INFORMATICA TEORICA

giosumarin

March 2021

Contents

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: $f : A \to B$: chiamiamo A il dominio della funzione e B il codominio.
Definizione locale: $a \to^f b$ oppure $f(a) = b$ con b immagine di a rispetto a f e a controimmagine di b rispetto a f.
$f : \mathbb{N} \to \mathbb{N}$ con $\mathbb{N} = \{0, 1, 2, 3, 4, ..\}$ e con $\mathbb{N}^+ = \{1, 2, 3, 4, ..\}$
Globale: $f(n) = \lfloor\sqrt{n}\rfloor$; Locale: $f(5) = \lfloor\sqrt{n}\rfloor$ 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

$$f : A \to B \textbf{ è iniettiva sse } \forall a_1, a_2 \in A \Rightarrow f(a_1) \neq f(a_2)$$

ovvero non ci sono confluenze verso un punto del codominio.

Funzione Suriettiva

$$f : A \to B \textbf{ è suriettiva sse } \forall b \in B, \exists a \in A : f(a) = b$$

Definiamo con Im_f l'insieme delle immagini. Quindi

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

Possiamo quindi dire che in generale $Im_f \subseteq B$ ed è suriettiva sse $Im_f = B$, ovvero quando il grafico della funzione compre 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

$$f : A \to B$$
$$g : B \to C$$
$$f \textbf{ composto } g\textbf{: } g \cdot f : A \to C$$
$$\textbf{definita come } g \cdot f(a) = g(f(a))$$

Funzione Inversa

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

Definiamo

$$i_A : A \to A \textbf{ con } i_A(a) = a$$

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

$$f(a)\downarrow\text{: } f \textbf{ definita } \forall a \in A \textbf{ si dice che } f \textbf{ è totale}$$
$$f(a)\uparrow\text{: } \textbf{non definita per ogni } a \in A.$$

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

$$Dom_f = \{a \in A : f(a)\downarrow\} \subseteq A$$
$$Dom_f \subsetneq A \Rightarrow f \textbf{ parziale (incluso stretto)}$$
$$Dom_f = Af \textbf{ totale}$$

Totalizzare

$$f : A \to B \textbf{ parziale} \Rightarrow \tilde{f} : A \to B \cup \{\perp\} \textbf{ totale,}$$
$$\textbf{Indichiamo } B \cup \{\perp\} \to B_\perp$$
$$\tilde{f} = \begin{cases} f(a) \text{ se } a \in Dom_f \\ \perp \textbf{ altrimenti} \end{cases}$$

Prodotto Cartesiano

$$A \times b = \{(a, b) : a \in A \wedge b \in B\}$$
$$\textbf{Nota: } \times \textbf{ non commutativa } A \times B \neq B \times A$$
$$\textbf{Proiettore -iesimo} \pi_i : A_1 \times \cdots \times A_n \to A_i$$
$$\pi_i(a_1, \ldots, a_n) = a_i$$
$$\textbf{Indichiamo A per n volte come } A \times \cdots \times A = A^n$$

Insieme di funzioni

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

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

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

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

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

$$C : DATI_\perp^{DATI} \times DATI \to DATI_\perp$$
$$\textbf{dove } C(P, x) \textbf{ è la funzione calcolata da } P$$

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

Potenza computazionale di C

$$F(C) = \{C(P, \_) : P \in PROG\} \subseteq DATI_\perp^{DATI}$$
$$F(C) = DATI_\perp^{DATI} \Rightarrow \textbf{ informatica può tutto}$$
$$F(C) \subsetneq DATI_\perp^{DATI} \Rightarrow \textbf{ esistono compiti non automatizzabili}$$

Cardinalità Indichiamo con $|A|$ il numero di elementi di A. Ha senso però solo su insiemi infiniti. Infatti $|\mathbb{N}| = \inf = |\mathbb{R}|$ 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, $aRb \wedge bRc \to aRc$

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

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

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

- Non esistono classi di equivalenza vuote (per riflessività ho almeno dentro me stesso);
- dati $a, b \in A \Rightarrow [a]_R \cap [b]_R = \varnothing$ **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 biezione tra A e B (associazione 1 a 1 tra elementi di A e B).
Propietà 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 sottoinsime di $\mathbb{N}$ che non c'è nell'elenco.

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

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

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

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 sottoinsimeme di $P(\mathbb{N}$ che non appartiene a $\mathbb{N}$.

$\mathbb{N}^{\mathbb{N}}$ Insieme non numerabile $\mathbb{N}^{\mathbb{N}} = \{f : \mathbb{N} \to \mathbb{N}\}$.
Anche in questo caso procedo per diagonalizzazione per ipotesi assurda. Metto sulle colonne i valori di N e sulle righe le funzioni.

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

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

3.2 Cosa è calcolabile?

Considerazioni ragionevoli:

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

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

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. $\mathbb{N} \times \mathbb{N} \underline{\sim} \mathbb{N}^+ \Rightarrow \mathbb{N} \times \mathbb{N} \sim \mathbb{N}$, da cui si può ottenere $\mathbb{Q} \sim \mathbb{N}$ consierando 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 $<,>: \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+$ iniettiva e suriettiva. Abbiamo $< x, y >= n$ con $sin : \mathbb{N}^+ \to \mathbb{N}$ e $des : \mathbb{N}^+ \to \mathbb{N}$. Per il "ritorno" abbiamo quindi che $sin(n) = x$ e $des(n) = y$.

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;

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			

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

		y			
		...	...	y	...
x	x	...	...	$< x, y >$	
	...		...		
	$x + y$	...			
	...				

suriettiva perchè riempio fino all' n voluto e guardo immmagine $< x, y >$ corrispondente. Per esempio $< 2, 1 >= 8$.

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

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

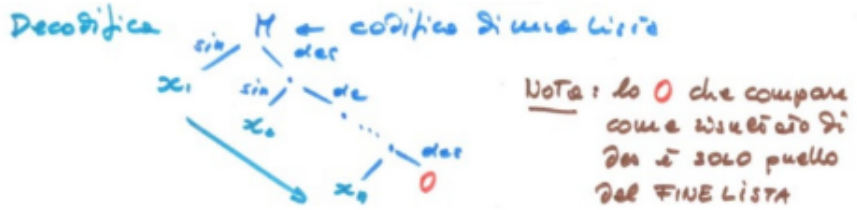
Il punto 2 è dato dal fatto che un generico valore nella colonna 0 è dato dalla somma degli indici fino a quello cercato +1, vediamo per esempio nella Tabella 4.1 che il valore 7 nella riga 3 è calcolabile come $3+2+1+0$ a cui aggiungiamo ancora 1. Unendo i due punti troviamo che

$$< x, y >=< x + y, 0 > +y = \frac{(x + y)(x + y + 1)}{1} + y + 1.$$

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

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

Figure 1: Decodifica lista



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

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

troviamo infine che $des(n) = y = n - < \gamma, 0 >$ 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 semplicentente definire una nuova funzione, ovvero

$$[,] : \mathbb{N} \times \mathbb{N} \sim \mathbb{N} \textbf{ t.c. } [x, y] =< x, y > -1$$

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

4.1.1 *DATI*$\sim\mathbb{N}$

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

Codifica: $1, 2, 5 \Rightarrow < 1, 2, 5, 0 > \Rightarrow < 1, < 2, < 5, 0 >> > \Rightarrow < 1, < 2, 16 > \Rightarrow < 1, 188 > \Rightarrow 18144$.

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):

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

Matrici:

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

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 comfronteremo le loro potenze computazionali. Se avremo $F(\_)$:

- diverse $\Rightarrow$ ciò che è computabile dipende dallo strumento
- uguale $\Rightarrow$ computabilità (tesi di Church)? posso calcosare 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, programmma, formato da istruzioni;
- R, memoria, insieme di registri e ogni cella può contenere un numero $\in \mathbb{N}$, dove R_1 contiene l'inpur e R_0 l'output.

La terminazione è data da $L = 0$.
Output: $\phi_P(x) = contenuto(R_o)$ o $\perp$ in caso di loop, indichiamo con ϕ_P la semantica di P.

Linguaggio *RAM*

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

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=Stato=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

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

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

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

stato finale : $S(L) = 0 \Rightarrow$ **HALT**

dato : $\mathbb{N}(infattiDATI \sim \mathbb{N})$

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 $PROG = \{programmi\ RAM\}, P \in PROG; |P| = \#istrizioni$

Esecuzione Dinamica del programma $\Rightarrow$ funzione stato prossimo.
$\sigma : stati \times PROG \to stati_\perp$; $\sigma(S, P) = S'$
Lo stato che segue lo stato S dopo l'eesecuzione 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 $S(L) - esima$:
   - $R_k \leftarrow R_k + / \dot{-} 1$:
     - $S'(R_k) = S(R_k) + / \dot{-} 1$
     - $S'(L) = S(L) + 1$

- $\forall i \neq k : S'(R_i) = S(R_i)$

- *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

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

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 Applichiamo 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?

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

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

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

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

Da programmi a numeri $cod(P) =< Ar(istr_1), \ldots, Ar(istr_m) >$

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 $F(RAM) \sim \mathbb{N} \not\sim \mathbb{N}^{\mathbb{N}}_{\perp}$, quindi alcuni problemi non sono automatizzabili;
- forse, cosiderando un sistem adi calcolo C più soffisticato 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 $x_0, \ldots, x_20$ 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: $x_k = 0$, $x_k = x_j + 1$, $x_k = x_j \dot{-} 1$;
- $[PASSI]$: comando while: while $x_k \neq 0$ do C, con C che può essere:
  - comando di assegnamento
  - comando while
  - comando composto
- $[PASSI]$: comando composto: $\underline{BEGIN}\ C_1, \ldots, C_m\ \underline{END}$, con C come sopra.

Possiamo quindi dire che un programma *WHILE* è un comando composto.
$w - PROG = \{programmi\ WHILE\} \leftarrow$ costruiti induttivamente.
Semantica di w: $\psi_w : \mathbb{N} \to \mathbb{N}$

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

1. dimostro proposizione sugli assegnamenti;
2. suppongo proposizione su C e la dimostro su $whilex_k \neq 0doC$;
3. suppongo vera la proposizione su $C_1, \ldots, C_m$ e la dimostro su $\underline{BEGIN}$ $C_1, \ldots, C_m$ $\underline{END}$.

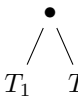
6.3.1 Dimostrazioni induttive su alberi bianri

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  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:
   - T_1: foglie F_1, nodi interni $F_1 - 1$
   - T_2: foglie F_2, nodi interni $F_2 - 1$

Dimostro vero P per:  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. $[BASE]$: $depth(\ \overset{\bullet}{\underset{T_1}{/}}\ \underset{T_2}{\backslash}\ ) = 1 + max(depth(T_1) + depth(T_2))$

7 Lezione 7

7.1 Esecuzione su una macchina WHILE (intuituvamente)

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

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

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: $(c_0, \ldots, c_{20})$ con c_i contenuto della cella x_i.

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

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

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

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

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

- $[BASE]$ Assegnamenti:

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

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

- $[PASSO]$ Comando composto: $[\underline{BEGIN}\ C_1, \dots, C_m\ \underline{END}]$, 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: $[\underline{WHILE}\ x_k \neq 0 \underline{DO}\ C]$, 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

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

dove μt è il minor numero di volte.

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

7.3 Potenza computazionale sistema WHILE

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

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 soffisticata 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-$PROG$.

$$F(C_1) = \{f \in \mathbb{N}_\perp^{\mathbb{N}} : f = \Psi_{P_1}\ per\ qualche\ P_1\ e\ C_1 - PROG\} = \{\Psi_{P_1} : P_1 \in C_1\text{-}PROG\}$$
$$F(C_2) = \{f \in \mathbb{N}_\perp^{\mathbb{N}} : f = \phi_{P_2}\ per\ qualche\ P_2\ e\ C_2 - PROG\} = \{\phi_{P_2} : P_2 \in C_2\text{-}PROG\}$$

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

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

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 $T : C_1\text{-}PROG \to C_2\text{-}PROG$ con le seguenti proprietà:

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

Se esiste $T : C_1\text{-}PROG \to C_2\text{-}PROG$ allora $F(C_1) \subseteq F(C_2)$. Dimostrazione:

> $f \in F(C_1) \Rightarrow \exists P \in C_1\text{-}PROG : f = \Psi_p$
> *a P applico T, ottengo*
> $T(P) \in C_2\text{-}PROG$ *(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 $comp : W - PROG \to PROG$. 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 $comp(C_1), \ldots, comp(C_m)$ e mostro come compilare il comando composto $\underline{BEGIN}$ $C_1, \ldots, C_m$ $\underline{END}$;

3. [$PASSO$] per ipotesi induttiva assumo dato $comp(C)$ e mostro come compilare il comando while $\underline{WHILE}\ X_k \neq 0\ \underline{DO}\ C$.

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**

- $comp(X_k := 0)$

$$\begin{aligned} &\underline{loop} : if\ R_k == 0\ then\ goto\ \underline{exit} \\ &\qquad R_k \leftarrow R_k \dot{-} 1 \\ &\qquad if\ R_{21} == 0\ then\ goto\ \underline{loop} \\ &\underline{exit} : R_k \leftarrow R_k \dot{-} 1 \end{aligned}$$

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

- $comp(X_k := X_j + / \dot{-} 1)$
  - se $k == j$: $R_k \leftarrow R_k + / \dot{-} 1$
  - altrimenti:
    1. salvo X_j in R_{22};
    2. azzero R_k;
    3. metto in R_j e R_k il contenuto di $R_2 2$ (un ciclo che fa + 1 al R_{22} e somma 1 a 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: $comp(\underline{while}\ x_k \neq 0\ \underline{do}\ C)$:

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

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: $< x, n >$ con x il dato di input e n la codifica del programma, output: $\Phi_n(x) = \Phi_P(x)$

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

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 $cod(P) = n$ allora P non userà mai R_j con $j > n$. Posso quindi restringermi a modellare la sequenza $R_0, \ldots, R_{n+2}$. Il contenuto della memoria in cui si esegue P può essere considerato in una sola variabile contenente $< a_0, a_1, \ldots, a_{n+2} >$. Avremo quindi la seguente configurazione:

- $X_0 \Leftarrow < R_0, \ldots, R_{n+2} >$;
- $X_1 \Leftarrow L$ 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. $X_1 \Leftarrow input(< x, n >)$
2. $X_2 := sin(X_1)$ ovvero il dato di input;
3. $X_3 := sin(X_1)$ ovvero il "listato" del programma P;
4. $X_1 := 1$.

9.1.2 Codice interprete in macrowhile

```
1:  while (X_1 ≠ 0) do                                          ▷ HALT con L = 0
2:      if (X_1 > length(X_2)) then X_1 := 0                    ▷ Ho finito le istrizioni
3:      else
4:          X_4 := Pro(X_1, X_3)        ▷ Prendo elemento in pos X_1 di X_3 (fetch)
5:          if (X_4 mod 3 == 0) then                            ▷ R_k ← R_k + 1
6:              X_5 := X_4/3
7:              X_0 := incr(X_5, X_0)          ▷ Increm di 1 elem in pos X_5 di X_0
8:              X_1 := X_1 + 1                      ▷ Incremento program counter
9:          end if
10:         if (X_4 mod 3 == 1) then                            ▷ R_k ← R_k ∸ 1
11:             X_5 := (X_4 − 1)/3
12:             X_0 := decr(X_5, X_0)
13:             X_1 := X_1 + 1
14:         end if
15:         if (X_4 mod 3 == 2) then                ▷ if R_k == 0 then goto m
16:             X_5 := sin((X_4 + 1)/3)                                    ▷ k
17:             X_6 := des((X_4 + 1)/3)                                    ▷ m
18:             if (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 := sin(X_0)
```

9.1.3 Conseguenze

Posso costruire $Comp : PROG \to W - PROG$:

- $n = cod(P)$
- $X_1 :=< x, n >$

- I_w

Programmabile, completo e corretto.
$\Psi_{comp(P)}(x) = \Psi_{I_w}(< x, n >) = \Phi_n(x) = \Phi_P(x)$ OK!

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

Finalmente.. $\mathbb{N}_{\perp}^{\mathbb{N}} \tilde{\sim} \mathbb{N} \sim PROG \sim F(RAM) = F(WHILE)$

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

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

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 $op : U \times \cdots \times U \to U$ con U ripetuta k volte in input: k=arietà dell'operazione.

Esempi $U = \mathbb{N}$

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

10.1.1 Chiusura

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

Esempi

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

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

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

10.1.2 Chiusura di un'insieme

Problema: Sia $A \subseteq U$ e $op : U^k \to U$. Qual è il più piccolo sottoinsieme di U che contiente 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 $+ : \mathbb{N}^2 \to \mathbb{N}$, 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 $op : U^k \to U$. Il più piccolo sottoinsieme di U contenente A è chiuso rispetto a op si ottienecalcolando la chiusura rispetto a op, cioè l'insieme A^{op} definito induttivamente come:

1. $\forall a \in A \Rightarrow a \in A^{op}$;
2. $\forall a_1, \ldots, a_k \in A^{op} \Rightarrow op(a_1, \ldots, a_k) \in A^{op}$;

3. nient'altro sta in A^{op}.

Possiamo dare la seguente definizione più operativa:

1. metti in A^{op} tutti gli element di A;
2. applica op a una k-tupla di elementi in A^{op};
3. aggiungo il riultato in A^{op} se non è già presente;
4. reitero i punti 2 e 3 finche A^{op} cresce;
5. output A^{op}.

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

1. $A^+ \leftarrow A$;
2. $2 + 3 = 5 \not\in A^+ \Rightarrow A^+ = \{2, 3, 5\}$;
3. $2 + 2, 3 + 3, 5 + 5, \ldots \not\in A^+ \Rightarrow A^+ = \{2, 3, 4, 5, 6, 10, \ldots\}$;
4. ....
5. output: $A^+ = \mathbb{N} \backslash \{0, 1\}$.

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. $\forall i \in \{1, \ldots, t\} \forall a_1, \ldots, a_k \in A^\Omega \Rightarrow op_i(a_1, \ldots, a_k) \in A^\Omega$;
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

$ELEM =$

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

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 $f(x) = x + 2 \not\in ELEM$ 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 $h : \mathbb{N}^k \to \mathbb{N}$ e $g_1, \ldots, g_k : \mathbb{N}^k \to \mathbb{N}$, denoteremo $\underline{x} \in \mathbb{N}^k$.

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

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

$$f(x) = x + 2 \not\in ELEM\ ma\ f(x) \in ELEM^{COMP}$$
$$infatti\ f(x) = COMP(SUCC, SUCC)(x) = SUCC(SUCC(x)) = f(x)$$

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

11.3 Operatore di ricorsione primitiva

Esempio Fattoriale:

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

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

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

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

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

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

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

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

con il predecessore posso definire la differenza come

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

11.4 $RICPRIM$ vs $WHILE$

$RICPRIM$ contiene già molte funzioni e potrei già chiedermi se ho raggiunto $F(WHILE)$. Mostreremo $RICPRIM \subseteq f(WHILE)$ per induzione strutturare. ($F(WHILE)$ 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 $h, g_1, \ldots, f_k \in RICPRIM \Rightarrow COMP(h, g_1, \ldots, g_k \in RICPRIM$;
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 $h, g_1, \ldots, g_k \in RICPRIM$ siano in $F(WHILE)$ e dimostro che $COMP(h, g_1, \ldots, g_k \in F(WHILE)$;
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 $h, g_1, \ldots, g_k \in RICPRIM$ siano in $F(WHILE) \Rightarrow$ esistono $H, G_1, \ldots, G_k \in W - PROG$ t.c. $\Psi_H = h, \Psi_{G_1} = g_1, \ldots$. Mostro quindi un programma $WHILE$ che calcola $COMP(h, g_1, \ldots, g_k) = h(g_1(\underline{x}), \ldots, g_k(\underline{x}))$.

Algorithm 1 $w \equiv$

```
1: BEGIN                          ▷ x_1 ← x come < a_1, ..., a_n >
2: x_0 := G_1(x_1)
3: x_0 := [x_0, G_2(x_1)]         ▷ calcolo i vari G e poi metto insieme in H
4: ...
5: x_0 := [x_0, G_k(x_1)]         ▷ x_0 := [G_1(x), ..., G_k(x)]
6: x_0 := H(x_0)                  ▷ x_0 := H(G_1(x), ..., G_k(x))
7: END
```

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

- 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 qundi un programma $WHILE$ che calcola

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

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

Algorithm 2 $w \equiv$

```
1: BEGIN                          ▷ x_1 ←< x, y >
2: t := G(x)
3: k := 1
4: while k ≤ y do
5:     t := H(t, k − 1, 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);
- $F(WHILE)$ continene 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'∞, consideramo 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:

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

  cresce troppo in fretta per essere in $RICPRIM \Rightarrow$ Ackermann $\in \tilde{F}(WHILE)$ ma $\not\in RICPRIM$ 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 $f : \mathbb{N}^{n+1} \to \mathbb{N}; f(\underline{x}, y)$ con $\underline{x} \in \mathbb{N}$. Definiamo

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

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

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

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

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

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: $\underline{BEGIN}$
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: $\quad y := y + 1$
6: **end while**
7: $\underline{END}$

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

- $[[\_]](\_) : \mathbb{N}^{21} \to \mathbb{N}^{21}$ 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: $[[\_]](\_) : \mathbb{N}^{21} \to \mathbb{N}^{21}$ 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 $[[c]](\underline{x}) = \underline{y} \in P$ con $\underline{x}, \underline{y} \in \mathbb{N}^{21}$ mostreremo che $f_c(x) = y$ con $x = [\underline{x}]$ e $y = [\underline{y}]$ CANTOR.

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

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:

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

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

    f^n: composizione di funzione n volte. ATTENZIONE: $f_c^{e(x)}$ è la composizione di f_c per $e(n)$ volte, che NON è un numero COSTANTE. So scrivere $COMP(h, g_1, \ldots, g_k)$ 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?

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

    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 minimalizzazione 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 Prprietà 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) $COMP : w{-}PROG \to PROG$ 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
$$S_1^1(n, y) = \overline{n} \ tale \ che \ \Phi_{\overline{n}}(x) = \Phi_n(x, y)$$

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

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

```
1: R_0 ← R_0 + 1
2: ...
3: R_0 ← R_0 + 1                          ▷ y volte
4: R_1 ←< R_1, R_0 >                      ▷ Creo input per P (∗)
5: R_0 ← 0                                ▷ Azzero R_0 (∗∗)
6: P                                      ▷ Richiamo P su R_1
```

Posso esplicitare la funzione S_1^1 come

$$S_1^1(n, y) = \overline{n} =< 0, \ldots, 0, s, t, n >$$

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 $\forall n, x, y \in \mathbb{N} : \Phi_n(< x, y >) = \Phi_{S_i(n,y)}(x)$

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 Sistempi 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: (ssiomi di Rogers):

1. $\{\Phi_i\} = P$ (aderisce alla tesi di Church-Turing);
2. $\exists u \in \mathbb{N} : \Phi_u(< x, n >) = \Phi_n(x)$ (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 $\Phi_k(< \underline{x}, \underline{y} >) = \Phi_{S_n^m(k,\underline{y})}(< \underline{x})$

Non sono assiomi assolutamente restrittuvi: 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 $t : \mathbb{N} \to \mathbb{N}$ tale che:

1. Correttezza: $\forall i \in \mathbb{N}$ vale $\Phi_i = \Psi_{t(i)}$;
2. Compretezza: 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):

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

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, cio 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 $t : \mathbb{N} \to \mathbb{N}$ 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

14.1 Quesiti

14.1.1 I quesito: Programmi autoreplicanti

Dato un spa, esistono all'intenro di esso progreammi che stampano se stessi (il proprio listato)? Questi programmi sono detti Quine. Se ne possono trovare in svariati linguaggio come Python, Java, C, C++, ...

In *RAM* il quesito può essere posto come $\exists j \in \mathbb{N} : \Phi_j(x) = j \forall x \in \mathbb{N}$?

14.1.2 II quesito: compilatore completamente errato

Dato due spa $\{\Phi_i\}$ e $\{\Psi_j\}$, un compilatore dal primo al secondo è una funzione $t : \mathbb{N} \to \mathbb{N}$ che sia:

1. $t \in T$ (programmabile e totale);
2. $\forall i \in \mathbb{N} : \Phi_i = \Psi_{t(i)}$ (corretta, mantiene la semantica).

Un compilatore completamente errato è una funzione $t : \mathbb{N} \to \mathbb{N}$ che sia:

1. $t \in T$ (programmabile e totale);
2. $\forall i \in \mathbb{N} : \Phi_i \neq \Psi_{t(i)}$ (completamente errato).

14.2 Teorema di ricorsione [TR]

Dato un spa, $\{\Phi_i\}$, per ogni $t : \mathbb{N} \to \mathbb{N}$ ricorsiva totale vale: $\exists n \in \mathbb{N} : \Phi_n = \Phi_{t(n)}$.

Una chiave di lettura del teorema:

- considero t come un programma che trasforma programmi: prende in ingresso un programma n e lo scambia con un programma $t(n)$, anche nella maniera più assurda;
- il teorema dice che t può essere la manipolazione più assurda possibile, ma esiste sempre almeno un programma il cui significato con sarà stravolto da t.

14.2.1 Risposta al I quesito

Consideriamo il programma P in RAM che somma 1 al registro R_0 j-volte. $COD(P) =< 0, \ldots, 0, >= Z(j) \in T$. La codifica avrà quindi 0 (aritmetizzazione di $R_0 = R_0 + 1$), ripetuto j volte in cantor. Quindi $\Phi_{Z(j)} = j$.

Per TR:

$$\exists j \in \mathbb{N} : \Phi_j(x) = \Phi_{Z(j)}(x) = j \Rightarrow \exists j \in \mathbb{N} : \Phi_j(x) = j$$

I quesito: SI!!! per RAM, in generale per tutti gli spa che ammettono sempre codifiche di programi.

14.2.2 Risposta al II quesito

Assiomi spa:

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

Considero una qualunque trasformazione di programmi $t \in T$.

$$(*)\ \Psi_{t(i)} =^{2)}\ \Psi_u(x, t(i)) =^{1)}= \Phi_e(x, t(i)) =^{3)}\ \Phi_{S_1^1(e,t(i))}(x) =$$

Cosideriamo $S_1^1(e, t(i))$: questa è una funzione a 1 varibile in quanto e è fissato, in aggiunta S_1^1 e $t(i)$ sono funzioni totali. Quindi:

$$= \Phi_{g(i)}(x)$$

Per TR: $\exists i \in \mathbb{N} : \Phi_i = \Phi_g(i)$ $(**)$

$(*) + (**) \Rightarrow \exists i \in \mathbb{N} : \Psi_{t(i)} = \Phi_i \forall t \in T$. La risposta è quindi NO!!!

14.2.3 Dimostrazione TR

Dato un spa $\{\Phi_i\}$, $\forall t : \mathbb{N} \to \mathbb{N} \in T \exists n \in \mathbb{N}$ *t.c.* $\Phi_n = \Phi_{t(n)}$. Per semplicità scriveremo $\Phi_n(x, y)$ per $\Phi_n(< x, y >)$.

$\Phi_{\Phi_i(i)}(x) =^{2)}\ \Phi_{\Phi_u(i,i)}(x) =^{2)}\ \Phi_u(x, \Phi_i(i)) =$ *composizione di funzioni* $\in P = f(x, i) \in P = \ldots$

Diamo qualche spiegazione/dettaglio ai passaggi fatti:

- $\Phi_i(i)$ è un numero, l'output del programma i su input i;
- $\Phi_u(x, \Phi_i(i))$ è una funzione in due variabili: i, x;

Continuiamo ora la sequenza di ugualianze:

$$\ldots =^{1)}\ \Phi_e(x, i) =^{3)}= \Phi_{S_1^1(e,i)}(x)\ (*)$$

Possiamo usare "l'operazione" $=^{1)}$ perchè abbiamo in programma per ogni funzione in P (assioma 1 degli spa).

Ora considero $t(S_1^1(e,i))$: è una funzione in i ricorsiva totale perchè composizione di t e $S_1^1 \Rightarrow^{1)}$

$$\exists m \in \mathbb{N} : \Phi_m(i) = t(S_1^1(e,i)) \ (**)$$

Pongo $n = S_1^1(e,m)$. Mostriamo che per tale n vale $Phi_n = \Phi_{t(n)}$. Riassumiamo le vare equazioni e l'assegnamento che abbiamo appena fatto:

- $\Phi_{\Phi_i(i)}(x) = \Phi_{S_1^1(e,i)}(x) \ (*)$;
- $\Phi_m(i) = t(S_1^1(e,i)) \ (**)$;
- $n = S_1^1(e,m)$.

Quindi:

1. $\Phi_n(x) = \Phi_{S_1^1(e,m)}(x) =^{(*)} \Phi_{\Phi_m(m)}(x)$;
2. $\Phi_{t(n)}(x) = \Phi_{t(S_1^1(e,m)}(x) =^{(**)} = \Phi_{\Phi_m(m)}(x)$

QUINDI $\Phi_n = \Phi_{t(n)}$.

15 Lezione 15

15.1 Problemi di decisione

Definiti in modo generale da:

- NOME = nome del problema di decisione;
- ISTANZA (generica) = dominio degli oggetti che verranno considerati;
- DOMANDA = proprietà che gli oggetti del dominio possono soddisfare o meno; dato un oggetto del dominio, devo rispondere SI se l'oggetto soddisfa la proprietà, altrimenti rispondo NO.

Una descrizione più formale è:

- Π = nome del problema;
- ISTANZA: $x \in D$
- DOMANDA: $p(x)$?, oggetto $x \in D$ soddisfa la proprietà $p \Leftrightarrow p(x)$ è vera?

Esempi

- PARITA'
- ISTANZA: $n \in \mathbb{N}$ (un numero dato in input si chiamano istanze particolari)
- DOMANDA: n è pari?

- EQUAZIONI DIOFANTEE
- ISTANZA: $a, b, c \in \mathbb{N}^+$
- DOMANDA: $\exists x, y \in \mathbb{Z} : ax + by = c$?

15.1.1 Decidibilità

Π è decidibile sse $\exists$ programma P_Π t.c. $\forall x \in D$: 1 se $p(x)$; 0 se $\neg p(x)$.

Equivalentemente: associamo a Π la sua funzione soluzione:

$$\Phi_\Pi : D \to \{0,1\} \ tale\ che\ \Phi_\Pi(x) = \begin{cases} 1 & se\ p(x) \\ 0 & se\ \neg p(x) \end{cases}$$

Π è decidibile sse $\Phi_\Pi \in T$. Le due definizioni si equivalgono:

1. il programma P_Π calcola $\Phi_\Pi \Rightarrow \Phi_\Pi \in T$;
2. se $\Phi_\Pi \in T$ allora esiste un programma che calcola Φ_Π e che il comportamenti di $¶_\Pi$.

Dato il problema di decisione Π, posso dimostrare la decidibilità in questi due modi:

1. esibendo l'algoritmo di soluzione, oppure
2. mostrare che $\Phi_P i$ è ricorsiva totale.

Esempi

1. PARITA' (PR): $\Phi_{PR}(n) = 1 \dot{-} (n \ mod\ 2) \in T$
2. EQ DIOFANTEA (ED): $\Phi_{ED}(a, b, c) = 1 \dot{-} (c \ mod\ (mcd\ (a, b)) \in T$