Informatica Teorica

Mauro Tellaroli

Prof. Mereghetti

Indice

0 Introduzione

L'informatica è la disciplina che studia l'informazione e la sua elaborazione **automatica**. L'elaborazione in questione non è legata a nessun mezzo, si tratta quindi di una qualsiasi elaborazione che può avvenire con o senza un computer.

Obiettivo di questo corso è rispondere a due domande:

1. Cosa è calcolabile automaticamente? → Teoria della calcolabilità
2. Quanto "costa" risolvere un problema? → Teoria della complessità

1 Prerequisiti matematici

Classi di funzioni $f : A \to B$

Iniettive

f è iniettiva se $\forall a_1, a_2 \in A : a_1 \neq a_2 \Rightarrow f(a_1) \neq f(a_2)$

Suriettive

f è suriettiva se $\forall b \in B \;\; \exists a \in A : f(a) = b$

Biettive

f è biettiva se è sia iniettiva che suriettiva.

Composizione di funzioni

Date $f : A \to B$ e $g : B \to C$, si definisce f composto g come la funzione $g \circ f : A \to C$ come:

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

La composizione non è un operatore commutativo.

Funzioni parziali e totali

La notazione $f(a)\downarrow$ indica che la funzione è definita su a, ovvero che esiste un valore b del codominio tale che $f(a) = b$.

Al contrario, la notazione $f(a)\uparrow$ indica che la funzione **non** è definita su a.

Una funzione $f : A \to B$ definita su tutto il suo dominio è detta totale. Se invece esistono dei valori del dominio nei quali f non è definita, f è detta parziale:

$$f \text{ è } \textbf{totale} \text{ se } \forall a \in A \;\; f(a)\downarrow$$

$$f \text{ è } \textbf{parziale} \text{ se } \exists a \in A : f(a)\uparrow$$

Campo di esistenza

Dalla definizione di funzione parziale si intuisce come l'insieme di tutti i valori nel quale la funzione $f : A \to B$ è definita, non sempre coincide con il dominio A. Questo insieme è detto **campo di esistenza di** f e si denota con Dom_f:

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

Totalizzazione di una funzione parziale

Presa una funzione $f : A \to B$ parziale, la si può totalizzare, ovvero rendere totale, aggiungendo al codominio un valore $\perp$ che rappresenta il caso indefinito:

$$f : A \to B \underset{\text{totalizzazione}}{\longrightarrow} f : A \to B \cup \{\perp\}$$

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

L'insieme $B \cup \{\perp\}$ viene abbreviato con $B_\perp$.

Prodotto cartesiano

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

L'operatore $\times$ non gode della proprietà commutativa.

$$\underbrace{A \times A \times \cdots \times A}_{n \text{ volte}} = A^n$$

Insiemi di funzioni

Tutte le funzioni che vanno da A a B è detto B^A:

$$B^A = \{f : A \to B\}$$

$$B_\perp^A = \{f : A \to B_\perp\}$$

Funzione di valutazione

Si definisce funzione di valutazione $\omega : B_\perp^A \times A \to B$ con:

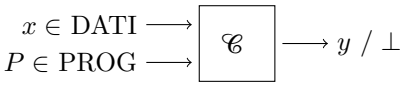
$$w(f, a) = f(a)$$

- Fissando a provo tutte le funzioni su a;
- Fissando f ottengo il suo grafico.

2 Teoria della calcolabilità

2.1 Sistema di calcolo $\mathscr{C}$

Si vuole modellare matematicamente un calcolatore o sistema di calcolo $\mathscr{C}$:



La figura mostra il sistema di calcolo $\mathscr{C}$ che, preso un programma P su input x, restituisce in output il risultato y o il valore $\perp$ se il programma va in loop.

DATI è l'insieme di tutti i possibili dati di input e PROG l'insieme di tutti i possibili programmi.

Il sistema di calcolo $\mathscr{C}$ non fa altro che eseguire il programma P su input x ricavandone il risultato y:

$$\mathscr{C} : \text{PROG} \times \text{DATI} \to \text{DATI}_\perp \tag{1}$$

Quello che fa il programma P è trasformare il dato di input x in un dato di output y; si può quindi dire che un programma non è altro che una funzione che agisce da DATI in DATI:

$$\begin{gathered} P : \text{DATI} \to \text{DATI}_\perp \\ \Downarrow \\ \text{PROG} = \text{DATI}_\perp^{\text{DATI}} \end{gathered} \tag{2}$$

La funzione associata al programma P è detta **semantica di** P.

Da (1) e (2) si ottiene che:

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

$\mathscr{C}$ è una funzione di valutazione; $\mathscr{C}(P, x)$ è infatti la semantica di P.

2.2 Potenza computazionale di $\mathscr{C}$

Si definisce potenza computazionale di $\mathscr{C}$:

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

$F(\mathscr{C})$ **contiene tutto ciò che un qualsiasi sistema di calcolo** $\mathscr{C}$ **può calcolare**. Quindi, per stabilire cosa l'informatica può risolvere, basta stabilire il carattere dell'inclusione:

- $F(\mathscr{C}) \subset \text{DATI}_\perp^{\text{DATI}} \Rightarrow$ esistono problemi che l'informatica non può risolvere;
- $F(\mathscr{C}) = \text{DATI}_\perp^{\text{DATI}} \Rightarrow$ l'informatica può risolvere tutto.

2.3 Cardinalità di insiemi infiniti

Per riuscire a capire se l'inclusione $F(\mathscr{C}) \subseteq \text{DATI}_\perp^{\text{DATI}}$ sia propria o meno, si confronterà la cardinalità dei due insiemi. Infatti dalla cardinalità si può ricavare che:

- Se $|F(\mathscr{C})| < \left|\text{DATI}_\perp^{\text{DATI}}\right| \quad \Rightarrow \quad F(\mathscr{C}) \subset \text{DATI}_\perp^{\text{DATI}}$;
- Se $|F(\mathscr{C})| = \left|\text{DATI}_\perp^{\text{DATI}}\right| \quad \Rightarrow \quad F(\mathscr{C}) = \text{DATI}_\perp^{\text{DATI}}$.

Il concetto di cardinalità è semplice quando si tratta di insiemi finiti: basta contare il numero di elementi che compongono l'insieme. Tuttavia, in presenza di insiemi infiniti le cose si complicano.

Per esempio, si confrontino $\mathbb{N}$ e $\mathbb{R}$: entrambi hanno cardinalità infinita ($|\mathbb{N}| = |\mathbb{R}| = \infty$) eppure $\mathbb{N} \subset \mathbb{R}$! Per comprendere quindi meglio la cardinalità di insiemi infiniti si dovrà andare più nel dettaglio.

2.3.1 Relazione binaria

Si definisce relazione binaria R sull'insieme A, un elenco di coppie ordinate di elementi di A: $R \subseteq A^2$. Due elementi $a, b \in A$ sono in relazione R se $(a, b) \in R$. Si usa la notazione:

- $a \ R \ b$: a è in relazione R con b;
- $a \not R \ b$: a non è in relazione R con b;

2.3.2 Relazione di equivalenza

$R \subseteq A^2$ è una relazione di equivalenza se gode di:

1. Riflessività: $\forall a \in A \quad a \ R \ a$
2. Simmetria: $\forall a, b \in A \quad a \ R \ b \ \Leftrightarrow \ b \ R \ a$
3. Transitività: $\forall a, b, c \in A \quad a \ R \ b \ \wedge \ b \ R \ c \Rightarrow a \ R \ c$

2.3.3 Classe di equivalenza

Si definisce classe di equivalenza $[a]_R$ l'insieme degli elementi in relazione R con a:

$$[a]_R = \{b \in A : a \ R \ b\}$$

Tutte le classi di equivalenza di R formano una partizione di A. L'insieme A partizionato attraverso le classi di equivalenza di R è detto **quoziente** di A rispetto a R ed è denotato da $A/_R$.

Esempio

Si consideri la relazione $\equiv_4 \subseteq \mathbb{N}^2$ di equivalenza modulo 4. Due numeri sono in relazione di equivalenza modulo 4 se il resto della divisione per 4 è uguale per entrambi.

$$5 \equiv_4 9 \ , \ 10 \equiv_4 2 \ , \ \ldots$$

Le classi di equivalenza sono:

$$\begin{aligned} [0]_4 &= \{4k\} && \text{(Multipli di 4)} \\ [1]_4 &= \{4k+1\} && \text{(Resto 1)} \\ [2]_4 &= \{4k+2\} && \text{(Resto 2)} \\ [3]_4 &= \{4k+3\} && \text{(Resto 3)} \end{aligned}$$

L'insieme $\{[0]_4, [1]_4, [2]_4, [3]_4\} = \mathbb{N}/_{\equiv_4}$ è una partizione di $\mathbb{N}$.

2.3.4 Insiemi isomorfi

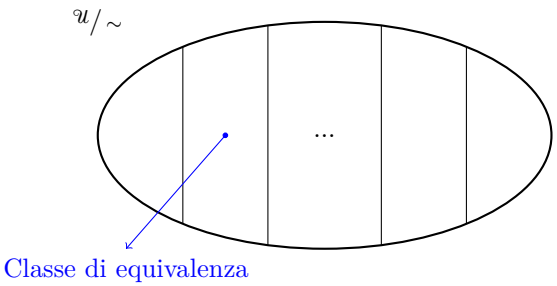
Due insiemi A e B sono **isomorfi** (o equinumerosi) se esiste una funzione biettiva tra essi. Formalmente si indica con:

$$A \sim B$$

La relazione di isomorfismo $\sim$ è una relazione di equivalenza in quanto:

1. Riflessiva: si usi la funzione identità;
2. Simmetrica: se esiste una funzione biettiva allora anche la sua inversa è biettiva;
3. Transitiva: la composizione di due funzioni biettive è una funzione biettiva.

Sia $\mathcal{U}$ l'insieme universo, ovvero l'insieme che contiene tutti gli insiemi. Il quoziente di $\mathcal{U}$ rispetto a $\sim$ ($\mathcal{U}/\sim$) definisce il concetto di cardinalità:



Ogni partizione di $\mathcal{U}/\sim$ contiene gli insiemi tra loro isomorfi, ovvero che hanno la stessa cardinalità.

Insiemi finiti

Si definisca la famiglia di insiemi:

$$J_n = \begin{cases} \varnothing & n = 0 \\ \{1, \dots, n\} & n > 0 \end{cases}$$

$$J_0 = \{\}\ ,\ J_1 = \{1\}\ ,\ J_2 = \{1, 2\}\ ,\ J_3 = \{1, 2, 3\}\ ,\ \dots$$

Un'insieme A ha cardinalità finita se $\exists n \in \mathbb{N} : A \sim J_n$ e si può dire che $|A| = n$.

Insiemi infiniti

Un insieme che non è finito ha cardinalità infinita.

2.3.5 Insiemi numerabili

Un insieme A è numerabile se $\mathbb{N} \sim A$ (ovvero $A \in [\mathbb{N}]_\sim$). Vuole quindi dire che esiste una biezione $f : \mathbb{N} \to A$ che permette di listare A come:

$$A = \{f(0), f(1), f(2), \dots\}$$

senza tralasciare nessun elemento.

Esempi

PARI :	$f(n) = 2n$
DISPARI :	$f(n) = 2n + 1$
$\mathbb{Z}$:	mappo i pari nei non-negativi e i dispari nei negativi
$\{0\} \cup 1\{0, 1\}^*$:	converto da binario a decimale

2.3.6 Insiemi non numerabili

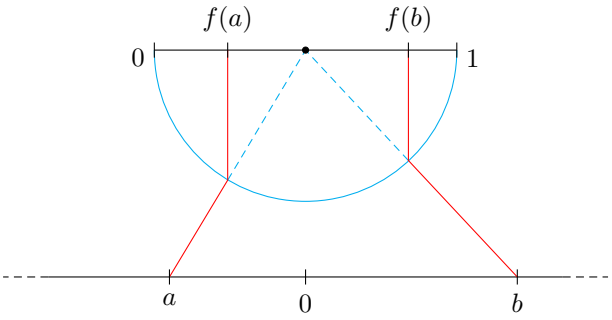
Gli insiemi non numerabili sono insiemi a cardinalità infinita ma non listabili come $\mathbb{N}$ (sono "più fitti"). Il re di questi insiemi è $\mathbb{R}$.

Teorema 1. *$\mathbb{R}$ è un insieme non numerabile:*

$$\mathbb{N} \nsim \mathbb{R}$$

Dimostrazione. Per dimostrarlo dimostro che:

1. $\mathbb{R} \sim (0,1)$: la biezione è rappresentata graficamente in figura:



   (In realtà $\mathbb{R}$ è isomorfo a un suo qualsiasi intervallo).

2. $\mathbb{N} \nsim (0,1)$: dimostrazione per assurdo: assumo che $\mathbb{N} \sim (0,1)$; Questo vorrebbe dire che tutti i numeri compresi tra 0 e 1 sono numerabili. Elenco tutti i numeri associandoli a un numero naturale:

$$
\begin{array}{lcl}
0 & \mapsto & 0.a_{00}\ a_{01}\ a_{02}\ a_{03}\ a_{04}\ \dots \\
1 & \mapsto & 0.a_{10}\ a_{11}\ a_{12}\ a_{13}\ a_{14}\ \dots \\
2 & \mapsto & 0.a_{20}\ a_{21}\ a_{22}\ a_{23}\ a_{24}\ \dots \\
3 & \mapsto & 0.a_{30}\ a_{31}\ a_{32}\ a_{33}\ a_{34}\ \dots \\
4 & \mapsto & 0.a_{40}\ a_{41}\ a_{42}\ a_{43}\ a_{44}\ \dots \\
\vdots & \vdots & \vdots\ \vdots\ \vdots\ \ddots
\end{array}
$$

   a_{ij} è la i-esima cifra dopo lo zero del j-esimo numero nella lista.
   Se $(0,1)$ fosse numerabile tutti i suoi numeri dovrebbero far parte della lista.
   Si consideri il numero:

$$0.c_0c_1c_2c_3\dots$$

   con:

$$c_i = \begin{cases} 2 & a_{ii} \neq 2 \\ 3 & a_{ii} = 2 \end{cases}$$

   Chiaramente $0.c_0c_1c_2c_3\cdots \in (0,1)$ ma non appare nella lista:

   - Differisce dal primo numero perchè $c_0 \neq a_{00}$;
   - Differisce dal secondo numero perchè $c_1 \neq a_{11}$;
   - ...
   - Differisce da qualunque numero nella lista sulla cifra diagonale.

   Ho trovato l'assurdo quindi $\mathbb{N} \nsim (0,1)$ (dimostrazione per diagonalizzazione).

Sfruttando la transitività di $\sim$ posso si può affermare quindi che:

$$\mathbb{R} \underset{(1)}{\sim} (0,1) \underset{(2)}{\nsim} \mathbb{N} \quad \Rightarrow \quad \mathbb{R} \nsim \mathbb{N}$$

□

Tutti gli insiemi isomorfi a $\mathbb{R}$ sono detti continui. Altri insiemi non numerabili sono:

- $2^{\mathbb{N}} \nsim \mathbb{N}$: insieme delle parti di $\mathbb{N}$ ovvero $2^{\mathbb{N}} = \{\text{sottoinsiemi di } \mathbb{N}\}$
- $\mathbb{N}_\perp^{\mathbb{N}} \nsim \mathbb{N}$: insieme delle funzioni da $\mathbb{N}$ a $\mathbb{N}$ ovvero $\mathbb{N}_\perp^{\mathbb{N}} = \{f : \mathbb{N} \to \mathbb{N}_\perp\}$

2.4 Esistono funzioni non calcolabili?

Ora che il concetto di cardinalità è più chiaro, si riprenda il concetto di potenza computazionale di un sistema di calcolo $\mathscr{C}$ (paragrafo 2.2):

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

Per definizione $F(\mathscr{C})$ ha la stessa numerosità di PROG:

$$F(\mathscr{C}) \sim \text{PROG}$$

Ragionevolmente, **ma non formalmente**, si può notare che:

- PROG $\sim \mathbb{N}$: si prenda la stringa binaria con la quale il programma è salvato sul disco e si converta da binario a decimale;
- DATI $\sim \mathbb{N}$: si applichi lo stesso ragionamento del punto precedente.

Ne segue che:

$$\begin{gathered} F(\mathscr{C}) \sim \text{PROG} \sim \mathbb{N} \nsim \mathbb{N}_\perp^{\mathbb{N}} \sim \text{DATI}_\perp^{\text{DATI}} \\ \Downarrow \\ F(\mathscr{C}) \nsim \text{DATI}_\perp^{\text{DATI}} \\ \Downarrow \\ F(\mathscr{C}) \subset \text{DATI}_\perp^{\text{DATI}} \end{gathered}$$

Quello che questa osservazione dice è che ho pochi programmi ($\mathbb{N}$) e troppe funzioni ($\mathbb{N}_\perp^{\mathbb{N}}$). **Alla domanda "Esistono funzioni non calcolabili?" si può quindi rispondere con un sì!**

2.5 DATI $\sim \mathbb{N}$

Obiettivo di questa sezione è dimostrare formalmente che:

$$\text{DATI} \sim \mathbb{N}$$

Vogliamo quindi trovare una biezione che è in grado di associare biunivocamente dei dati a un numero e quindi anche di ottenere i dati di partenza dal numero. Per farlo si userà il seguente teorema.

Teorema 2. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+$

Dimostrazione. Si definisca la funzione coppia di Cantor $\langle\ ,\ \rangle$:

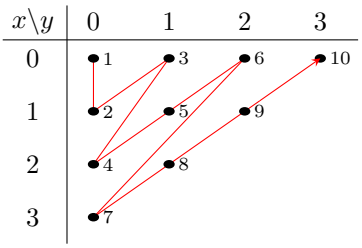
$$\langle\ ,\ \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+$$

$\langle\ ,\ \rangle$ associa biunivocamente una coppia di numeri x e y a un numero n:

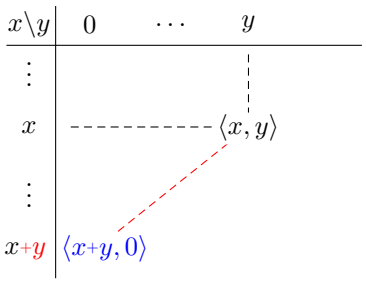
$$\langle x, y \rangle = n$$

La mappa che $\langle\ ,\ \rangle$ usa per assegnare i valori di ogni coppia viene descritta nelle seguenti tabelle:

$x\backslash y$	0	1	2	3	4
0	1	3	6	10	15
1	2	5	9	14	...
2	4	8	13	...	
3	7	12	...		
4	11	...			
5	↗				



Si vuole calcolare ora la forma analitica di $\langle\ ,\ \rangle$; si prenda una generica coppia di numeri $\langle x, y \rangle$:



Per come è definita $\langle x, y \rangle$ (vedi tabella precedente) si ha che:

$$\langle x, y \rangle = \langle x + y, 0 \rangle + y \tag{3}$$

Ora l'incognita da calcolare resta $\langle z, 0 \rangle$ che, si può ottenere come:

$$\langle z, 0 \rangle = \sum_{i=0}^{z} i + 1 = \frac{z(z+1)}{2} + 1 \tag{4}$$

Da (3) e (4) segue che:

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

$\langle\ ,\ \rangle$ si dimostra quindi mappare univocamente le coppie di numeri in numeri ($\mathbb{N}^2 \to \mathbb{N}^+$). Si cercherà ora di mostrare il passaggio inverso, ovvero come riottenere la coppia di numeri dal numero risultante ($\mathbb{N}^+ \to \mathbb{N}^2$).

Si definiscano le seguenti funzioni:

$$\langle x, y\rangle = n \quad , \quad \sin(n) = x \quad , \quad \mathrm{des}(n) = y$$

Da (3) si ha che:

$$\begin{aligned} y &= \langle x, y\rangle - \langle x + y, 0\rangle \\ &= n - \langle x + y, 0\rangle \\ &= n - \langle \gamma, 0\rangle \end{aligned}$$

Il valore di γ è il più grande valore che, messo sulla prima colonna ($\langle\gamma, 0\rangle$), non supera n:

$$\gamma = \max\{z \in \mathbb{N} : \langle z, 0\rangle \leq n\}$$

$$\langle z, 0\rangle \leq n$$

$$\frac{z(z+1)}{2} + 1 \leq n$$

$$z^2 + z + 2 - 2n \leq 0$$

$$\frac{-1 - \sqrt{8n-7}}{2} \leq z \leq \frac{-1 + \sqrt{8n-7}}{2}$$

$$\Downarrow$$

$$\gamma = \left\lfloor \frac{-1 + \sqrt{8n-7}}{2} \right\rfloor$$

In conclusione:

$$\begin{aligned} \mathrm{des}(x) &= n - \langle \gamma, 0\rangle \\ \sin(x) &= \gamma - des(x) \qquad \text{(non è il seno)} \end{aligned}$$

La funzione coppia di Cantor $\langle\ ,\ \rangle$ si è quindi mostrata essere una biezione tra $\mathbb{N}^+$ e $\mathbb{N}^2$, mostrando che i due insiemi hanno la stessa cardinalità. □

È facile poi, partendo da $\langle\ ,\ \rangle$, creare una biezione tra $\mathbb{N}$ e $\mathbb{N}^2$ (dimostrando che $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$):

$$[\ ,\ ] : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$$

$$[x, y] = \langle x, y\rangle - 1$$

Il precedente risultato mette alla luce anche che:

$$\mathbb{Q} \sim \mathbb{N}$$

in quanto ogni suo elemento non è altro che una coppia di numeri messi a frazione.

Ora che si ha appurato l'esistenza di una biezione tra coppie di numeri e numeri si può facilmente estendere questa relazione a liste d'interi, dove con lista si intende una sequenza di numeri di lunghezza non nota:

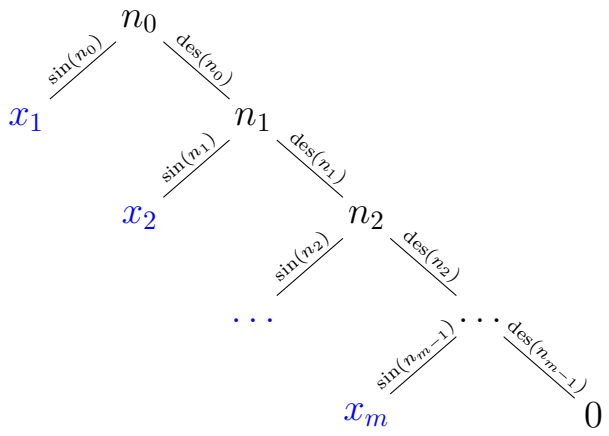
$$x_1, x_2, \ldots, x_m \to \langle x_1, x_2, \ldots, x_m\rangle$$

Per farlo basterà applicare $\langle\ ,\ \rangle$ come segue:

$$\langle x_1, x_2, \ldots, x_m \rangle = \langle x_1, \langle x_2, \langle \ldots, \langle x_m, 0 \rangle \ldots \rangle \rangle \rangle$$

Dove lo 0 a destra della coppia di Cantor più interna rappresenta il fine lista.

La decodifica invece avverrà nel seguente modo:



Qualsiasi tipo di dato può essere convertito a una lista di numeri:

- Testi: non sono altro che liste di caratteri i quali possono essere convertiti in numeri tramite tabella ASCII;
- Suoni: si usa un campionamento a una data frequenza ottenendo una lista di valori;
- Matrici: una matrice è una lista di liste;
- Immagini: ogni pixel contiene la codifica numerica di un colore; in questo modo un'immagine non è altro che una matrice di numeri grande quanto la sua risoluzione;
- Grafi: uso liste o matrici di adiacenza.

Si può quindi affermare che DATI $\sim \mathbb{N}$.

2.6 PROG $\sim \mathbb{N}$

Obiettivo di questa sezione è dimostrare formalmente che:

$$\mathrm{PROG} \sim \mathbb{N}$$

Per poterlo fare si dovrà definire formalmente un sistema di calcolo specifico: il sistema di calcolo RAM, composto dalla macchina RAM e il linguaggio RAM. Quest'ultimo si può riassumere come un assembly molto semplificato.

L'idea è di usare il sistema RAM come rappresentativo di tutti i possibili sistemi di calcolo; ne segue che $F(\mathrm{RAM})$, ovvero la potenza computazionale di un sistema RAM, permetterà di capire cosa i sistemi di calcolo sono in grado di calcolare.

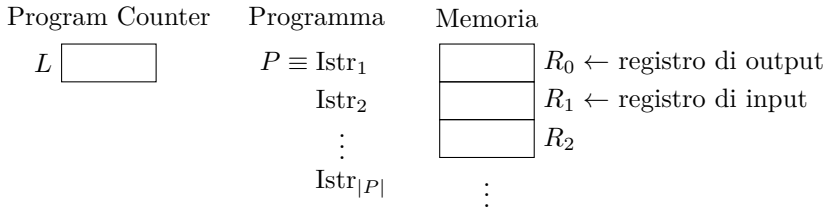
Può però sorgere spontaneo un dubbio: il sistema RAM non è troppo semplice per rappresentare tutti i sistemi di calcolo? Se il sistema RAM non fosse in grado di risolvere certi problemi, magari altri sistemi più complessi lo sarebbero.

Per verificare questo caso si vedrà successivamente un altro sistema di calcolo più sofisticato: quello WHILE. Il confronto tra le due potenze computazionali porterebbe a:

- $F(\mathrm{WHILE}) \neq F(RAM) \Rightarrow$ la computabilità dipende dallo strumento usato;
- $F(\mathrm{WHILE}) = F(RAM) \Rightarrow$ la computabilità è intrinseca nei problemi (tesi di Church-Turing).

2.6.1 Sistema di calcolo RAM

Macchina RAM



- L contiene l'indirizzo della prossima istruzione da eseguire ($1 \leq L \leq |P|$)
- P è il programma ovvero una lista di istruzioni Istr_i
- R_i è un generico registro di memoria che può contenere un numero naturale:
  - R_0 è il registro specifico dove verrà deposto l'output del programma
  - R_1 è il registro specifico dove verrà letto l'input del programma
  - Il numero dei registri è illimitato

Linguaggio RAM

La sintassi del linguaggio RAM è molto intuitiva; ci sono tre tipi di istruzioni:

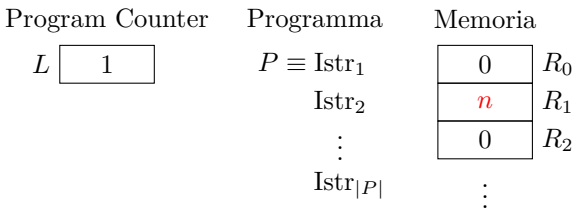
1. $R_k \leftarrow R_k + 1$
2. $R_k \leftarrow R_k \dot{-} 1$
3. `IF` $R_k = 0$ `THEN GOTO` m

$$x \dot{-} y = \begin{cases} x - y & x \geq y \\ 0 & \text{altrimenti} \end{cases}$$

Si noti che il numero di istruzione m usato nel terzo comando deve essere compreso tra 1 e $|P|$ inclusi.

Esecuzione

Per eseguire un programma P su input n la macchina verrà inizializzata come segue:



Successivamente si eseguirà un'istruzione dopo l'altra, a partire dalla prima, facendo quindi incrementare di uno il program counter ($L \leftarrow L + 1$) dopo l'esecuzione di ogni istruzione. Se l'istruzione è un'istruzione di salto (`IF` $R_k = 0$ `THEN GOTO` m) e la sua condizione $R_k = 0$ è verificata, il program counter verrà cambiato in $L \leftarrow m$.

Per convenzione $L = 0 \Rightarrow$ Fine del programma (con possibiliità di loop infinito).

L'output del programma sarà il contenuto di R_0 o $\perp$ in presenza di loop.

Semantica operazionale

La semantica operazionale descrive formalmente il significato di ogni istruzione; per farlo specifica l'effetto che l'istruzione ha sui registri della macchina.

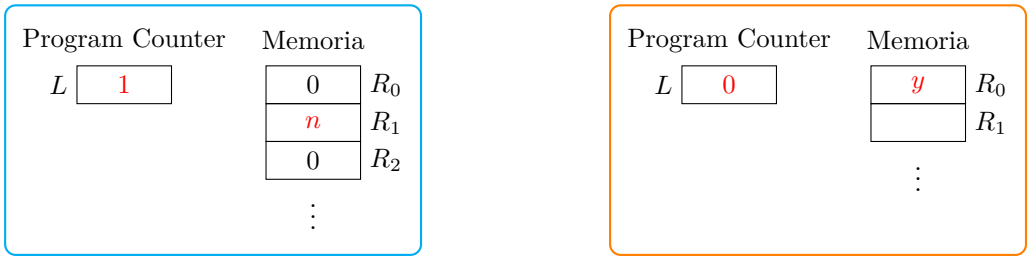
L'esecuzione di un programma è una sequenza di stati della macchina, dove uno stato descrive precisamente l'attuale situazione della macchina. Ogni istruzione fa passare la macchina da uno stato ad un altro:

$$\text{STATO}_1 \to \boxed{\text{Istr}_i} \to \text{STATO}_2$$

$$(\text{STATO}_1, \text{STATO}_2) = \text{semantica operazionale di Istr}_i$$

Ampliando il concetto di semantica dalla singola istruzione all'intero programma si ha che quest'ultimo induce una sequenza di stati:

$$\boxed{\mathcal{S}_{init}} \to \mathcal{S}_1 \to \mathcal{S}_2 \to \cdots \to \boxed{\mathcal{S}_{fin}}$$



La semantica di P è:

$$\varphi_P : \mathbb{N} \to \mathbb{N}_\perp$$

$$\varphi_P(n) = \begin{cases} \perp & \text{se } P \text{ va in loop} \\ y & \text{altrimenti} \end{cases}$$

Stato

Come già anticipato, uno stato è una "foto" di tutte le componenti della macchina in un dato istante. Formalmente si definisca uno stato come una funzione:

$$\mathcal{S} : \{L, R_i\} \to \mathbb{N}$$

$$\mathcal{S}(R_j) = \text{contenuto del registro } R_j \text{ quando la macchina è nello stato } \mathcal{S}$$

I possibili stati della macchina sono:

$$\text{STATI} = \mathbb{N}^{\{L, R_i\}}$$

Uno stato è finale se $\mathcal{S}(L) = 0$.

La funzione di inizializzazione in, preso l'input del programma, restituisce lo stato iniziale:

$$in : \text{DATI} \to \text{STATI}$$

$$in(n) = \mathcal{S}_{init}$$

$$\mathcal{S}_{init}(L) = 1 \qquad \mathcal{S}_{init}(R_i) = \begin{cases} n & i = 1 \\ 0 & i \neq 1 \end{cases}$$

Funzione stato prossimo

A definire la dinamica del programma è la funzione stato prossimo δ:

$$\delta : \text{STATI} \times \text{PROG} \to \text{STATI}_\perp$$

$$\delta(\mathcal{S}, P) = \mathcal{S}'$$

Stato attuale Stato prossimo

1. Se $\mathcal{S}(L) = 0$ allora $\mathcal{S}' = \perp$
2. Se $\mathcal{S}(L) > |P|$ allora $\mathcal{S}'(L) = 0$ e $\forall i : \mathcal{S}'(R_i) = \mathcal{S}(R_i)$
3. Se $1 \leq \mathcal{S}(L) \leq |P|$: considera la $\mathcal{S}(L)$-esima istruzione:
   (a) Se $R_k \leftarrow R_k + / \dot{-} 1$ allora:
   - $\mathcal{S}'(R_k) = \mathcal{S}(R_k) + / \dot{-} 1$
   - $\mathcal{S}'(L) = \mathcal{S}(L) + 1$
   - $\forall i : i \neq k \quad \mathcal{S}'(R_i) = \mathcal{S}(R_i)$

   (b) Se `IF` $R_k = 0$ `THEN GOTO` m allora:
   - Se $\mathcal{S}(R_k) = 0$ allora $\mathcal{S}'(L) = m$
   - Altrimenti $\mathcal{S}'(L) = \mathcal{S}(L) + 1$

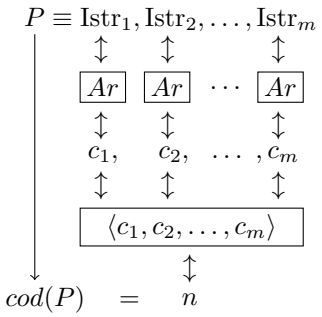
Esempio di programma

$P \equiv$ `IF` $R_1 = 0$ `THEN GOTO` 6
$R_0 \leftarrow R_0 + 1$
$R_0 \leftarrow R_0 + 1$
$R_1 \leftarrow R_1 \dot{-} 1$
`IF` $R_2 = 0$ `THEN GOTO` 1
$R_1 \leftarrow R_1 \dot{-} 1$

$\varphi_P(n) = 2n$

Aritmetizzazione di un programma RAM

Essendo un programma RAM una lista di istruzioni, per poter codificare e decodificare dei programmi basterà trovare una funzione Ar che codifica le singole istruzioni, ottenenendo una lista di numeri che si può facilmente codificare con $\langle\ ,\ \rangle$:



L'associazione biunivoca di un numero ad una struttura si dice aritmetizzazione o Gödellizzazione.

Aritmetizzazione delle istruzioni RAM

Per poter aritmetizzare un'istruzione RAM c'è bisogno di una funzione che:

$$Ar : \text{Istr} \to \mathbb{N} \quad , \quad Ar^{-1} : \mathbb{N} \to \text{Istr}$$

Il linguaggio RAM è formato da tre tipi di istruzioni; si può quindi ottenere la seguente aritmetizzazione:

$$Ar(R_k \leftarrow R_k + 1) = 3k$$
$$Ar(R_k \leftarrow R_k \dot{-} 1) = 3k + 1$$
$$Ar(\texttt{IF } R_k = 0 \texttt{ THEN GOTO } m) = 3\langle k, m\rangle - 1$$

Mentre per calcolare la sua inversa Ar^{-1}, ovvero a partire da un numero n decodificare l'istruzione:

- Se $n \bmod 3 = 0$:
  - È un'istruzione di primo tipo
  - $n = 3k$

  $\Rightarrow \quad R_{\frac{n}{3}} \leftarrow R_{\frac{n}{3}} + 1$
- Se $n \bmod 3 = 1$:
  - È un'istruzione di secondo tipo
  - $n = 3k + 1$

  $\Rightarrow \quad R_{\frac{n-1}{3}} \leftarrow R_{\frac{n-1}{3}} \dot{-} 1$
- Se $n \bmod 3 = 2$:
  - È un'istruzione di terzo tipo
  - $n = 3\langle k, m\rangle - 1$

  $\Rightarrow \quad \texttt{IF } R_{\sin(\frac{n+1}{3})} = 0 \texttt{ THEN GOTO } \text{des}(\frac{n+1}{3})$

Potenza computazionale del sistema RAM

$$F(\text{RAM}) = \{f \in \mathbb{N}_\perp^{\mathbb{N}} : \exists P \in \text{PROG} : \varphi_P = f\}$$
$$= \{\varphi_P : P \in \text{PROG}\} \subset \mathbb{N}_\perp^{\mathbb{N}}$$

Vista la possibilità di rappresentare un programma con un numero si ha:

$$= \{\varphi_i : i \in \mathbb{N}\}$$

Dove φ_i è la semantica del programma la cui codifica è i.

Conclusioni

Nelle ultime sezioni si è mostrata una biezione tra programmi RAM (PROG) e numeri ($\mathbb{N}$):

- Da programmi a numeri: $cod(P) = \langle Ar(\text{Istr}_1), Ar(\text{Istr}_2), \ldots, Ar(\text{Istr}_m)\rangle$
- Da numeri a programmi: si decodifichi la lista di numeri e si applichi su ogni numero Ar^{-1}

Per quanto riguarda i programmi RAM, **si può quindi affermare che**:

$$F(\text{RAM}) \sim \mathbb{N} \nsim \mathbb{N}_\perp^{\mathbb{N}}$$

e quindi esistono problemi non risolubili automaticamente da una macchina RAM.

2.6.2 Sistema di calcolo WHILE

Per mostrare se il concetto di calcolabilità è legato al sistema di calcolo o meno, se ne vedrà uno più sofisticato: quello WHILE.

Macchina WHILE

La macchina WHILE è più semplice di quella RAM; è formata infatti da un'unica memoria con un numero finito di registri:

$$\text{Memoria: } x_0, x_1, x_2, \ldots, x_{20}$$

- x_i è un generico registro di memoria detto variabile:
  - x_0 è la variabile specifica dove verrà deposto l'output del programma
  - x_1 è la variabile specifica dove verrà letto l'input del programma
  - Ci sono 21 variabili
- Non c'è un Program Counter

Linguaggio WHILE

La sintassi del linguaggio WHILE è induttiva; le istruzioni, dette comandi, sono:

- Comando di assegnamento:
  1. $x_k := 0$
  2. $x_k := x_j + 1$
  3. $x_k := x_j \dot{-} 1$
- Comando WHILE:
  1. `while` $x_k \neq 0$ `do` C
- Comando composto:
  1. `begin` C_1; C_2; $\ldots$; C_m; `end`

dove C_i è un comando WHILE.

Un programma WHILE è un comando composto.

Semantica programma WHILE

La semantica di un programma W è la funzione

$$\psi_W : \mathbb{N} \to \mathbb{N}_\perp$$

Esempio di programma

$W \equiv$
```
begin
    x2 := x1 + 1;
    x2 := x2 ∸ 1;
    while x1 ≠ 0 do
        begin
            x0 := x0 + 1;
            x1 := x1 ∸ 1;
        end
    while x2 ≠ 0 do
        begin
            x0 := x0 + 1;
            x2 := x2 ∸ 1;
        end
end
```
$\psi_W(n) = 2n$

W-PROG e induzione

Sia W-PROG l'insieme dei programmi WHILE. La sua definizione è induttiva; per dimostrare una proprietà P su W-PROG sarà quindi naturale farlo tramite induzione.

Stato

Siccome la macchina WHILE è composta solo da 21 variabili, uno stato della macchina è una lista di 21 numeri:

$$\text{STATO} = (c_0, c_1, \ldots, c_{20})$$
$$c_i = \text{contenuto di } x_i$$
$$\text{W-STATI} = \mathbb{N}^{21}$$
$$\underline{x} \in \mathbb{N}^{21}$$

La funzione $W\text{-}in : \mathbb{N} \to \mathbb{N}^{21}$ restituisce lo stato iniziale del programma W:

$$W\text{-}in(n) = (0, n, 0, \ldots, 0)$$

Funzione stato prossimo

Si definisca la funzione stato prossimo e quindi la semantica operazionale:

$$[\![\ ]\!](\ ) : \text{W-COM} \times \text{W-STATI} \to \text{W-STATI}_\perp$$

dato un comando C e uno stato $\underline{x}$:

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

dove $\underline{y}$ è lo stato prossimo di $\underline{x}$ a seguito dell'esecuzione del comando C.

Si veda ora la definizione induttiva della semantica operazionale:

- (BASE) Assegnamenti
  - $[\![x_k := 0]\!](\underline{x}) = \underline{y}$ con $y_i = \begin{cases} x_i & i \neq k \\ 0 & i = k \end{cases}$
  - $[\![x_k := x_j \pm 1]\!](\underline{x}) = \underline{y}$ con $y_i = \begin{cases} x_i & i \neq k \\ x_j \pm 1 & i = k \end{cases}$ (con $\pm$ si intende + o $\dot{-}$)
- (PASSO) Comando composto
  - $[\![\texttt{begin } C_1 \texttt{ ; } \ldots \texttt{ ; } C_m \texttt{ end}]\!](\underline{x}) = [\![C_m]\!](\ldots([\![C_2]\!]([\![C_1]\!](\underline{x})))\ldots)$
    $= [\![C_m]\!] \circ \ldots \circ [\![C_1]\!](\underline{x})$
    $= \underline{y}$
- (PASSO) Comando WHILE
  - $[\![\texttt{while } x_n \neq 0 \texttt{ do } C]\!](\underline{x}) = [\![C]\!](\ldots([\![C]\!]([\![C]\!](\underline{x})))\ldots) = \underline{y}$
    $= \begin{cases} [\![C]\!]^e(\underline{x}) & e = \mu t(\text{componente } k \text{ di } [\![C]\!]^t(\underline{x}) = 0) \\ \perp & \text{altrimenti (se non esiste } e) \end{cases}$

    dove $\mu t(\text{condizione}) = \min_t\{\text{condizione vera}\}$

Si può quindi definire la semantica di un programma W:

$$\psi_W(n) = \text{Pro}(0, [\![W]\!](W\text{-}in(n)))$$

dove Pro è la funzione proiezione:

$$\text{Pro}(i, (x_0, x_1, \ldots, x_m)) = x_i$$

Potenza computazionale del sistema WHILE

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

2.6.3 Traduzione

Dati i sistemi di calcolo $\mathscr{C}_1$ e $\mathscr{C}_2$, una traduzione da $\mathscr{C}_1$ a $\mathscr{C}_2$ è una funzione

$$T : \mathscr{C}_1\text{-PROG} \to \mathscr{C}_2\text{-PROG}$$

tale che T sia:

1. Programmabile: è programmabile effettivamente
2. Completa: traduce ogni programma
3. Corretta: mantiene la semantica: $\forall P \in \mathscr{C}_1\text{-PROG} \quad \varphi_{T(P)} = \psi_P$

2.6.4 Confronto tra sistemi di calcolo

Si prendano due sistemi di calcolo $\mathscr{C}_1$ e $\mathscr{C}_2$ con le rispettive potenze computazionali:

$$F(\mathscr{C}_1) = \{\varphi_{P_1} : P_1 \in \mathscr{C}_1\text{-PROG}\}$$

$$F(\mathscr{C}_2) = \{\psi_{P_2} : P_2 \in \mathscr{C}_2\text{-PROG}\}$$

Si vuole mostrare che $F(\mathscr{C}_1) \subseteq F(\mathscr{C}_2)$, ovvero che $\mathscr{C}_1$ non è più potente di $\mathscr{C}_2$; per farlo bisogna dimostrare che:

$$\forall f \in F(\mathscr{C}_1) \quad f \in F(\mathscr{C}_2)$$

$$\Vert\vert$$

$$\exists P_1 \in \mathscr{C}_1\text{-PROG} : f = \varphi_{P_1} \;\Rightarrow\; \exists P_2 \in \mathscr{C}_2\text{-PROG} : f = \psi_{P_2}$$

In altre parole, per ogni programma di $\mathscr{C}_1$ ne esiste uno equivalente in $\mathscr{C}_2$.

Teorema 3. *Se esiste una traduzione T da $\mathscr{C}_1$ a $\mathscr{C}_2$, allora $F(\mathscr{C}_1) \subseteq F(\mathscr{C}_2)$.*

Dimostrazione. Si prenda una funzione f tale che:

$$f \in F(\mathscr{C}_1)$$

Si ha quindi che esiste un programma in $\mathscr{C}_1$-PROG che può calcolare f:

$$\exists P \in \mathscr{C}_1\text{-PROG} : f = \psi_P \tag{5}$$

Si prenda la traduzione T da $\mathscr{C}_1$ a $\mathscr{C}_2$; vista la completezza di T si ha che:

$$T(P) \in \mathscr{C}_2\text{-PROG} \tag{6}$$

e vista la correttezza si ha che:

$$\varphi_{T(P)} = \psi_P \underset{(5)}{=} f \tag{7}$$

Si ha quindi che esiste un programma $T(P)$ in $\mathscr{C}_2$-PROG (6) la cui semantica è f (7). Si può concludere che:

$$f \in F(\mathscr{C}_2)$$

□

Quindi, per mostrare che un sistema di calcolo $\mathscr{C}_1$ non è più potente di un altro sistema $\mathscr{C}_2$, basterà trovare una traduzione da $\mathscr{C}_1$ a $\mathscr{C}_2$.

2.6.5 $F(\text{WHILE}) \subseteq F(\text{RAM})$

In questa sezione si mostrerà che il una macchina WHILE **non** è più potente di una macchina RAM. Per farlo, come suggerisce il teorema 3, verrà mostrata una traduzione da W-PROG a PROG; la funzione di traduzione viene detta compilatore:

$$Comp : \text{W-PROG} \to \text{PROG}$$

Per comodità si userà un linguaggio RAM etichettato, dove nell'istruzione di `GOTO` si useranno delle etichette e non il numero dell'istruzione. Questo non influenzerà in alcun modo la potenza del linguaggio.

Compilatore

Vista la natura induttiva di W-PROG, anche *Comp* verrà definita induttivamente:

1. (BASE) Compilazione degli assegnamenti:
   - $Comp(x_k := 0) =$

     ```
     LP:  IF R_k = 0 THEN GOTO EX
          R_k ← R_k ∸ 1
          IF R_21 = 0 THEN GOTO LP
     EX:  R_k ← R_k ∸ 1
     ```

   - $Comp(x_k := x_j + / \dot{-} 1) =$
     - Se $k = j$: $R_k \leftarrow +/ \dot{-} 1$
     - Se $k \neq j$:

       ```
       LP:  IF R_j = 0 THEN GOTO E1
            R_22 ← R_22 + 1
            R_j ← R_j ∸ 1
            IF R_21 = 0 THEN GOTO LP
       E1:  IF R_k = 0 THEN GOTO E2
            R_k ← R_k ∸ 1
            IF R_21 = 0 THEN GOTO E1
       E2:  IF R_22 = 0 THEN GOTO E3
            R_k ← R_k + 1
            R_j ← R_j + 1
            R_22 ← R_22 ∸ 1
            IF R_21 = 0 THEN GOTO E2
       E3:  R_k ← R_k + / ∸ 1
       ```

       } Salva R_j in R_{22}
       } Azzera R_k
       } Rigenera R_j e R_k da R_{22}

2. (PASSO) Per ipotesi induttiva si assumi data la compilazione di un comando C:
   (a) Compilazione di comando composto:
   - $Comp(\texttt{begin}\ C_1;\ C_2;\ \ldots;\ C_m;\ \texttt{end}) =$

     $Comp(C_1)$
     $Comp(C_2)$
     $\vdots$
     $Comp(C_m)$

   (b) Compilazione di comando WHILE:

- $Comp($`while` $x_k \neq 0$ `do` $C) =$

`LP:`	`IF` $R_k = 0$ `THEN GOTO EX`
	$Comp(C)$
	`IF` $R_{21} = 0$ `THEN GOTO LP`
`EX:`	$R_k \leftarrow R_k \dot{-} 1$

Conclusioni

La funzione $Comp$: W-PROG $\rightarrow$ PROG precedentemente definita soddisfa le tre condizioni di una traduzione:

1. È programmabile
2. Compila sempre
3. Mantiene la semantica

Si ha quindi che $Comp$ è una traduzione da W-PROG a PROG e che:

$$F(\text{WHILE}) \subseteq F(\text{RAM})$$

2.6.6 $F(\text{RAM}) \subseteq F(\text{WHILE})$

Ora che si è mostrato che una macchina WHILE non è più potente di una RAM, si mostrerà l'inverso.

Per farlo si userà il concetto di interprete.

Interprete

A differenza di un compilatore un interprete non produce nessun oggetto ma esegue direttamente tutte le istruzioni del programma. In altre parole, un interprete è un programma che prende in ingresso un altro programma P e un dato x e restituisce il risultato della semantica di P su x ovvero $\varphi_P(x)$.

Interprete WHILE di programmi RAM

Si definirà ora un interprete I_W scritto in WHILE di programmi RAM.

$$\begin{array}{r} P \in \text{PROG} \longrightarrow \\ x \in \mathbb{N} \longrightarrow \end{array} \boxed{I_W} \longrightarrow \varphi_P(x)$$

Problema: un programma WHILE prende in input solo numeri; non è in grado di leggere un programma RAM "puro". P

Soluzione: codifico P in un numero n.

$$\begin{array}{r} cod(P) = n \longrightarrow \\ x \in \mathbb{N} \longrightarrow \end{array} \boxed{I_W} \longrightarrow \varphi_n(x) = \varphi_P(x)$$

Problema: un programma WHILE prende in input solo un numero.

Soluzione: aggrego n e x tramite la funzione coppia di Cantor.

$$\langle x, n \rangle \longrightarrow \boxed{I_W} \longrightarrow \varphi_n(x) = \varphi_P(x)$$

La semantica dell'interprete I_W è:

$$\forall x, n \in \mathbb{N} : \psi_{I_W}(\langle x, n \rangle) = \varphi_n(x) = \psi_P(x)$$

Per questioni di comodità si usera una variante di WHILE in cui si potranno definire delle macro, ovvero delle chiamate a funzioni comunque definibili in WHILE puro; alcuni esempi di macro sono:

- $x_k := x_j + x_s$
- $x_k := \langle x_j, x_s \rangle$ coppia di Cantor
- $x_k := \langle a_1, a_2, \ldots, a_N \rangle$ "lista di Cantor"
- $x_k := \text{Pro}(x_j, x_s)$ estrae dalla lista codificata x_s il x_j-esimo elemento
- $x_k := \text{incr}(x_j, x_s)$ codifica la lista x_s in cui il x_j-esimo elemento viene incrementato di 1
- $x_k := \text{decr}(x_j, x_s)$ codifica la lista x_s in cui il x_j-esimo elemento viene decrementato di 1
- $x_k := \text{sin}(x_j)$ estrae l'elemento sinistro dalla codifica di Cantor x_j
- $x_k := \text{des}(x_j)$ estrae l'elemento destro dalla codifica di Cantor x_j
- `if...then...else`

Stato della macchina RAM nell'interprete

L'interprete I_W per eseguire il programma P ricrea nelle sue variabili lo stato della macchina RAM in cui verrebbe eseguito P.

Problema: la macchina RAM ha infiniti registri mentra quella WHILE ne ha ventuno.

Soluzione: un programma P userà sempre un numero finito di registri il cui contenuto può essere racchiuso in una lista di valori $a_0, a_1, \ldots, a_n$; la soluzione consiste nel raggruppare tutti i valori dei registri tramite Cantor:

$$\langle a_1, a_2, \ldots, a_n \rangle$$

e salvarne la codifica in un unica variabile.

I_W salva lo stato della macchina RAM nel seguente modo:

- $x_0 \leftarrow \langle R_0, \ldots, R_n \rangle$ stato della memoria RAM del programma P con $cod(P) = n$
- $x_1 \leftarrow L$ program counter
- $x_2 \leftarrow x$ dato di input del programma P
- $x_3 \leftarrow n$ lista di istruzioni che formano P
- $x_4 \leftarrow$ codice dell'istruzione da eseguire

Si noti come il valore di n (primo punto), ovvero dell'indice del registro più alto presente nel programma P, sia direttamente il valore della codifica di P. Questo perchè n vuole essere un limite superiore dell'indice più alto presente in P e non un valore esatto.

Implementazione

Verrà mostrata ora l'implementazione dell'interprete I_W scritto in (macro-)WHILE di un programma RAM P:

$$\psi_{I_W}(\langle x, n \rangle) = \varphi_n(x)$$

```
// Inizialmente l'input si trova in x_1
x_2 := sin(x_1);
x_3 := des(x_1);
x_0 := ⟨0, x_2, ..., 0⟩;
x_1 := 1;
while (x_1 ≠ 0) do                                  // se x_1 = 0 allora STOP
    if (x_1 > length(x_3)) then                     // supero l'ultima istruzione
        x_1 := 0;                                                  // STOP
    else
        x_4 := Pro(x_1, x_3);                       // estraggo istruzione corrente
        if (x_4 mod 3 = 0) then                                    // R_k ← R_k + 1
            x_5 := x_4/3;                                                  // k
            x_0 := incr(x_5, x_0);
            x_1 := x_1 + 1;
        if (x_4 mod 3 = 1) then                                    // R_k ← R_k ∸ 1
            x_5 := (x_4 − 1)/3;                                            // k
            x_0 := decr(x_5, x_0);
            x_1 := x_1 + 1;
        if (x_4 mod 3 = 2) then                         // IF R_k = 0 THEN GOTO m
            x_5 := sin((x_4 + 1)/3);                                       // k
            x_6 := des((x_4 + 1)/3);                                       // m
            if (Pro(x_5, x_0) = 0) then                 // verifico R_k = 0
                x_1 := x_6;
            else
                x_1 := x_1 + 1;
x_0 = sin(x_0);                                 // metto in x_0 il risultato φ_n(x)
```

Conclusioni

L'esistenza di I_W permette la costruzione immediata di un compilatore:

$$Comp : \text{PROG} \to \text{W-PROG}$$

$$Comp(P) = \boxed{\begin{array}{l} n \leftarrow cod(P); \\ x_1 := \langle x_1, n \rangle; \\ I_W; \end{array}}$$

Comp non fa altro che mettere in x_1 il dato x (già presente nella variabile x_1) e il programma codificato in n aggregati tramite $\langle x, n \rangle$ e far partire l'interprete I_W che restituirà in x_0 il risultato della computazione.

Si è quindi appena dimostrata l'esistenza di una traduzione da RAM a WHILE che implica che:

$$F(\text{RAM}) \subseteq F(\text{WHILE})$$

2.6.7 Teorema di Böhm-Jacopini

I risultati delle sezioni 2.6.5 e 2.6.6 permettono di dire che:

$$\begin{array}{l} F(\text{RAM}) \subseteq F(\text{WHILE}) \\ F(\text{RAM}) \supseteq F(\text{WHILE}) \end{array} \quad \Rightarrow \quad F(\text{RAM}) = F(\text{WHILE})$$

Teorema 4 (Teorema di Böhm-Jacopini)**.** *Per ogni programma con il comando `GOTO` (RAM) ne esiste uno equivalente in linguaggio strutturato (WHILE)*

Questo teorema, enunciato nel 1966 da due informatici italiani, rappresenta un risultato cruciale dell'informatica: mostra infatti che:

1. Il comando di `GOTO` ha solo nati negativi e non è necessario;
2. La programmazione a basso livello può essere sostituita da quella ad alto livello.