mathbb{Q}$ come coppie di numeratore e denominatore ovvero $\mathbb{N} \times \mathbb{N}$.

Funzione coppia di Cantor Definiamo $\langle, \rangle: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}^+$ iniettiva e suriettiva. Abbiamo $\langle x, y \rangle = n$ con $\sin: \mathbb{N}^+ \rightarrow \mathbb{N}$ e $\text{des}: \mathbb{N}^+ \rightarrow \mathbb{N}$. Per il "ritorno" abbiamo quindi che $\sin(n) = x$ e $\text{des}(n) = y$.

Consideriamo una rappresentazione grafica come in Tabella 4.1, riempita con i numeri $\in \mathbb{N}^+$ seguendo la diagonale. Cantor è iniettiva perchè le coordinate di punti diverse individuano celle diverse che vengono riempite successivamente; suriettiva perchè riempio fino all' n voluto e guardo immagine $\langle x, y \rangle$ corrispondente. Per esempio $\langle 2, 1 \rangle = 8$.

Forma analitica di Cantor Come vediamo nella Tabella 4.1 troviamo il valore della coppia $\langle x, y \rangle$ sulla diagonale che inizia in $\langle x + y, 0 \rangle$.

1. $\langle x, y \rangle = \langle x + y, 0 \rangle + y$
2. trovo la coppia $\langle z, 0 \rangle = \sum_{i=1}^z \frac{z(z+1)}{2} + 1$

Il punto 2 è dato dal fatto che un generico valore nella colonna 0 è dato dalla somma degli indici fino a quello cercato +1, vediamo per esempio nella Tabella

Table 2: Rappresentazione analitica di cantor, la coppia $\langle x, y \rangle$ si trova sulla diagonale della riga $x + y$

		y			
		y	...
x	x	$\langle x, y \rangle$	
		
	$x + y$...			
	...				

4.1 che il valore 7 nella riga 3 è calcolabile come $3 + 2 + 1 + 0$ a cui aggiungiamo ancora 1. Unendo i due punti troviamo che

$$\langle x, y \rangle = \langle x + y, 0 \rangle + y = \frac{(x + y)(x + y + 1)}{1} + y + 1.$$

Come tornare a \mathbb{N}^+ e \mathbb{N}^+ Vogliamo capire come trovare sinistra e destra partendo da n .

1. trovare le coordinate $\langle \gamma, 0 \rangle$ del punto iniziale della diagonale dove si trova n ;
2. $y = n - \langle \gamma, 0 \rangle$ e $x = \gamma - y$,

Per il punto 1 possiamo dire che $\gamma = \max\{z \in \mathbb{N} : \langle z, 0 \rangle \leq n\}$, quindi

$$\begin{aligned} \langle z, 0 \rangle \leq n &\Rightarrow \frac{z(z+1)}{2} + 1 \leq n \\ &\Rightarrow z^2 + z + 2 - 2n \leq 0 \Rightarrow \text{eq 2° grado} \\ &\Rightarrow z_{1,2} = \frac{-1 \mp \sqrt{8n-7}}{2} \Rightarrow \text{solo } \leq 0 \\ &\Rightarrow \frac{-1 - \sqrt{8n-7}}{2} \leq z \leq \frac{-1 + \sqrt{8n-7}}{2} \\ &\Rightarrow \text{intero più grande} \Rightarrow \gamma = \lfloor \frac{-1 + \sqrt{8n-7}}{2} \rfloor; \end{aligned}$$

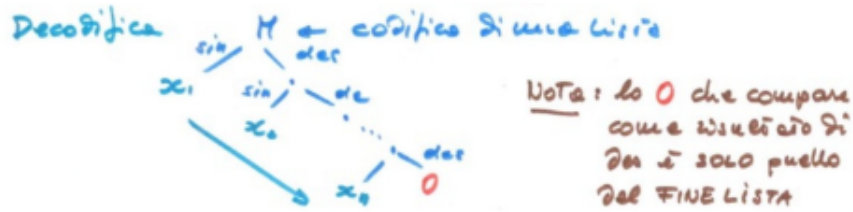
troviamo infine che $des(n) = y = n - \langle \gamma, 0 \rangle$ e $sin(n) = x = \gamma - y$.

Abbiamo quindi dimostrato $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+$, per dimostrare $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$ basta semplicemente definire una nuova funzione, ovvero

$$[,] : \mathbb{N} \times \mathbb{N} \sim \mathbb{N} \text{ t.c. } [x, y] = \langle x, y \rangle - 1$$

e possiamo notare che , mostra che \mathbb{Q} è numerabile.

Figure 1: Decodifica lista



4.1.1 DATI_{~N}

Liste di interi Codifichiamo x_1, \dots, x_n in $\langle x_1, \dots, x_n \rangle$. Ricordiamo che le liste non hanno lunghezza nota, quindi metto uno 0 a fine lista per capire che sono arrivato alla fine.

Codifica: $1, 2, 5 \Rightarrow \langle 1, 2, 5, 0 \rangle \Rightarrow \langle 1, \langle 2, \langle 5, 0 \rangle \rangle \rangle \Rightarrow \langle 1, \langle 2, 16 \rangle \rangle \Rightarrow \langle 1, 188 \rangle \Rightarrow 18144$.

Decodifica: Creo albero a partire da n , a sinistra troverò i vari x ordinati con in cima quello di indice inferiore e a destra o un sottoalbero o uno 0. Quando trovo 0 a destra mi fermo. Un esempio è mostrato in Figura 4.1.1.

Strutture dati derivanti Array(lunghezza nota):

$$x_1, \dots, x_n \Rightarrow [x_1, \dots, x_n] = [x_1, \dots, [x_{n-1}, x_n]] \dots]$$

Matrici:

$$\begin{matrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{matrix} \Rightarrow \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \Rightarrow [[a_{11}, a_{12}], [a_{21}, a_{22}]]$$

Grafi: utilizzando le liste di adiacenza o le matrici di adiacenza.

5 Lezione 5

Sistema di calcolo *RAM*: macchina *RAM* + linguaggio *RAM* (assembly semplificato). Consente di definire rigorosamente:

- $PROG \sim \mathbb{N}$
- $C(P, _) RAM(P, _)$, semantica dei programmi
- $F(RAM)$, potenza computazionale

Forse l'idea di potenza computazionale fornita in prima istanza ($F(RAM)$) è stringente in quando la macchina *RAM* è molto semplice, successivamente introdurremo la macchina *WHILE* (JVM) e confronteremo le loro potenze computazionali. Se avremo $F(_)$:

- diverse \Rightarrow ciò che è computabile dipende dallo strumento
- uguale \Rightarrow computabilità (tesi di Church)? posso calcolare stessi insiemi di funzioni?

Sistema di calcolo RAM La macchina *RAM* è composta da:

- L , program counter, indica indirizzo della prossima istruzione da eseguire;
- P , programma, formato da istruzioni;
- R , memoria, insieme di registri e ogni cella può contenere un numero $\in \mathbb{N}$, dove R_1 contiene l'input e R_0 l'output.

La terminazione è data da $L = 0$.

Output: $\phi_P(x) = \text{contenuto}(R_0)$ o \perp in caso di loop, indichiamo con ϕ_P la semantica di P .

Linguaggio RAM

- $R_k \leftarrow R_k + 1$
- $R_k \leftarrow R_k - 1; x \dot{-} y = x - y \text{ if } x \geq y \text{ else } 0$
- *if* $R_k = 0$ *then goto* $m; m = \{1, |P|\}; |P| = \text{numero di istruzioni di } P$

Semantica Operazionale Ovvero specificare il significato di ogni istruzione, e quindi dei programmi, specificando l'effetto che quell'istruzione ha sui registri della macchina.

Come descrivo l'effetto di un'istruzione? $S = \text{Stato} = \text{foto della macchina}$. Prendo S prima e dopo l'esecuzione di un'istruzione. $S_{init}, S_1, S_{fin}, P$ induce una sequenza di stati.

La semantica di P :

$$\phi_P = \begin{cases} y & \text{se finisce} \\ \perp & \text{altrimenti} \end{cases}$$

5.1 Ingredienti della definizione formale di semantica

Stato

$$S : \{L, R\} \rightarrow \mathbb{N}$$

$$\text{Stati} = \mathbb{N}^{\{L, R\}}$$

$S(R_k)$: contenuto del registro R_k quando la macchina è nello stato S

stato finale : $S(L) = 0 \Rightarrow \mathbf{HALT}$

dato : $\mathbb{N}(\text{infatti DATI} \sim \mathbb{N})$

Inizializzazione Dato il dato n prepara la macchina nello stato iniziale:

- $S_{init}(L) = 1$
- $S_{init}(R_1) = n$
- $\forall i \neq 1 : S_{init}(R_i) = 0$

Programmi $PROG = \{\text{programmi RAM}\}, P \in PROG; |P| = \# \text{istruzioni}$

Esecuzione Dinamica del programma \Rightarrow funzione stato prossimo.

$$\sigma : stati \times PROG \rightarrow stati_{\perp}; \sigma(S, P) = S'$$

Lo stato che segue lo stato S dopo l'esecuzione di un'istruzione di P :

- dipende dall'istruzione che devo eseguire
 - l'istruzione dipende da $S(L)$
1. se $S(L) = 0$ allora $S' = \perp$
 2. se $S(L) > |P|$ allora $S'(L) = 0$ e $\forall i : S'(R_i) = S(R_i)$
 3. se $1 \leq S(L) \leq |P|$: considero l'istruzione $S(L) -esima$:
 - $R_k \leftarrow R_k + /-1$:
 - $S'(R_k) = S(R_k) + /-1$
 - $S'(L) = S(L) + 1$
 - $\forall i \neq k : S'(R_i) = S(R_i)$
 - *if $R_k = 0$ then goto m :*
 - $S'(R_i) = S(R_i)$
 - $S'(L) = m$ if $S(R_k) == 0$ else $S(L) + 1$

Semantica di P

$$\phi_P : \mathbb{N} \rightarrow \mathbb{N}_{\perp}$$

$$\phi_P(n) = \begin{cases} y & \text{se } S_m(L) = 0 \text{ e } S_m(R_0) = y \\ \perp & \text{se va in loop} \end{cases}$$

Potenza computazionale di RAM $F(RAM) = \{f \in \mathbb{N}_{\perp}^{\mathbb{N}} : \exists P \in PROG, \phi_P = f\} = \{\phi_P : P \in PROG\} \subseteq \mathbb{N}_{\perp}^{\mathbb{N}}$.

Incluso stretto per intuizione.

6 Lezione 6

6.1 $PROG \sim \mathbb{N}$ su programmi RAM

Come codificare programmi in numeri e ritorno biunivocamente Appliciamo a ogni istruzione del programma un'aritmetizzazione e poi uniamo i vari numeri generati per creare un singolo numero tramite la codifica utilizzata con la lista di numeri + Cantor. Per il ritorno, sappiamo decodificare la lista finale; se l'aritmetizzazione (Ar) è invertibile allora da n posso ricostruire univocamente il sorgente.

Ci MANca solo il passaggio fatto da Ar : da istruzioni a numeri e viceversa. Questo passaggio si dice aritmetizzare o godelizzare.

6.2 Come aritmetizzare?

$Ar : \text{istruzione} \rightarrow \mathbb{N}$ t.c. $Ar(istr = n) \leftrightarrow Ar^{-1}(n) = istr$

$Ar(R_k \leftarrow R_k + 1) = 3k$

$Ar(R_k \leftarrow R_k - 1) = 3k + 1$

$Ar(\text{if } R_k = 0 \text{ then goto } m = 3 < k, m > - 1 \text{ come fare } + 2$

Com'è fatto Ar^{-1} Utilizzo il resto della divisione per 3, quindi $\|n\|_3$ (n modulo 3):

- 0: $n = 3k \Rightarrow R_{\frac{n}{3}} \leftarrow R_{\frac{n}{3}} + 1;$
- 1: $n = 3k + 1 \Rightarrow R_{\frac{n-1}{3}} \leftarrow R_{\frac{n-1}{3}} - 1;$
- 2: $n = < k, m > - 1 \Rightarrow < k, m > = \frac{n+1}{3} \Rightarrow \text{if } R_{\sin \frac{n+1}{3}} = 0 \text{ then goto } R_{\text{des} \frac{n+1}{3}}.$

Da programmi a numeri $\text{cod}(P) = < Ar(istr_1), \dots, Ar(istr_m) >$

Da numeri a programmi Come la decodifica destro/sinistro con le parti sinistra che "subiscono" Ar^{-1} , anche in questo caso ci fermiamo quando troviamo lo 0 nel lato destro.

Osservazioni

- i numero diventano linguaggio di programmazione;
- potrei scrivere $F(RAM) = \{\phi_P : P \in PROG\}$ come $F(RAM) = \{\phi_i\}_{i \in \mathbb{N}}$;
- per il sistema RAM si ha rigorosamente $F(RAM) \sim \mathbb{N}^{\mathbb{N}}$, quindi alcuni problemi non sono automatizzabili;
- forse, cosiderando un sistem adi calcolo C più sofisticato ma comunque rigorosamente trattabile come RAM , potremmo dare un'idea formale di "ciò che è calcolabile automaticamente" come $F(C)$ che sia più ampia di $F(RAM)$;

- se dimostriamo che $F(C) = F(RAM)$ allora cambiare tecnologia non cambia ciò che è calcolabile \Rightarrow la calcolabilità è intrinseca ai problemi: perchè non catturarla matematicamente? (no macchine, no linguaggio, ...).

6.3 Sistema di calcolo *WHILE*

Memoria x_0, \dots, x_{20} con x_0 output e x_1 input. Le variabili contengono numeri arbitrariamente grandi e di conseguenza in una singola variabile posso salvare, per esempio con Cator, più di un semplice numero.

Non abbiamo un program counter in quanto il linguaggio *WHILE* è strutturato.

Linguaggio *WHILE* Sintassi induttiva: ho delle fasi semplici, con passi induttivi faccio cose più complicate.

- [*BASE*]: comando assegnamento: $x_k = 0$, $x_k = x_j + 1$, $x_k = x_j - 1$;
- [*PASSI*]: comando while: while $x_k \neq 0$ do C , con C che può essere:
 - comando di assegnamento
 - comando while
 - comando composto
- [*PASSI*]: comando composto: BEGIN C_1, \dots, C_m END, con C come sopra.

Possiamo quindi dire che un programma *WHILE* è un comando composto.

$w - PROG = \{\text{programmi } WHILE\} \leftarrow$ costruiti induttivamente.

Semantica di w : $\psi_w : \mathbb{N} \rightarrow \mathbb{N}$

$w - \mathbf{PROG}$ Per dimostrare una proposizione su $w - PROG$:

1. dimostro proposizione sugli assegnamenti;
2. suppongo proposizione su C e la dimostro su $while x_k \neq 0 do C$;
3. suppongo vera la proposizione su C_1, \dots, C_m e la dimostro su BEGIN C_1, \dots, C_m END.

6.3.1 Dimostrazioni induttive su alberi bianri



Dividiamo i nodi in nodi interni e foglie.

1. *[BASE]*: \bullet è un albero binario



2. *[PASSO]*: se T_1 e T_2 sono alberi binari allora $T_1 \quad T_2$ è un albero binario

3. nient'altro è un albero binario

su ogni albero binario, il numero di nodi interni è minore di 1 rispetto alle foglie = **P** Per induzione:

1. *[BASE]*: \bullet , 1 foglia e 0 nodi interni \Rightarrow **VERO**
2. *[PASSO]*: suppongo vero **P** su T_1 e T_2 , ovvero suppongo vero:
 - \bullet T_1 : foglie F_1 , nodi interni $F_1 - 1$
 - \bullet T_2 : foglie F_2 , nodi interni $F_2 - 1$



Dimostro vero **P** per: $T_1 \quad T_2$ ovvero $F_1 + F_2$ foglie, e $F_1 - 1 + F_2 - 1 + 1 = F_1 + F_2 - 1$ nodi interni \Rightarrow **VERO**

Depth

1. *[BASE]*: $depth(\bullet) = 0$



2. *[BASE]*: $depth(T_1 \quad T_2) = 1 + \max(depth(T_1), depth(T_2))$

7 Lezione 7

7.1 Esecuzione su una macchina WHILE (intuitivamente)

1. inizializzazione: imposto le variabili x_0, \dots, x_{20} come $0, n, 0, \dots, 0$;
2. esecuzione: eseguo le istruzioni del programma w (non ho bisogno del program counter);
3. terminazione: quando le istruzioni di w terminano oppure ho un loop;
4. output: contenuto di x_0 , quindi $\Psi(n) = cont(x_0) / \perp$.

Semantica del programma $w \quad \Psi_w : \mathbb{N} \rightarrow \mathbb{R}_\perp$.

7.2 Definizione formale di semantica di un programma WHILE

- Stato: foto della macchina in un tempo t ovvero una tupla/vettore di 21 componenti: (c_0, \dots, c_{20}) con c_i contenuto della cella x_i .

$$W - Stati = \mathbb{N}^{21}, \text{ stato } \underline{x} \in \mathbb{N}^{21} \text{ (vettore } \underline{x})$$

- Dati: \mathbb{N}
- Inizializzazione: $w-in : \mathbb{N} \rightarrow \mathbb{N}^{21}$ con $w-in(n) = (0, n, 0, \dots, 0)$
- Semantica operazione:

$$[]() : w-comandi \times w-stati \rightarrow w-stati_{\perp}; [C](\underline{x}) = \underline{y}$$

con C comandando, \underline{y} è lo stato prossimo partendo da \underline{x} eseguendo C

Posso definire intuitivamente $[C](\underline{x})$ sulla struttura induttiva del comando C :

- $[BASE]$ Assegnamenti:

$$[x_k = 0](\underline{x}) = \underline{y} \text{ con } y_i = \begin{cases} x_i & \text{se } i \neq k \\ 0 & \text{se } i = k \end{cases}$$

$$[x_k = x_j + /-1](\underline{x}) = \underline{y} \text{ con } y_i = \begin{cases} x_i & \text{se } i \neq k \\ x_j + /-1 & \text{se } i = k \end{cases}$$

- $[PASSO]$ Comando composto: $[BEGIN C_1, \dots, C_m END]$, conosco $[C_i]$ per ipotesi induttiva, ovvero conosco cosa fa ogni singolo comando.

$$[C_m](\dots([C_2]([C_1](\underline{x})))\dots) = \underline{y} = [C_1] \cdot \dots \cdot [C_m](\underline{x})$$

- $[PASSO]$: Comando while: $[WHILE x_k \neq 0 DO C]$, conosco $[C_i]$ come sopra.

$$[C](\dots([C]([C](\underline{x})))\dots) = \underline{y}$$

Quante volte applico C ? Tante volte quanto serve per azzerare x_k dello stato risultante durante l'iterazione del comando C ovvero

$$\begin{cases} [C]^e(\underline{x}) \text{ con } e = \mu t (k\text{-esima componente di } [C]^{(t)}(\underline{x}) = 0) \\ \perp \text{ altrimenti} \end{cases}$$

dove μt è il minor numero di volte.

Semantica di w e $w\text{-prog}$ $\Psi_w(n) = \mathbf{Pr}(0, [w](w-in(n)))$; proiezione 0-esima ci restituisce l'output applicando $[w]$ allo stato prodotto dall'inizializzazione con $x_1 = n$.

7.3 Potenza computazionale sistema WHILE

$$F(WHILE) = \{f \in \mathbb{N}_\perp^\mathbb{N} : \exists w \in w-PROG, f = \Psi_w\} = \{\Psi_w : w \in w-PROG\}$$

7.3.1 Relazione tra $F(RAM)$ e $F(WHILE)$?

Che relazione esiste tra $F(WHILE)$ e $F(RAM) = \{\phi_P : P \in PROG\}$?

- $F(RAM) \subsetneq F(WHILE)$, sarebbe anche comprensibile vista la semplicità della macchina RAM ;
- Insiemi con intersezioni o disgiunti, sarebbe preoccupante perchè il concetto di calcolabile dipenderebbe dalla macchina;
- $F(WHILE) \subsetneq F(RAM)$, sarebbe sorprendente poichè $WHILE$ sembra più sofisticata di RAM ;
- $F(RAM) = F(WHILE) \Rightarrow$ calcolabile non dipende dalla tecnologia

7.3.2 Confronto tra due sistemi di calcolo

Poniamo di avere due sistemi di calcolo C_1 e C_2 con i relativi programmi C_1-PROG e C_2-PROG .

$$F(C_1) = \{f \in \mathbb{N}_\perp^\mathbb{N} : f = \Psi_{P_1} \text{ per qualche } P_1 \in C_1-PROG\} = \{\Psi_{P_1} : P_1 \in C_1-PROG\}$$

$$F(C_2) = \{f \in \mathbb{N}_\perp^\mathbb{N} : f = \phi_{P_2} \text{ per qualche } P_2 \in C_2-PROG\} = \{\phi_{P_2} : P_2 \in C_2-PROG\}$$

Come mostriamo che $F(C_1) \subseteq F(C_2)$ ovvero che il primo non supera il secondo? Dimostro che

$$\forall f \in F(C_1) \Rightarrow f \in F(C_2)$$

ovvero che per ogni elemento del primo insieme allora è anche nel secondo insieme (dimostrazione di inclusione).

Risolviamo questo problema con un traduttore, prende un programma in un linguaggio e lo traduce in un altro linguaggio; per esempio quando compilo un programma in $C++$ creo un assembly, questo implica che $C++$ è al più potente come assembly. Matematicamente possiamo descrivere un traduttore come

$$\exists P_1 \in C_1-PROG : f = \Psi_{P_1} \Rightarrow \exists P_2 \in C_2-PROG : f = \Phi_{P_2},$$

per ogni programma nel primo sistema ne esiste uno equivalente a un programma del secondo sistema.

7.4 Concetto di traduttore

Dati i sistemi C_1 e C_2 , una traduzione da C_1 a C_2 è una funzione $T : C_1-PROG \rightarrow C_2-PROG$ con le seguenti proprietà:

- $[Programmabile]$: è programabile;

- [Completa]: traduce ogni C_1 -*PROG* in C_2 -*PROG*;
- [Corretta]: mantiene la semantica $\forall P \in C_1$ -*PROG* : $\phi_{T(P)} = \Psi_P$, è la formalizzazione del concetto di compilatore (nota che $\phi_{T(P)}$ è l'oggetto e Ψ_P il sorgente).

Se esiste $T : C_1$ -*PROG* \rightarrow C_2 -*PROG* allora $F(C_1) \subseteq F(C_2)$. Dimostrazione:

$f \in F(C_1) \Rightarrow \exists P \in C_1$ -*PROG* : $f = \Psi_P$
 a P applico T , ottengo
 $T(P) \in C_2$ -*PROG* (completezza), con $\phi_{T(P)} = \Psi_P = f$ (correttezza)
 esiste dunque un *PROG* in C_2 -*PROG* per f , per cui $f \in F(C_2)$.

8 Lezione 8

Mostreremo $F(WHILE) \subseteq F(RAM)$, mostreremo una traduzione $comp : W - PROG \rightarrow PROG$. Per comodità utilizzeremo il linguaggio RAM etichettato: etichetto una riga (istruzione) per fare un salto, ovviamente ho la stessa potenza computazionale di RAM.

In quanto $W - PROG$ è definito induttivamente $comp$ può essere definito induttivamente:

1. [BASE] mostro come compilare gli assegnamenti;
2. [PASSO] per ipotesi induttiva assumo dato $comp(C_1), \dots, comp(C_m)$ e mostro come compilare il comando composto BEGIN C_1, \dots, C_m END;
3. [PASSO] per ipotesi induttiva assumo dato $comp(C)$ e mostro come compilare il comando while WHILE $X_k \neq 0$ DO C .

8.1 Costruzione induttiva di $comp$

In generale tradurremo X_k con R_k ovvero variabili di WHILE in registri RAM ($21 \Rightarrow \infty$).

[BASE] **assegnamenti**

- $comp(X_k := 0)$

loop : *if* $R_k == 0$ *then goto* exit
 $R_k \leftarrow R_k - 1$
if $R_{21} == 0$ *then goto* loop
exit : $R_k \leftarrow R_k - 1$

Uso R_{21} perchè sarà sempre uguale a 0, WHILE ha 21 variabili (da 0 a 20).

- $comp(X_k := X_j + /-1)$
 - se $k == j$: $R_k \leftarrow R_k + /-1$
 - altrimenti:
 1. salvo X_j in R_{22} ;
 2. azzero R_k ;
 3. metto in R_j e R_k il contenuto di R_{22} (un ciclo che fa +1 al R_{22} e somma 1 a R_k e R_k);
 4. sommo/sottraggo 1 a R_k .

[PASSO]

- comando composto: basta prendere la sequenza di comando e creare la sequenza di compilazione dei comandi
- comando while: $comp(\underline{while} \ x_k \neq 0 \ \underline{do} \ C)$:

$$\begin{aligned} &\underline{loop} : \text{if } R_k == 0 \text{ then goto } \underline{exit} \\ &\quad \quad \quad comp(C) \\ &\quad \quad \quad \text{if } R_{21} == 0 \text{ then goto } \underline{loop} \\ &\underline{exit} : R_k \leftarrow R_k \dot{-} 1 \end{aligned}$$

Considerazioni

1. il compilatore è facilmente programmabile;
2. compila ogni sorgente while \Rightarrow completo;
3. mantiene la semantica: $\Psi_w = \Phi_{comp(w)} \Rightarrow$ corretto;

quindi $F(WHILE) \subseteq F(RAM)$.

8.2 $F(RAM) \subseteq F(WHILE)$

Faremo un interprete (compila riga per riga e ha come output l'output del programma), I_w : interprete scritto in WHILE di programmi scritti in RAM.

- input di I_w : $P \in PROG$ e $x \in \mathbb{N}$, output: $\Phi_P(x)$
- input di I_w : $codifica(P) = n$ e $x \in \mathbb{N}$, output: $\Phi_n(x) = \Phi_P(x)$
- input di I_w : $\langle x, n \rangle$ con x il dato di input e n la codifica del programma, output: $\Phi_n(x) = \Phi_P(x)$

La semantica di I_w : $\forall x, n \in \mathbb{N} : \Psi_{I_w}(\langle x, n \rangle) = \Phi_n(x) = \Phi_P(x)$.

Macro while Per comodità, nella scrittura di I_w utilizzeremo un WHILE che ingloba alcune macro che possono essere tradotte in WHILE puro.

9 Lezione 9

9.1 Interprete WHILE di programmi RAM

9.1.1 Variabili

I_w ricrea nelle sue variabili l'ambiente (macchina RAM) in cui esegue P . Se $\text{cod}(P) = n$ allora P non userà mai R_j con $j > n$. Posso quindi restringermi a modellare la sequenza R_0, \dots, R_{n+2} . Il contenuto della memoria in cui si esegue P può essere considerato in una sola variabile contenente $\langle a_0, a_1, \dots, a_{n+2} \rangle$. Avremo quindi la seguente configurazione:

- $X_0 \Leftarrow \langle R_0, \dots, R_{n+2} \rangle$;
- $X_1 \Leftarrow L$ con L program counter;
- $X_2 \Leftarrow x$ ovvero il dato di input;
- $X_3 \Leftarrow n$ ovvero il "listato" del programma P ;
- $X_4 \Leftarrow$ codice dell'istruzione da eseguire prelevata da X_3 nella posizione X_1 .

Inizializzazione

1. $X_1 \Leftarrow \text{input}(\langle x, n \rangle)$
2. $X_2 := \text{sin}(X_1)$ ovvero il dato di input;
3. $X_3 := \text{sin}(X_1)$ ovvero il "listato" del programma P ;
4. $X_1 := 1$.

9.1.2 Codice interprete in macrowhile

9.1.3 Conseguenze

Posso costruire $\text{Comp} : \text{PROG} \rightarrow W - \text{PROG}$:

- $n = \text{cod}(P)$
- $X_1 := \langle x, n \rangle$
- I_w

Programmabile, completo e corretto.

$$\Psi_{\text{comp}(P)}(x) = \Psi_{I_w}(\langle x, n \rangle) = \Phi_n(x) = \Phi_P(x) \text{ OK!}$$

Teorema di Bohm-Jacopini Per ogni programma con goto (RAM) ne esiste uno equivalente in linguaggio strutturato (WHILE).

```

1: while ( $X_1 \neq 0$ ) do                                      $\triangleright$  HALT con  $L = 0$ 
2:   if ( $X_1 > \text{length}(X_2)$ ) then  $X_1 := 0$                 $\triangleright$  Ho finito le istruzioni
3:   else
4:      $X_4 := \text{Pro}(X_1, X_3)$                                 $\triangleright$  Prendo elemento in pos  $X_1$  di  $X_3$  (fetch)
5:     if ( $X_4 \bmod 3 == 0$ ) then                              $\triangleright R_k \leftarrow R_k + 1$ 
6:        $X_5 := X_4 / 3$ 
7:        $X_0 := \text{incr}(X_5, X_0)$                             $\triangleright$  Increment di 1 elem in pos  $X_5$  di  $X_0$ 
8:        $X_1 := X_1 + 1$                                         $\triangleright$  Incremento program counter
9:     end if
10:    if ( $X_4 \bmod 3 == 1$ ) then                              $\triangleright R_k \leftarrow R_k - 1$ 
11:       $X_5 := (X_4 - 1) / 3$ 
12:       $X_0 := \text{decr}(X_5, X_0)$ 
13:       $X_1 := X_1 + 1$ 
14:    end if
15:    if ( $X_4 \bmod 3 == 2$ ) then                              $\triangleright$  if  $R_k == 0$  then goto  $m$ 
16:       $X_5 := \text{sin}((X_4 + 1) / 3)$                             $\triangleright k$ 
17:       $X_6 := \text{des}((X_4 + 1) / 3)$                             $\triangleright m$ 
18:      if ( $\text{Pro}(X_5, X_0) == 0$ ) then
19:         $X_1 = X_6$ 
20:      else
21:         $X_1 := X_1 + 1$ 
22:      end if
23:    end if
24:  end if
25: end while
26:  $X_0 := \text{sin}(X_0)$ 

```

Finalmente.. $\mathbb{N}_1^{\mathbb{N}} \sim \mathbb{N} \sim \text{PROG} \sim F(\text{RAM}) = F(\text{WHILE})$

- nei sistemi di programmazione RAM e WHILE esistono funzioni non computabili, dimostrato formalmente;
- i sistemi RAM e WHILE, pur profondamente diversi, calcolano le stesse cose.

Usiamo $\text{comp} : W - \text{PROG} \rightarrow \text{PROG}$ sse $I_w(\Psi_{I_w}(< x, n >) = \Phi_n(x))$.

$$U = \text{comp}(I_w) \in \text{PROG}$$

$$\Phi_U(< x, n >) = \Psi_{I_w}(< x, n >) = \Phi_n(x)$$

Nel sistema di programmazione RAM esiste un programma RAM capace di simulare qualunque altro programma RAM! U è detto interprete universale (RAM).

9.1.4 Riflessioni concetto di calcolabilità

Due sistemi profondamente diversi che determinano la stessa idea di calcolabilità. Possiamo sublimare il concetto di calcolabile?

Possiamo pensare di definire ciò che è calcolabile a prescindere dalle macchine che usiamo per calcolare?

Possiamo pensare di definire ciò che è calcolabile in termini più astratti, matematici, "lontani dall'informatica"?

10 Lezione 10

10.1 Chiusura di insiemi rispetto ad operazioni

Dato un insieme U , si definisce un'operazione su U una qualunque funzione $op : U \times \dots \times U \rightarrow U$ con U ripetuta k volte in input: k -arietà dell'operazione.

Esempi $U = \mathbb{N}$

- $+: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, somma: $+(5, 3) = 8$, operazione binaria;
- $\lfloor \sqrt{\cdot} : \mathbb{N} \rightarrow \mathbb{N}$, radice troncata, operazione unaria;
- $Pro_t^n : \mathbb{N} \times \dots \times \mathbb{N} \rightarrow \mathbb{N}$ con n volte in input, proiezione t -esima, operazione n -aria.

10.1.1 Chiusura

L'insieme $a \subseteq U$ è chiuso rispetto all'operazione $op : U^k \rightarrow U$ sse $\forall a_1, \dots, a_k \in A : op(a_1, \dots, a_k) \in A$, ovvero l'operazione non mi "amplia" A .

Esempi

- $+: \mathbb{N}^2 \rightarrow \mathbb{N}, PARI \subseteq \mathbb{N}$. $PARI$ è chiuso per $+$? sì, $2k + 2j = 2(k + j)$
- $+: \mathbb{N}^2 \rightarrow \mathbb{N}, DISPARI \subseteq \mathbb{N}$. $DISPARI$ è chiuso per $+$? no, $23 + 3 = 6$
- $/: \mathbb{Q}^2 \rightarrow \mathbb{Q}, \mathbb{N} \subseteq \mathbb{Q}$. \mathbb{N} è chiuso per $/$? no, $5/2 \notin \mathbb{N}$

Per rispondere si dobbiamo dimostrare che vale, nel nostro caso, per ogni coppia; per rispondere no basta un controesempio.

In generale: se Ω è un'insieme di operazioni su U , allora $A \subseteq U$ è chiuso rispetto a Ω se è chiuso per ogni operazione in Ω .

Esempi

- $\Omega = \{+, -\}, PARI \subseteq \mathbb{N} \rightarrow$ somma sì, prodotto sì ($2k \cdot 2j = 4kj$);
- $\Omega = \{+, -\}, DISPARI \subseteq \mathbb{N} \rightarrow$ somma no, prodotto sì ($(2k+1)(2j+1) = 4kj + 2k + 2j + 1$ è dispari).

10.1.2 Chiusura di un'insieme

Problema: Sia $A \subseteq U$ e $op : U^k \rightarrow U$. Qual è il più piccolo sottoinsieme di U che contiene A e sia chiuso per op ? In pratica voglio allargare A per renderlo chiuso.

Risposte ovvie:

- se A è chiuso per op allora A stesso;
- sicuramente U soddisfa le due richieste ma non è sempre il più piccolo.

Esempio Sia $A = \{2, 3\} \subseteq \mathbb{N}$ e $+: \mathbb{N}^2 \rightarrow \mathbb{N}$, qual è il più piccolo insieme che soddisfa le proprietà sopra? Sicuramente \mathbb{N} ma non è il più piccolo: sicuramente non ci servono 0 e 1. Non basta nemmeno aggiungere $\{4, 5, 6\}$ perchè devo soddisfare la chiusura anche sui valori che aggiungo all'insieme che sto cercando.

Teorema: Sia $A \subseteq U$ e $op : U^k \rightarrow U$. Il più piccolo sottoinsieme di U contenente A è chiuso rispetto a op si ottiene calcolando la chiusura rispetto a op , cioè l'insieme A^{op} definito induttivamente come:

1. $\forall a \in A \Rightarrow a \in A^{op}$;
2. $\forall a_1, \dots, a_k \in A^{op} \Rightarrow op(a_1, \dots, a_k) \in A^{op}$;
3. nient'altro sta in A^{op} .

Possiamo dare la seguente definizione più operativa:

1. metti in A^{op} tutti gli elementi di A ;

2. applica op a una k -tupla di elementi in A^{op} ;
3. aggiungo il risultato in A^{op} se non è già presente;
4. reitro i punti 2 e 3 finché A^{op} cresce;
5. output A^{op} .

Esempio $A = \{2, 3\}, A \subseteq \mathbb{N}$ e voglio trovare A^+ :

1. $A^+ \leftarrow A$;
2. $2 + 3 = 5 \notin A^+ \Rightarrow A^+ = \{2, 3, 5\}$;
3. $2 + 2, 3 + 3, 5 + 5, \dots \notin A^+ \Rightarrow A^+ = \{2, 3, 4, 5, 6, 10, \dots\}$;
4.
5. output: $A^+ = \mathbb{N} \setminus \{0, 1\}$.

10.1.3 Chiusura di un insieme rispetto a un insieme di operazioni

A^Ω si ottiene generalizzando il processo per un'operazione induttivamente:

1. $\forall a \in A \Rightarrow a \in A^\Omega$;
2. $\forall i \in \{1, \dots, t\} \forall a_1, \dots, a_k \in A^\Omega \Rightarrow op_i(a_1, \dots, a_k) \in A^\Omega$;
3. nient'altro sta in A^Ω .

10.1.4 Verso una definizione teorica di calcolabilità

Astratta : che astraeda qualunque connotato informatico;

Roadmap :

1. $ELEM$: insieme di funzioni che qualunque idea di calcolabile si voglia proporre deve considerare calcolabili; ELE non può esaurire il concetto di calcolabilità \Rightarrow ampliare;
2. Ω : insieme di operazioni su funzioni che costruiscono funzioni. Le op in Ω sono banalmente implementabili \Rightarrow se le applico a funzioni calcolabili ottengo nuove funzioni calcolabili;
3. $ELEM^\Omega = P$ = classe delle funzioni ricorsive parziali. $P \rightarrow$ la nostra idea astratta della classe delle funzioni calcolabili.

11 Lezione 11

11.1 *ELEM*: nucleo delle funzioni calcolabili

ELEM =

$$\{\text{successore} : S(x) = x + 1, x \in \mathbb{N} \\ \text{zero} : O^n(x_1, \dots, x_n) = 0, x \in \mathbb{N}, // \text{prende } n \text{ input e restituisce } 0 \\ \text{proiettori} : Pro_k^n(x_1, \dots, x_n) = x_k, x \in \mathbb{N}\}$$

Queste funzioni sono facilmente implementabili e sicuramente calcolabili.

Ovviamente *ELEM* non può essere considerato come l'idea teorica di tutto ciò che è calcolabile, per esempio $f(x) = x + 2 \notin ELEM$ ma f deve essere considerata calcolabile!

\Rightarrow *ELEM* deve essere ampliata mantenendo l'idea di induttivamente calcolabile.

11.2 Operatore di composizione di funzioni

Sia $h : \mathbb{N}^k \rightarrow \mathbb{N}$ e $g_1, \dots, g_k : \mathbb{N}^k \rightarrow \mathbb{N}$, denoteremo $\underline{x} \in \mathbb{N}^k$.

$$COMP(h, g_1, \dots, g_k) : \mathbb{N}^k \rightarrow \mathbb{N} \text{ definito come} \\ COMP(h, g_1, \dots, g_k)(\underline{x}) = h(g_1(\underline{x}), \dots, g_k(\underline{x}))$$

Intuitivamente *COMP* è implementabile, per cui la composizione di "cose" programmabili rimane programmabile.

11.2.1 Ampliamo *ELEM* chiudendo rispetto a *COMP*

$ELEM^{COMP}$ effettivamente amplia *ELEM* infatti

$$f(x) = x + 2 \notin ELEM \text{ ma } f(x) \in ELEM^{COMP} \\ \text{infatti } f(x) = COMP(SUCC, SUCC)(x) = SUCC(SUCC(x)) = f(x)$$

$ELEM^{COMP}$ può incarnare teoricamente l'idea di classe delle funzioni calcolabili? **NO!**

11.3 Operatore di ricorsione primitiva

Esempio Fattoriale:

$$FATT(n) = \begin{cases} 1 & \text{se } n = 0 \\ n(FATT(n-1)) & \text{se } n > 0 \end{cases}$$

Siano $g : \mathbb{N}^n \rightarrow \mathbb{N}$ e $h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$; $g(\underline{x})$ e $h(z, y, \underline{x})$ con $x \in \mathbb{N}^n$

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

11.3.1 Ampliamo $ELEM^{COMP}$ chiudendo rispetto a RP

$ELEM^{COMP,RP} = \underline{RICPRIM}$ = funzioni ricorsive primitive

$$SOMMA(x, y) = \begin{cases} x = Pro_1^2 & \text{se } y = 0 \text{ //non ho costante } x \text{ in } ELEM \\ SUCC(SOMMA(x, y - 1)) & \text{se } y > 0 \end{cases}$$

$$PROD(x, y) = \begin{cases} 0 = O^2(x, y) & \text{se } y = 0 \\ SOMMA(x, PROD(x, y - 1)) & \text{se } y > 0 \end{cases}$$

$$PRED(x) = \begin{cases} 0 & \text{se } x = 0 \\ x - 1 & \text{se } x > 0 \end{cases}$$

con il predecessore posso definire la differenza come

$$DIFF(x, y) = \begin{cases} x & \text{se } y = 0 \\ DIFF(PRED(x), y - 1) & \text{se } y > 0 \end{cases}$$

11.4 $RICPRIM$ vs $WHILE$

$RICPRIM$ contiene già molte funzioni e potrei già chiedermi se ho raggiunto $F(WHILE)$. Mostriamo $RICPRIM \subseteq F(WHILE)$ per induzione strutturale. ($F(WHILE)$ ha possibilità di indefinito mentre $RICPRIM$ no).

11.4.1 Road map

1. $[BASE]$ le funzioni in $ELEM$ sono in $RICPRIM$;
2. $[PASSO INDUTTIVO]$ se $h, g_1, \dots, g_k \in RICPRIM \Rightarrow COMP(h, g_1, \dots, g_k) \in RICPRIM$;
3. $[PASSO INDUTTIVO]$ se $g, k \in RICPRIM \Rightarrow RP(g, h) \in RICPRIM$;
4. null'altro è in $RICPRIM$;

quindi:

1. dimostro che $ELEM \subseteq F(WHILE)$;
2. assumo per ipotesi induttiva che $h, g_1, \dots, g_k \in RICPRIM$ siano in $F(WHILE)$ e dimostro che $COMP(h, g_1, \dots, g_k) \in F(WHILE)$;
3. assumo per ipotesi induttiva che $g, h \in RICPRIM$ siano in $F(WHILE)$ e dimostro che $RP(g, h) \in F(WHILE)$

$\Rightarrow RICPRIM \subseteq F(WHILE)$.

11.4.2 Dimostrazione induttiva

- $[BASE]$: $ELEM \subseteq F(WHILE)$, ovvio (le 3 funzioni iniziali);
- $[PASSI INDUTTIVI]$:
 - $COMP$: assumiamo per ipotesi induttiva $h, g_1, \dots, g_k \in RICPRIM$ siano in $F(WHILE) \Rightarrow$ esistono $H, G_1, \dots, G_k \in W - PROG$ t.c. $\Psi_H = h, \Psi_{G_1} = g_1, \dots$. Mostro quindi un programma $WHILE$ che calcola $COMP(h, g_1, \dots, g_k) = h(g_1(\underline{x}), \dots, g_k(\underline{x}))$.

Algorithm 1 $w \equiv$

```

1: BEGIN  $\triangleright x_1 \leftarrow \underline{x}$  come  $< a_1, \dots, a_n >$ 
2:  $x_0 := G_1(x_1)$ 
3:  $x_0 := [x_0, G_2(x_1)]$   $\triangleright$  calcolo i vari  $G$  e poi metto insieme in  $H$ 
4:  $\dots$ 
5:  $x_0 := [x_0, G_k(x_1)]$   $\triangleright x_0 := [G_1(\underline{x}), \dots, G_k(\underline{x})]$ 
6:  $x_0 := H(x_0)$   $\triangleright x_0 := H(G_1(\underline{x}), \dots, G_k(\underline{x}))$ 
7: END

```

quindi $Psi_w(\underline{x}) = COMP(h, g_1, \dots, g_k)(\underline{x})$

- RP : assumo per ipotesi induttiva $g(\underline{x}), h(z, y, \underline{x}) \in RICPRIM$ in $F(WHILE) \Rightarrow$ esistono $G, H \in W - PROG$ con $\Psi_G = g$ e $\Psi_H = h$. Mostro quindi un programma $WHILE$ che calcola

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

Notiamo che $f(\underline{x}, 2) = h(h(g(\underline{x}), 0), 1, \underline{x})$

Algorithm 2 $w \equiv$

```

1: BEGIN  $\triangleright x_1 \leftarrow < \underline{x}, y >$ 
2:  $t := G(\underline{x})$ 
3:  $k := 1$ 
4: while  $k \leq y$  do
5:    $t := H(t, k - 1, \underline{x})$ 
6:    $k := k + 1$ 
7: end while
8: END

```

quindi $\Psi_w(< \underline{x}, y >) = RP(g, h)(\underline{x}, y)$

Quindi abbiamo dimostrato per induzione strutturale che $RICPRIM \subseteq F(WHILE)$.

11.4.3 Considerazioni su *RICPRIM*

- vorremo che la nostra "idea teorica" (*RICPRIM*) di calcolabilità raggiungesse almeno quella pratica ($F(WHILE)$), quindi la normale domanda è: l'inclusione è propria? (ovvero inclusione stretta);
- è facile dimostrare per induzione strutturale che tutte le funzioni in *RICPRIM* sono totali (ovvero che terminano sempre);
- $F(WHILE)$ contiene anche funzioni parziali (programmi infiniti) $\Rightarrow RICPRIM \subsetneq F(WHILE)$;
- posso tradurre il *WHILE* di prima in un *FOR*, $F(FOR) = RICPRIM$;
- poniamo di usare *WHILE* – *LOOP* che non vanno all' ∞ , consideriamo cioè l'insieme $\tilde{F}(WHILE) = \{\Psi_w : w \in w - PROG \text{ e } \Psi_w \text{ è totale}\}$;
- l'inclusione $RICPRIM = F(FOR) \subseteq \tilde{F}(WHILE)$ è propria? Per diagonalizzazione $RICPRIM \subsetneq \tilde{F}(WHILE)$. In particolare per la funzione di Ackermann:

$$A(m, n) = \begin{cases} n + 1 & \text{se } m = 0 \\ A(m - 1, 1) & \text{se } m > 0, n = 0 \\ A(m - 1, f(m, n - 1)) & \text{se } m > 0, n > 0 \end{cases}$$

cresce troppo in fretta per essere in *RICPRIM* \Rightarrow Ackermann $\in \tilde{F}(WHILE)$ ma $\notin RICPRIM$ perchè for non può cambiare il "TO y ", ovvero non può prevedere le iterazioni.

12 Lezione 12

12.1 Operatore di minimalizzazione di funzioni

Sia $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$; $f(\underline{x}, y)$ con $\underline{x} \in \mathbb{N}$. Definiamo

$$MIN(f(\underline{x}, y)) = g(\underline{x}) = \begin{cases} y & \text{se } f(\underline{x}, y) = 0 \text{ e } \forall t < y : f(\underline{x}, t) \neq 0 \\ \perp & \text{altrimenti} \end{cases}$$

con $\forall t < y : f(\underline{x}, t) \neq 0$ indichiamo che y è il più piccolo valore t.c. $f(\underline{x}, y) = 0$. Possiamo scrivere l'operatore di minimalizzazione come

$$\mu y (f(\underline{x}, y) = 0) \text{ e } \perp \text{ se } \nexists \text{ tale } y$$

dove con μy indichiamo il più piccolo valore di y tale che $f(\underline{x}, y) = 0$.

Esempi $(f(x, y) \Rightarrow MIN(f(\underline{x}, y)) = g(\underline{x}))$

- $x + y + 1 \Rightarrow \perp$: sono in \mathbb{N} e quindi non ho i numeri negativi, il più piccolo y è 0 quindi non ho minimalizzazione;
- $x \dot{-} y \Rightarrow x$;
- $y \dot{-} x \Rightarrow 0$;
- $x \dot{-} y^2 \Rightarrow \lceil \sqrt{x} \rceil$.

12.2 La classe P delle funzioni ricorsive parziali

Ampio $RICPRIM$ chiudendo rispetto all'operatore MIN , ovvero $RICPRIM^{MIN} = ELEM^{COMP, RP, MIN} = P = \{\text{funzioni ricorsive parziali}\}$.

Sicuramente P , che grazie a MIN contiene anche funzioni parziali, amplia $RICPRIM$ fatto solo da funzioni totali. Ma come si pone rispetto a $F(WHILE)$?

12.2.1 Teorema $P \subseteq F(WHILE)$

Dimostrazione: $P \equiv RICPRIM^{MIN}$ può essere definito induttivamente come:

1. le funzioni in $RICPRIM$ sono in P ;
2. se f appartiene a P allora $MIN(f) \in P$
3. null'altro è in P .

Quindi, per induzione strutturale, dimostriamo il punto 2 con i seguenti passi:

1. le funzioni $RICPRIM$ sono $WHILE$ -programmabili (FATTO!);
2. sia $f \in P$ $WHILE$ -programmabile per ipotesi induttiva, allora esibisci un programma $WHILE$ che calcoli $MIN(f)$.

Sia $f(\underline{x}, y \in P)$; per ipotesi induttiva, sia f calcolata dal programma $WHILE$ F . Scriviamo un programma $WHILE$ per

$$MIN(f(\underline{x}, y)) = g(\underline{x}) = \begin{cases} \mu y (f(\underline{x}, y) = 0) \\ \perp \end{cases} \quad \text{se } \nexists \text{ tale } y$$

NOTA: la dicitura $MIN(f(\underline{x}, y)) = g(\underline{x})$ indica che la parte sinistra produce la funzione $g(x)$.

Per induzione strutturare $P \subseteq F(WHILE)$.

```

1: INPUT( $\underline{x}$ )
2: BEGIN
3:  $y := 0$  ▷ output di  $MIN(f)$ 
4: while  $F(\underline{x}, y) \neq 0$  do ▷ posso scrivere  $F(\underline{x}, y)$  per ip.ind.:  $F$  è in WHILE
5:    $y := y + 1$ 
6: end while
7: END

```

12.2.2 Teorema $F(WHILE) \subseteq P$

Dimostrazione $F(WHILE) = \{\Psi_w : w \in W-PROG\} \subseteq P = ELEM^{COMP, RP, MIN}$.

$\Psi_w \in F(WHILE) \Rightarrow \Psi_w \in P = ELEM^{COMP, RP, MIN} \Rightarrow$ dimostro che Ψ_w può essere espressa come composizione, ricorsione primitiva e minimalizzazione a partire da funzioni in *ELEM*.

$Psi_w(x) = Pro_0^{21}([w])(w - in(x))$ con

- $[[\cdot]](\cdot) : \mathbb{N}^{21} \rightarrow \mathbb{N}^{21}$ dove $[[c]](\underline{x}) = \underline{y}$ è la funzione stato prossimo che calcola lo stato $y \in \mathbb{N}^{21}$ a seguito del comando *WHILE* c a partire dallo stato $\underline{x} \in \mathbb{N}^{21}$;
- $w - in(x)$ prepara lo stato iniziale su input x ;
- $w \in W - PROG$ è un comando composto.

Possiamo quindi dire che $Psi_w(x) = Pro_0^{21}([w])(w - in(x))$ è composizione della funzione $Pro_0^{21} \in ELEM$ con la funzione stato prossimo $[[w]](w - in(x))$.

Quindi

1. $Pro_0^{21} \in ELEM \Rightarrow Pro_0^{21} \in P$;
2. P è chiuso rispetto alla composizione;
3. se dimostro che $[[c]](\underline{x}) = y \in P$ allora $\Psi_w \in P$.

Dettaglio tecnico: $[[\cdot]](\cdot) : \mathbb{N}^{21} \rightarrow \mathbb{N}^{21}$ ha come codominio \mathbb{N}^{21} mentre P , per come l'abbiamo definita, contiene funzioni che hanno codominio in \mathbb{N} ; questo non è un problema perchè usiamo le coppie di Cantor. Quindi invece di mostrare $[[c]](\underline{x}) = y \in P$ con $\underline{x}, y \in \mathbb{N}^{21}$ mostreremo che $f_c(x) = y$ con $x = [\underline{x}]$ e $y = [\underline{y}]$ CANTOR.

$$f_c(x) = y \Leftrightarrow [[c]](Pro(0, x), \dots, Pro(20, x)) = (Pro(0, y), \dots, Pro(20, y))$$

Posso andare da f_c a $[[c]]$ e viceversa con operatori in P : $f_c \in P \Leftrightarrow [[c]] \in P$.

Equivalentemente posso pensare che $[[c]] \in P$ perchè ogni componente dello stato prossimo $[[c]](\underline{x})$ è esprimibile come funzione ricorsiva parziale.

12.2.3 $F(WHILE) \subseteq P$ dimostrando che $f_c \in P$

Induzione strutturale sul comando *WHILE* c :

- *[BASE]*:
 - $c \equiv x_k := 0$: $f_{x_k=0}(x) = [Pro(0, x), \dots, 0, \dots, Pro(20, x)]$ con 0 nella k -esima posizione, Pro e 0 (funzione $O \in ELEM$) sono tutte funzioni in P , così come la loro composizione $\Rightarrow f_{x_k=0}(x) \in P$;
 - $c \equiv x_k := x_j + / - 1$: $f_{x_k:=x_j+/-1}(x) = [Pro(0, x), \dots, Pro(j, x) + / - 1, \dots, Pro(20, x)]$ nella posizione k -esima, ancora tutto in $P \Rightarrow f_{x_k:=x_j+/-1}(x) \in P$;
- *[PASSI INDUTTIVI]*:
 - $c \equiv \underline{BEGIN} \ c_1, \dots, c_m \ \underline{END}$, per ipotesi induttiva $f_{c_i} \in P$. $f_c(x) = f_{c_m}(\dots, f_{c_1}(x), \dots)$, ogni $f_{c_i} \in P$ come la loro composizione.
 - $c' \equiv \underline{WHILE} \ x_k \neq 0 \ \underline{do} \ c$, ipotesi induttiva $f_c \in P$.

$$f_{c'}(x) = f_c^{e(x)}(x) \text{ con } e(x) = \mu y (Pro(k, f_c^y(x)) = 0)$$

f^n : composizione di funzione n volte. ATTENZIONE: $f_c^{e(x)}$ è la composizione di f_c per $e(x)$ volte, che NON è un numero COSTANTE. So scrivere $COMP(h, g_1, \dots, g_k)$ con k costante. Come rappresentare in P la composizione di una funzione con se stessa un numero NON COSTANTE di volte? come rappresentare $T(x, y) = f_c^y(x)$ in P ?

$$T(x, y) = \begin{cases} x & \text{se } y = 0 \\ f_c(T(x, y-1)) & \text{se } y > 0 \end{cases}$$

Quindi $T(x, y) = f_c^y(x)$ è un operatore ottenuto mediante ricorsione primitiva su una funzione $f_c \in P \Rightarrow T(x, y) \in P$.

Dunque $f_{c'} = f_c^{e(x)}(x)$ con $e(x) = \mu y (Pro(x, f_c^y(x)) = 0) \Rightarrow$ (quindi posso scrivere) $e(x) = \mu y (Pro(x, T(x, y)) = 0)$ ovvero minimizzazione di funzione $T(x, y) \in P$ quindi $e(x) \in P$. Infine $f_{c'}(x) = f_c^{e(x)}(x) = T(x, e(x))$ composizione di funzioni in P quindi $f_{c'} \in P$.

Abbiamo dimostrato, per induzione strutturale, che $F(WHILE) \subseteq P$.

12.3 Tesi di Church-Turing

La classe delle funzioni intuitivamente calcolabili coincide con la classe P delle funzioni ricorsive parziali. Con "intuitivamente calcolabile" intendiamo definito calcolabile da un modello di calcolo "ragionevole" (come i modelli forniti negli anni).

La tesi di Church-Turing è una congettura. Non può essere un teorema in quanto non è possibile caratterizzare i modelli di calcolo ragionevoli che sono stati e verranno proposti in maniera completa.

13 Lezione 13

13.1 Proprietà auspicabili per un sistema di programmazione

Individueremo un sistema di programmazione con $\{\Phi_i\}_{i \in \mathbb{N}}$ = insieme delle funzioni calcolabili con quel sistema, $i \in \mathbb{N}$ = programmi (codifica) di quel sistema.

Per individuare queste proprietà auspicabili, prendiamo spunto dal sistema *RAM*:

1. Potenza computazionale: $\{\Phi_i\} \in P$, coerente con la tesi di Church-Turing. Proprietà che vorrei soddisfatta dai sistemi di programmazione che investigheremo;
2. Interprete universale: esiste un programma $u \in \mathbb{N}$ tale che $\forall x, n \in \mathbb{N} : \Phi_u(< x, n >) = \Phi_n(x)$. Indichiamo con u l'interprete universale (scritto in *RAM*, interpreta ogni programma scritto in *RAM*)
 - (a) $COMP : w-PROG \rightarrow PROG$ tale che $\forall w \in w-PROG \Phi_{COMP(w)} = \Psi_w$
 - (b) T_w : interprete scritto in *WHILE* di programmi *RAM* tale che $\forall x, n \in \mathbb{N} : \Psi_{I_w}(< x, n >) = \Phi_n(x)$

$COMP(I_w) = U \in PROG$. Per (a) vale $\Phi_U = \Psi_{I_w}$. Quindi, per (b), $\forall x, n \in \mathbb{N} : \Phi_U(< x, n >) = \Psi_{I_w}(< x, n >) = \Phi_n(x)$ quindi $u = cod(U)$ (cod =codifica). La presenza di un interprete universale nei sistemi di programmazione permette un' "algebra dei programmi" (trasformazione programmi).

3. Teorema S_1^1 : è possibile costruire automaticamente programmi più specifici da programmi più generali, ottenuti prefissando alcuni input. Se prendiamo per esempio il programma che somma due numero posso facilmente scrivere un programa che svolte $x + 3$: imposto a 3 la y e poi chiamo il programma generale $(x + y)$. In generale il programma S_1^1 implementa la funzione

$$S_1^1(n, y) = \bar{n} \text{ tale che } \Phi_{\bar{n}}(x) = \Phi_n(x, y)$$

con n codifica dei P e due variabili di input x, y . Con \bar{n} codifica del programma \bar{P} a una variabile x la cui semantica è identica a quella di P in cui fisso a y il secondo input.

Algorithm 3 $\bar{n} = cod(\bar{P})$ con $\bar{P} \equiv$

- | | |
|----------------------------------|--------------------------|
| 1: $R_0 \leftarrow R_0 + 1$ | |
| 2: ... | |
| 3: $R_0 \leftarrow R_0 + 1$ | ▷ y volte |
| 4: $R_1 \leftarrow < R_1, R_0 >$ | ▷ Creo input per P (*) |
| 5: $R_0 \leftarrow 0$ | ▷ Azzerò R_0 (**) |
| 6: P | ▷ Richiamo P su R_1 |
-

Posso esplicitare la funzione S_1^1 come

$$S_1^1(n, y) = \bar{n} = \langle 0, \dots, 0, s, t, n \rangle$$

dove gli 0 y volte rappresentano l'aritmetizzazione di $R_0 \leftarrow R_0 + 1$, s rappresenta la codifica di $(*)$ e t la codifica di $(**)$.

Quali caratteristiche ha $S_1^1(n, y)$?

1. è una funzione totale;
2. è programmabile

quindi $S_1^1 \in T$ dove T è l'insieme delle funzioni ricorsive totali. In sintesi, per RAM , esiste una funzione S_1^1 ricorsiva totale che accetta come argomenti:

1. il codice n di un programma che ha due input;
2. un valore y cui fissare il secondo input

e produce il codice $S_1^1(n, y)$ di un programma che si comporta come n a cui il secondo input è fissato ad y .

Teorema S_1^1 Dato $\{\Phi_i\}$ RAM , esiste una funzione $S_1^1 \in T$ tale che $\forall n, x, y \in \mathbb{N} : \Phi_n(\langle x, y \rangle) = \Phi_{S_1^1(n, y)}(x)$

Teorema S_n^m in generale, il teorema S_n^m parte da programmi con $m+n$ input, in cui si dissano gli ultimi n input.

13.2 Sistemi di programmazione accettabili (spa)

Le tre caratteristiche auspicabili, evidenziate per il sistema RAM , rappresenta i tre assiomi che caratterizzano i sistemi di programmazione su cui ci concentreremo:

Un sistemi di programmazione $\{\Phi_i\}$ si dice accettabile (spa) sse soddisfa: (ssiom di Rogers):

1. $\{\Phi_i\} = P$ (aderisce alla tesi di Church-Turing);
2. $\exists u \in \mathbb{N} : \Phi_u(\langle x, n \rangle) = \Phi_n(x)$ (esiste interprete universale);
3. $\exists S_n^m \in T : \forall k \in \mathbb{N}, \underline{x} \in \mathbb{N}^m, \underline{y} \in \mathbb{N}^n$ vale $\Phi_k(\langle \underline{x}, \underline{y} \rangle) = \Phi_{S_n^m(k, \underline{y})}(\langle \underline{x} \rangle)$

Non sono assiomi assolutamente restrittivi: tutti i modelli di calcolo "ragionevoli" sono di fatto spa.

13.2.1 Esistenza di compilatori tra spa

DEF. Dati gli spa $\{\Phi_i\}$ e $\{\Psi_i\}$, un compilatore del primo al secondo spa è una funzione $t : \mathbb{N} \rightarrow \mathbb{N}$ tale che:

1. Correttezza: $\forall i \in \mathbb{N}$ vale $\Phi_i = \Psi_{t(i)}$;
2. Completezza: compila ogni $i \in \mathbb{N}$;
3. Programmabile: esiste un programma che implementa t .

I punti 2 e 3 ci dicono che $t \in T$ (ricorsiva totale).

In sintesi, un compilatore tra due spa è una funzione dai naturali ai naturali ricorsiva totale e corretta (mantiene la semantica).

Teorema Dati due spa,, esiste sempre un compilatore tra essi.

Dimostrazione Assiomi spa:

1. $\{\Phi_i\} = P$;
2. $\exists u \in \mathbb{N} : \Phi_u(< x, n >) = \Phi_n(x)$;
3. $\exists S_n^m \in T : \Phi_k(< \underline{x}, \underline{y} >) = \Phi_{S_n^m(k, \underline{y})}(< \underline{x} >)$.

Devo esibire $t \in T$ corretta (per definizione di compilatore):

$$\Phi_i(x) \stackrel{=2)}{=} \Phi_u(< x, i >) \stackrel{=1)}{=} \Psi_e(< x, i >) \stackrel{=3)}{=} \Psi_{S_1^1(e, i)}(x) \quad (*)$$

Il compilatore cercato è la funzione $t(i) = S_1^1(e, i)$, per ogni $i \in \mathbb{N}$. e è fissato: è quella e tale che $\Phi_u = \Psi_e$. Infatti:

1. $t \in T$ in quanto S_1^1 (3))
2. t è corretta: $\Phi_i = \Psi_{t(i)}$ per (*).

Corollario Dati gli spa A, B, C , esiste sempre un compilatore da A a B scritto nel linguaggio C .

Dimostrazione Per il teorema precedente, esiste il compilatore $t \in T$ da A a B . Poichè C è un spa, per l'assioma 1), C contiene programmi per tutte le funzioni ricorsive parziali. Dunque, contiene un programma anche per t che è funzione ricorsiva totale.

In pratica, ciò vuol dire che per qualunque coppia di linguaggi esistenti e che verranno progettati in futuro, sarò sempre in grado di scrivere un compilatore tra essi nel mio linguaggio preferito.

13.2.2 Un risultato più forte

Teorema di isomorfismo tra spa (Rogers) Dati due spa $\{\Phi_i\}$ e $\{\Psi_i\}$, esiste $t : \mathbb{N} \rightarrow \mathbb{N}$ tale che:

1. $t \in T$
2. $\forall i \in \mathbb{N} : \Phi_i = \Psi_{t(i)}$, con il punto sopra: esistenza del compilatore da $\{\Phi_i\}$ e $\{\Psi_i\}$, teorema precedente;
3. t è invertibile: t^{-1} è un DECOMPILATORE (tornare esattamente al listato del primo, stessa sintassi).

14 Lezione 14