Corso di Informatica Teorica
Anno accademico 2023/24

Note di Mauro Tellaroli

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.

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;

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: $\forall a, b, c \in A \quad a \ R \ b \ \wedge \ b \ R \ c \Rightarrow a \ R \ c$

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.

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

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 $\mathbb{N}$.

Chiusura di insiemi rispetto ad operazioni

Dato un insieme U, si definisce operazione su U una qualunque funzione

$$op : \underbrace{U \times \cdots \times U}_{k = \text{arietà}} \to U$$

L'insieme $A \subseteq U$ è chiuso rispetto all'operazione $op : U^k \to U$ se:

$$\forall a_1, \dots, a_k \in A : op(a_1, \dots, a_k) \in A$$

Sia $\Omega = \{op_1, \dots, op_t\}$ un insieme di operazioni su U, allora $A \subseteq U$ è chiuso rispetto a Ω se A è chiuso per ogni $op \in \Omega$.

Esempi

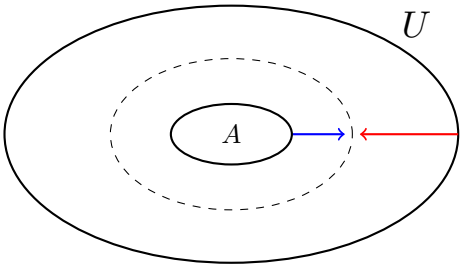
Sia $\Omega = \{+, *\}$:

- PARI $\subseteq \mathbb{N}$ è chiuso per Ω? Sì, la somma o la moltiplicazione di numeri parì è un numero pari.
- DISPARI $\subseteq \mathbb{N}$ è chiuso per Ω? No, un controesempio è $3 + 7 = 10 \notin$ DISPARI.

Chiusura di un insieme

Sia $A \subseteq U$ e $op : U^k \to U$. Si vuole trovare il più piccolo sottoinsieme di U che:

1. Contiene A
2. È chiuso per op



- Sicuramente U soddisfa le due condizioni ma si sta cercando l'insieme più piccolo;
- Se A è chiuso per op allora l'insieme cercato sarebbe A stesso;
- Se A non è chiuso per op allora devo allargare la ricerca ad un insieme più grande.

Teorema 1 (Chiusura di un insieme)**.** *Sia $A \subseteq U$ e $op : U^k \to U$. Il più piccolo sottoinsieme di U contenente A e chiuso rispetto a op si ottiene calcolando la chiusura di A rispetto a op, ovvero 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}

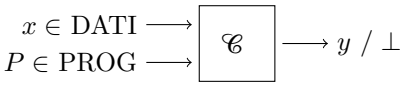
Una versione "più operativa" per trovare A^{op} è:

1. $A^{op} = A$
2. Calcola $op(a_1, \ldots, a_k) = r$ su una k-tupla di A
3. Se $r \notin A$ allora $A^{op} = A^{op} \cup \{r\}$
4. Ripeti il punto 2 con un'altra k-tupla fino ad averle provate tutte

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:

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

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 Insiemi isomorfi

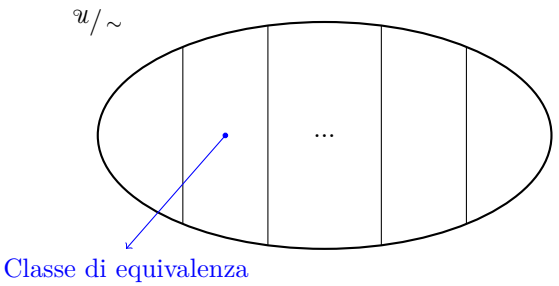
Due insiemi A e B sono **isomorfi** (o equinumerosi) se esiste una funzione biettiva tra essi. Formalmente si indica con:

$$A \sim B$$

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.

2.3.2 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

2.3.3 Insiemi non numerabili

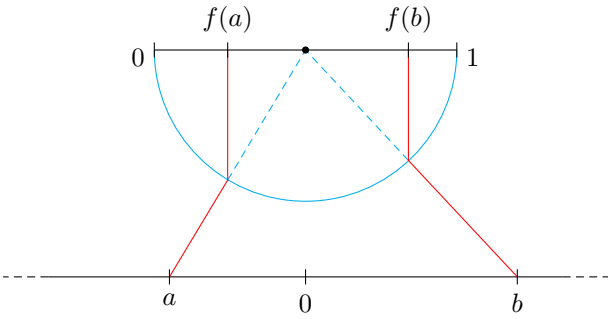
Gli insiemi non numerabili sono insiemi a cardinalità infinita ma non listabili come $\mathbb{N}$ (sono "più fitti"). Il re di questi insiemi è $\mathbb{R}$.

Teorema 2. *$\mathbb{R}$ 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 assumendo che $\mathbb{N} \sim (0,1)$; questo vorrebbe dire che tutti i numeri compresi tra 0 e 1 sono numerabili. Si elenchino tutti i numeri associandoli a un numero naturale:

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

   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.

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

Sfruttando la transitività di $\sim$ 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 $2^{\mathbb{N}} = \{\text{sottoinsiemi di } \mathbb{N}\}$

Dimostrazione. Si assuma, per assurdo, che $2^{\mathbb{N}} \sim \mathbb{N}$. Si descriva un elemento di $2^{\mathbb{N}}$ con il suo vettore caratteristico:

$\mathbb{N}$	0	1	2	3	4	5	...
A	0	1	1	0	1	0	...

$A = \{1, 2, 4, \dots\}$

Si elenchino tutti gli elementi di $2^{\mathbb{N}}$ tramite i loro vettori caratteristici:

$$\begin{array}{lcccccc} 0 & \mapsto & b_{00} & b_{01} & b_{02} & \dots \\ 1 & \mapsto & b_{10} & b_{11} & b_{12} & \dots \\ 2 & \mapsto & b_{20} & b_{21} & b_{22} & \dots \\ \vdots & & \vdots & \vdots & \vdots & \ddots \end{array}$$

Si costruisca il sottoinsieme $S \in 2^{\mathbb{N}}$ definito dal seguente vettore caratteristico:

$$S = \bar{b_{00}} \quad \bar{b_{11}} \quad \bar{b_{22}} \quad \bar{b_{33}} \quad \dots \qquad (\bar{b} = \neg b)$$

S non appare nella lista:

- Differisce dal primo sottoinsieme perchè $\bar{b_{00}} \neq b_{00}$;
- Differisce dal primo sottoinsieme perchè $\bar{b_{11}} \neq b_{11}$;
- ...
- Differisce da qualunque elemento nella diagonale.

Si è quindi trovato l'assurdo; ne segue che $2^{\mathbb{N}} \nsim \mathbb{N}$. □

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

  Dimostrazione. Si assuma, per assurdo, che $\mathbb{N}_\perp^{\mathbb{N}} \sim \mathbb{N}$: si possono quindi listare i suoi elementi:

  $$\mathbb{N}_\perp^{\mathbb{N}} = \{f_0, f_1, f_2, \dots\}$$

  Più nel dettaglio:

	0	1	2	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

  Si definisca una funzione $g : \mathbb{N} \to \mathbb{N}_\perp$:

  $$g(n) = \begin{cases} 1 & f_n(n) = \perp \\ f_n(n) + 1 & f_n(n)\downarrow \end{cases}$$

  Sicuramente $g \in \mathbb{N}_\perp^{\mathbb{N}}$; deve quindi apparire nella lista per qualche $k \in \mathbb{N}$.

  Non esiste nessun k tale che $f_k = g$; se si valutano le due funzioni nel loro valore k si vede che:

  $$g(k) = \begin{cases} 1 \neq f_k(k) & f_k(k) = \perp \\ f_k(k) + 1 \neq f_k(k) & f_k(k)\downarrow \end{cases}$$

  g non è mai uguale a un qualsiasi f_k visto che differisce sempre in un valore diagonale. Si è quindi trovato l'assurdo; ne segue che $\mathbb{N}_\perp^{\mathbb{N}} \nsim \mathbb{N}$. □

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:

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

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 3. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+$

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	↗				

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

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

$x\backslash y$	0	⋯	y
⋮			
x			$\langle x, y \rangle$
⋮			
$x+y$	$\langle x+y, 0 \rangle$		

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:

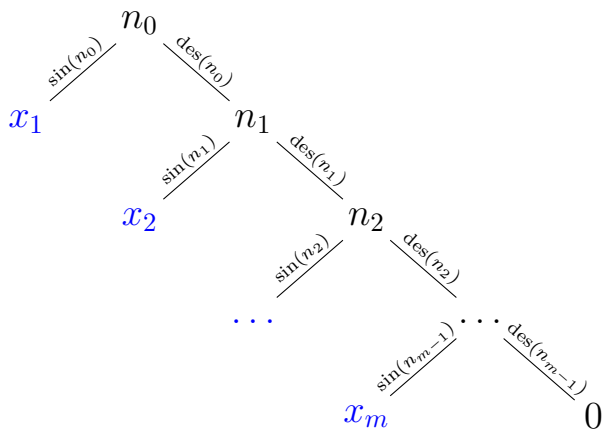
$$x_1, x_2, \ldots, x_m \to \langle x_1, x_2, \ldots, x_m\rangle$$

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:



Se si conosce il numero di elementi della sequenza di numeri, non si usa una lista ma un **array**:

$$[x_1, \ldots, x_n] = [x_1, [x_2, [\ldots, [x_{n-1}, x_n]] \ldots]]$$

Conoscendo già la lunghezza non c'è bisogno dello 0 di fine lista.

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.

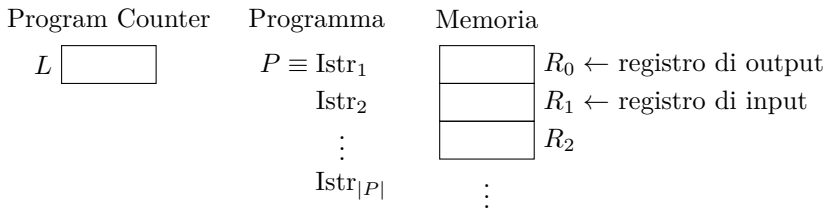
Può però sorgere spontaneo un dubbio: il sistema RAM non è troppo semplice per rappresentare tutti i sistemi di calcolo? Se il sistema RAM non fosse in grado di risolvere certi problemi, magari altri sistemi più complessi lo sarebbero.

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:

- $F(\text{WHILE}) \neq F(RAM) \Rightarrow$ la computabilità dipende dallo strumento usato;
- $F(\text{WHILE}) = F(RAM) \Rightarrow$ 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:

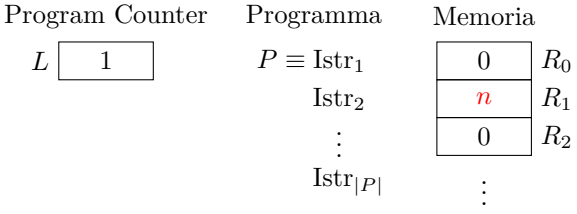
1. $R_k \leftarrow R_k + 1$
2. $R_k \leftarrow R_k \dot{-} 1$
3. `IF` $R_k = 0$ `THEN GOTO` m

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

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 possibilità 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.

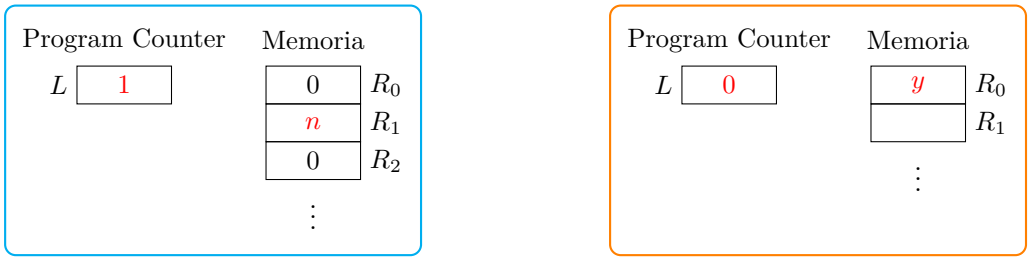
L'esecuzione di un programma è una sequenza di stati della macchina, dove uno stato descrive precisamente l'attuale situazione della macchina. Ogni istruzione fa passare la macchina da uno stato ad un altro:

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

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

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

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



La semantica di P è:

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

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

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:

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

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

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:

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

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

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

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 $\mathcal{S}'(L) = \mathcal{S}(L) + 1$

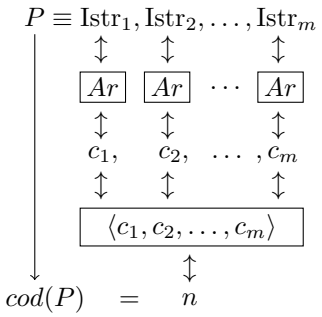
Esempio di programma

$P \equiv$ `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 4. *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 4, 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      } Salva R_j in R_22
       E1:  IF R_k = 0 THEN GOTO E2
            R_k ← R_k ∸ 1
            IF R_21 = 0 THEN GOTO E1      } Azzera R_k
       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      } Rigenera R_j e R_k da R_22
       E3:  R_k ← R_k + / ∸ 1
       ```

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

     $$\begin{array}{|l|}\hline Comp(C_1) \\ Comp(C_2) \\ \vdots \\ Comp(C_m) \\ \hline \end{array}$$

   (b) Compilazione di comando WHILE:

- $Comp(\texttt{while } x_k \neq 0 \texttt{ do } C) =$

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:

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

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
- $x_4 \leftarrow$ codifica 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 5 (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.

Concludendo, si è anche dimostrato che:

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

Esistono quindi funzioni non computabili da entrambi i sistemi.

2.6.8 Interprete universale

Si prenda il compilatore $Comp$: W-PROG $\to$ PROG e lo si utilizzi con I_W:

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

$$U = Comp(I_W) \in \text{PROG}$$

U è detto interprete universale ed è un programma RAM in grado di simulare qualsiasi altro programma RAM. La sua semantica è:

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

2.7 Calcolabilità

Nelle sezioni precedenti si è visto come due sistemi di calcolo molto diversi tra loro abbiano la stessa potenza computazionale numerabile:

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

e che quindi esistono delle funzioni non calcolabili da questi sistemi. Torna però qui la questione: "E se esistesse un altro sistema di calcolo più potente?"

Bisognerà affrontare il problema a prescindere dalla macchina usata, definendo la calcolabilità in termini più matematici e "lontani" dall'informatica.

2.7.1 Una prima idea di calcolabilità

Per introdurre la definizione "teorica" di calcolabilità si introducano due insiemi:

- ELEM: insieme di tre funzioni molto semplici e banalmente implementabili;
- Ω: insieme di operazioni su funzioni banalmente implementabili;

L'idea è che applicando un operazione facilmente calcolabile di Ω ad una funzione facilmente calcolabile di ELEM ottengo un'altra funzione facilmente calcolabile:

$$\text{ELEM}^{\Omega} = \mathscr{P}$$

$\mathcal{P}$ è la classe delle funzioni ricorsive parziali e vuole essere una prima idea di classe di funzioni calcolabili. **Bisognerà ora capire quali funzioni compongono ELEM e quali operatori compongono Ω.**

ELEM è composto da tre funzioni:

$$\begin{aligned} \text{ELEM} = \quad \{ \text{ successore:} \quad & s(x) = x + 1 \ , && x \in \mathbb{N} \\ \text{zero:} \quad & O^n(x_1, \ldots, x_n) = 0 \ , && x_i \in \mathbb{N} \\ \text{proiettori:} \quad & \text{pro}_k^n(x_1, \ldots, x_n) = x_k \ , && x_i \in \mathbb{N} \ \} \end{aligned}$$

Le funzioni appena mostrare di ELEM devono essere incluse in un'idea di calcolabile: sono proprio banali e semplici. Tuttavia ci saranno anche altre funzioni sicuramente calcolabili; basti pensare che $f(x) = x + 2$ non è presente in ELEM ma è certamente calcolabile.

Si deve quindi ampliare ELEM mantenendo sempre un'idea intuitiva di calcolabile: si introdurranno gli operatori di composizione di funzioni e di ricorsione primitiva.

Operatore di composizione di funzioni

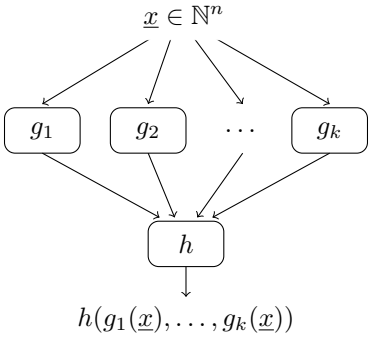
Siano:

$$\begin{aligned} h &: \mathbb{N}^k \to \mathbb{N} \\ g_1, \ldots, g_k &: \mathbb{N}^n \to \mathbb{N} \\ \underline{x} &\in \mathbb{N}^n \end{aligned}$$

Si definisce la composizione di funzioni come:

$$\text{COMP}(h, g_1, \ldots, g_k) : \ \mathbb{N}^n \to \mathbb{N}$$

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



Si ampli ELEM chiudendolo rispetto a COMP:

$$\text{ELEM}^{\text{COMP}}$$

ELEM viene ampliato; si può notare infatti che:

- $f(x) = x + 2 \notin \text{ELEM}$
- $f(x) = x + 2 \in \text{ELEM}^{\text{COMP}}$ infatti $f(x) = \text{COMP}(s, s)(x) = s(s(x)) = s(x + 1) = x + 2$

Tuttavia questo ampliamento non basta: $\text{ELEM}^{\text{COMP}}$ infatti non contiene funzioni banali come $\text{somma}(x, y) = x + y \notin \text{ELEM}^{\text{COMP}}$.

Operatore di ricorsione primitiva

Quest'operatore permette di definire funzioni ricorsive come:

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

Siano:

$$
\begin{aligned}
g:\ \mathbb{N}^n \to \mathbb{N} \qquad & \text{funzione da applicare alla base ricorsiva} \\
h:\ \mathbb{N}^{n+2} \to \mathbb{N} \qquad & \text{funzione da applicare nel passo ricorsivo} \\
g(\underline{x})\ ,\ h(z,y,\underline{x}) \text{ con } \underline{x} \in \mathbb{N}^n &
\end{aligned}
$$

Si definisce l'operatore di ricorsione primitiva:

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

Si noti che nel passo ricorsivo $h(f(\underline{x},y-1),y-1,\underline{x})$ non è detto debbano essere gli ultimi due argomenti della funzione h.

2.7.2 Funzioni ricorsive primitive

Si ampli $\mathrm{ELEM}^{\mathrm{COMP}}$ chiudendolo rispetto a RP:

$$
\mathrm{ELEM}^{\mathrm{COMP,RP}} = \mathrm{RICPRIM}
$$

$\mathrm{ELEM}^{\mathrm{COMP,RP}}$ è formato dalle funzioni ricorsive primitive.

Alcuni esempi di funzioni ricorsive primitive sono:

$$
\begin{aligned}
\mathrm{somma}(x,y) &= \mathrm{RP}(s,\mathrm{pro}_1^1) \\
&= \begin{cases} \mathrm{pro}_1^1(x) & y = 0 \\ s(\mathrm{somma}(x,y-1),y-1,x) & y > 0 \end{cases} \\
&= \begin{cases} x & y = 0 \\ s(\mathrm{somma}(x,y-1)) & y > 0 \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{prodotto}(x,y) &= \mathrm{RP}(\mathrm{somma},O^1) \\
&= \begin{cases} O^1(x) & y = 0 \\ \mathrm{somma}(\mathrm{prodotto}(x,y-1),y-1,x) & y > 0 \end{cases} \\
&= \begin{cases} 0 & y = 0 \\ \mathrm{somma}(\mathrm{prodotto}(x,y-1),x) & y > 0 \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{predecessore}(x) &= P(x) = \begin{cases} 0 & x = 0 \\ x-1 & x > 0 \end{cases} \\
\dot{-}(x,y) &= x \dot{-} y = \mathrm{RP}(P,\mathrm{pro}_1^1) \\
&= \begin{cases} \mathrm{pro}_1^1(x) & y = 0 \\ P(\dot{-}(x,y-1),y-1,x) & y > 0 \end{cases} \\
&= \begin{cases} x & y = 0 \\ P(\dot{-}(x,y-1)) & y > 0 \end{cases}
\end{aligned}
$$

Si noti, come anticipato, che alcuni argomenti del passo ricorsivo possono non essere usati dalla funzione h

RICPRIM vs F(WHILE)

RICPRIM contiene molte funzioni; si mostrerà ora che tutte le funzioni ricorsive primitive possono essere calcolate da una macchina WHILE, ovvero:

$$
\mathrm{RICPRIM} \subseteq F(\mathrm{WHILE})
$$

La dimostrazione è fatta tramite induzione strutturale a partire dalla definizione di RICPRIM = $\text{ELEM}^{\text{COMP,RP}}$:

1. (BASE) Le funzioni in ELEM sono in RICPRIM
2. (PASSO) $h, g_1, \ldots, g_k \in \text{RICPRIM} \Rightarrow \text{COMP}(h, g_1, \ldots, g_k) \in \text{RICPRIM}$
3. (PASSO) $g, h \in \text{RICPRIM} \Rightarrow \text{RP}(h, g) \in \text{RICPRIM}$
4. Nient'altro è in RICPRIM

Per dimostrare che RICPRIM $\subseteq F(\text{WHILE})$:

1. Dimostro che ELEM $\subseteq F(\text{WHILE})$, ovvero mostro per ogni funzione $f \in \text{ELEM}$ un programma $W \in W\text{-PROG}$ tale che $\psi_W = f$:
   - $s(x) = x + 1$:
     ```
     begin
         x0 := x1+1;
     end
     ```
   - $O^n(x_1, \ldots, x_n) = 0$:
     ```
     begin
         x0 := 0;
     end
     ```
   - $\text{pro}_k^n(x_1, \ldots, x_n) = x_k$:
     ```
     begin
         x0 := Pro(k, x1);
     end
     ```
2. Assumo che $h, g_1, \ldots, g_k \in F(\text{WHILE})$ ovvero che esistono dei programmi $H, G_1, \ldots, G_k \in W\text{-PROG}$ tali che:
   $$\psi_H = h \ , \ \psi_{G_1} = g_1 \ , \ \ldots \ , \ \psi_{G_k} = g_k$$
   e dimostro che $\text{COMP}(h, g_1, \ldots, g_k) \in F(\text{WHILE})$ ovvero mostro un programma WHILE che calcola:
   $$\text{COMP}(h, g_1, \ldots, g_k)(\underline{x}) = h(g_1(\underline{x}), \ldots, g_k(\underline{x}))$$
   ```
   begin                                   // x1 ← x = ⟨a1, ..., an⟩
       x0 := G1(x1);
       x0 := [x0, G2(x1)];                 // Accoda G2(x) all'array x0
       ⋮
       x0 := [x0, Gk(x1)];                 // x0 := [G1(x), ..., Gk(x)]
       x0 := H(x0);                        // x0 := H(G1(x), ..., Gk(x))
   end
   ```
3. Assumo che $g, h \in F(\text{WHILE})$ ovvero che esistono dei programmi $H, G \in W\text{-PROG}$ tali che:
   $$\psi_H = h \ , \ \psi_G = g$$

e dimostro che $\text{RP}(h, g) \in F(\text{WHILE})$ ovvero mostro un programma WHILE che calcola:

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

```
// Parto dalla base e salgo
// x_1 ← ⟨x, y⟩
begin
   t := G(x);
   k := 1;                          // k mi dice a che punto sono
   while k ≤ y do                     // finchè k non raggiunge y
      begin
         t := H(t, k − 1, x);
         k := k + 1;
      end;
end
```

Dunque per induzione strutturale si è dimostrato che:

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

Tuttavia resta la domanda sull'inclusione appena dimostrata: è propria?

Per rispondere basta notare che **RICPRIM contiene solo funzioni totali**. La dimostrazione è semplice e si basa sulla definizione induttiva di RICPRIM:

- (BASE) Le funzioni di ELEM sono totali;
- (PASSO) La composizione di funzioni totali è una funzione totale;
- (PASSO) La ricorsione primitiva di funzioni totali è una funzione totale: si ha sempre un caso base e per definzione si parte da y e si decrementa di 1 fino ad arrivare a 0.

F(WHILE) invece contiene anche funzioni parziali grazie ai suoi loop che possono non terminare.

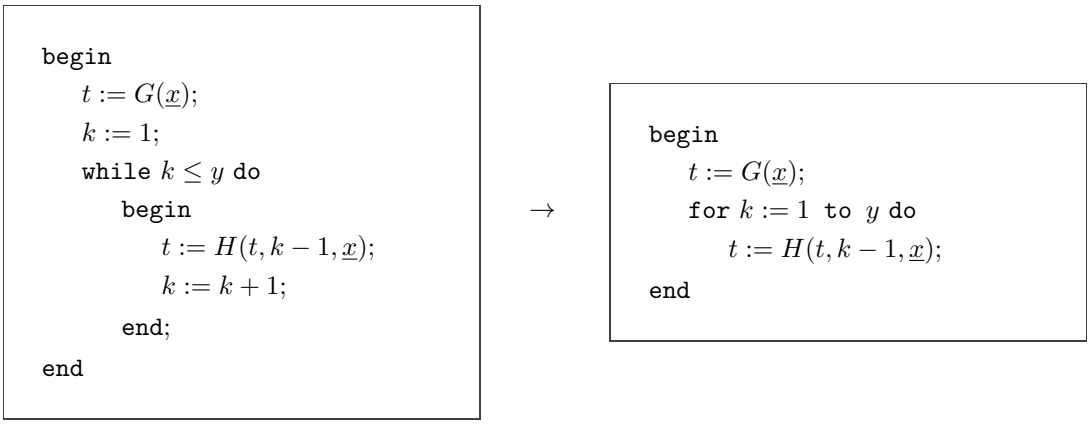
Si ha quindi che:

$$\text{RICPRIM} \subset \boldsymbol{F}(\text{WHILE})$$

RICPRIM quindi non basta a rappresentare il concetto di calcolabilità: bisogna ampliarlo introducendo una nuova operazione.

Sistema di calcolo FOR

Partendo dal linguaggio WHILE si può introdurre un nuovo linguaggio, identico a quello WHILE, ma che cambia l'istruzione di ciclo: viene usata l'istruzione, ben conosciuta, di ciclo `for`:



dove la variabile usata dal `for` non può essere toccata. Il linguaggio che ne deriva viene chiamato linguaggio FOR e si può dire che:

$$\text{RICPRIM} = F(\text{FOR}) \subset F(\text{WHILE})$$

2.7.3 Funzione di Ackermann

Come già visto, F(WHILE) contiene delle funzioni parziali che RICPRIM non ha. Queste funzioni parziali derivano dalla possibiliità del linguaggio WHILE di ottenere dei loop che non terminano. Sembra quindi che se si togliesse la possibilità di eseguire un loop infinito dal linguaggio WHILE otterremmo RICPRIM.

Si prenda il linguaggio WHILE privato della possibilità di eseguire loop infiniti:

$$\widetilde{F}(\text{WHILE}) = \{\psi_W : W \in W\text{-PROG} \ \wedge \ \psi_W \text{ è totale}\}$$

Si ha che:

$$\text{RICPRIM} \subseteq \widetilde{F}(\text{WHILE})$$

È un'inclusione propria? **La risposta è sì.**

Per dimostrarlo si deve mostrare una funzione calcolabile in WHILE che non è ricorsiva primitiva. Questa funzione è la funzione di Ackermann (1928):

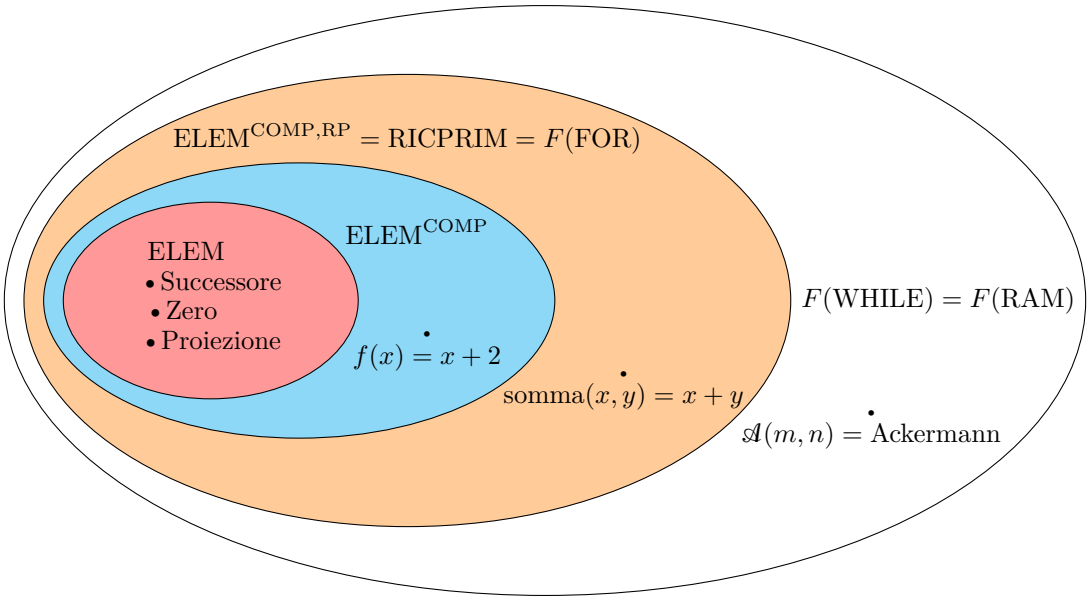
$$\mathscr{A}(m,n) = \begin{cases} n+1 & m = 0 \\ \mathscr{A}(m-1,1) & m > 0, n = 0 \\ \mathscr{A}(m-1, \mathscr{A}(m, n-1)) & m, n > 0 \end{cases}$$

Non si entrerà nel dettaglio ma basti sapere che $\mathscr{A}$ cresce troppo in fretta per poter essere ricorsiva primitiva.

Questo mostra che il sistema WHILE è più potente delle funzioni in RICPRIM anche escludendo le funzioni parziali.

2.7.4 RICPRIM non basta

La ricerca delle funzioni calcolabili ha portato fin'ora all'insieme delle funzioni ricorsive primitive RICPRIM che, però, non si è dimostrato all'altezza mostrando "alcune lacune" (Ackermann). Si dovrà quindi, ancora una volta, procedere con un ampliamento di RICPRIM.



Operatore di minimalizzazione di funzioni

Sia:

$$f : \mathbb{N}^{n+1} \to \mathbb{N}$$

$$f(\underline{x}, y) \text{ con } \underline{x} \in \mathbb{N}^n$$

si definisce operatore di minimalizzazione di una funzione:

$$\text{MIN}(f)(\underline{x}) = g(\underline{x}) = \begin{cases} y & f(\underline{x}, y) = 0 \ \wedge \ (\forall y' < y : f(\underline{x}, y')\downarrow \ \wedge \ f(\underline{x}, y') \neq 0) \\ \perp & \text{altrimenti} \end{cases}$$
$$= \mu y(f(\underline{x}, y) = 0)$$

A parole, MIN restituisce una funzione g la quale restituisce il valore di y più piccolo che, fissata $\underline{x}$, azzera $f(\underline{x}, y)$ o $\perp$ se non esiste un valore che azzera $f(\underline{x}, y)$.

Alcuni esempi dell'applicazione di MIN sono:

$\boldsymbol{f(x, y)}$	$\boldsymbol{\text{MIN}(f)(x) = g(x)}$
$x + y + 1$	$\perp$
$x \dot{-} y$	x
$y \dot{-} x$	0
$x \dot{-} y^2$	$\lceil \sqrt{x} \rceil$
$\left\lfloor \frac{x}{y} \right\rfloor$	$\perp$

2.7.5 Funzioni ricorsive parziali

Come già fatto precedentemente si ampli RICPRIM chiudendo RICPRIM = ELEM$^{\text{COMP,RP}}$ rispetto alla nuovo operazione introdotta MIN:

$$\text{ELEM}^{\text{COMP,RP,MIN}} = \mathcal{P}$$

Le funzioni di $\mathcal{P}$ sono dette funzioni ricorsive parziali.

Sicuramente $\mathcal{P}$ è più grande di RICPRIM vista la presenza di funzioni parziali. Ma che rapporto ha $\mathcal{P}$ con F(WHILE)?

Teorema 6. $\mathscr{P} \subseteq F(WHILE)$

Dimostrazione. Si usi l'induzione strutturale su $\mathscr{P} = \text{ELEM}^{\text{COMP,RP,MIN}}$, che può essere definito induttivamente:

1. Le funzioni di ELEM sono in $\mathscr{P}$
2. $h, g_1, \ldots, g_k \in \mathscr{P} \;\Rightarrow\; \text{COMP}(h, g_1, \ldots, g_k) \in \mathscr{P}$
3. $h, g \in \mathscr{P} \;\Rightarrow\; \text{RP}(h, g) \in \mathscr{P}$
4. $f \in \mathscr{P} \;\Rightarrow\; \text{MIN}(f) \in \mathscr{P}$
5. Null'altro è in $\mathscr{P}$

Per dimostrare che $\mathscr{P} \subseteq F(\text{WHILE})$:

1. Dimostro che ELEM $\subseteq F(\text{WHILE})$, ovvero mostro per ogni funzione $f \in$ ELEM un programma $W \in W\text{-PROG}$ tale che $\psi_W = f$: già visto nella sezione 2.7.2
2. Assumo che $h, g_1, \ldots, g_k \in F(\text{WHILE})$ ovvero che esistono dei programmi $H, G_1, \ldots, G_k \in W\text{-PROG}$ tali che:
$$\psi_H = h \;,\; \psi_{G_1} = g_1 \;,\; \ldots \;,\; \psi_{G_k} = g_k$$
e dimostro che $\text{COMP}(h, g_1, \ldots, g_k) \in F(\text{WHILE})$: già visto nella sezione 2.7.2.
3. Assumo che $g, h \in F(\text{WHILE})$ ovvero che esistono dei programmi $H, G \in W\text{-PROG}$ tali che:
$$\psi_H = h \;,\; \psi_G = g$$
e dimostro che $\text{RP}(h, g) \in F(\text{WHILE})$: già visto nella sezione 2.7.2.
4. Assumo che $f \in F(\text{WHILE})$ ovvero che esiste un programma $\mathcal{F} \in F(\text{WHILE})$ tale che $\psi_{\mathcal{F}} = f$ e dimostro che $\text{MIN}(f) \in F(\text{WHILE})$ ovvero mostro un programma WHILE che calcola:
$$\text{MIN}(f)(\underline{x}) = g(\underline{x}) = \begin{cases} \mu y(f(\underline{x}, y) = 0) & \\ \perp & \text{se non esiste tale } y \end{cases}$$

```
// x_1 ← x
begin
   y := 0;              // parto da 0
   while F(x, y) ≠ 0 do
      y := y + 1;       // salgo di 1
end
```

L'idea del programma è partire dal valore più basso che y può assumere in $\mathbb{N}$, ovvero 0, per poi salire di un'unita fino a trovare il primo valore che azzera $f(\underline{x}, y)$; se il valore non esiste o $F(\underline{x}, y)$ va in loop prima di trovarlo si avrà un loop che produce $\perp$ in output.

□

Il precedente teorema permette di dire che qualsiasi funzione ricorsiva parziale può essere calcolata da un programma WHILE. Si vedrà ora che per qualsiasi programma WHILE esiste una funzione ricorsiva parziale che è uguale alla semantica del programma ($F(\text{WHILE}) \subseteq \mathscr{P}$).

Teorema 7. $F(WHILE) \subseteq \mathscr{P}$

Dimostrazione. Se $F(\text{WHILE}) \subseteq \mathscr{P}$ allora:

$$\psi_W \in F(\text{WHILE}) \;\Rightarrow\; \psi_W \in \mathscr{P} = \text{ELEM}^{\text{COMP,RP,MIN}}$$

Si dovrà quindi dimostrare che per un qualsiasi programma W, la sua semantica ψ_W apparterrà a $\mathscr{P}$ e potrà essere espressa come composizione, ricorsione primitiva e minimalizzazione a partire da funzioni di ELEM.

$$\psi_W(x) = \text{Pro}_0^{21}([\![W]\!](W\text{-in}(x)))$$

- La funzione $[\![C]\!](\underline{x})$ restituisce un array di 21 elementi che rappresenta lo stato prossimo della macchina WHILE applicando il comando C allo stato attuale $\underline{x}$;
- La funzione W-in(n) restituisce lo stato iniziale della macchina WHILE su input n;
- Pro_0^{21} preleva l'output dal registro x_0.

ψ_W è la composizione di $\text{Pro}_0^{21} \in$ ELEM con la funzione stato prossimo $[\![W]\!](W\text{-in}(x))$. Si ha quindi che:

1. $\text{Pro}_0^{21} \in \text{ELEM} \Rightarrow \text{Pro}_0^{21} \in \mathscr{P}$
2. $\mathscr{P}$ è chiuso rispetto alla composizione

$$\Rightarrow \quad [\![C]\!](\underline{x}) \in \mathscr{P} \Rightarrow \psi_W \in \mathscr{P}$$

Se quindi si dimostra che $[\![C]\!](\underline{x}) \in \mathscr{P}$ allora anche la sua semantica ψ_W sarà una funzione ricorsiva parziale.

La funzione $[\![\,]\!]() : \mathbb{N}^{21} \to \mathbb{N}^{21}$ ha come codominio $\mathbb{N}^{21}$ mentre $\mathscr{P}$ contiene funzioni che lavorano in $\mathbb{N}$: per risolvere questo problema si definisca una nuova funzione f_C in cui viene applicato Cantor sull'array degli stati:

$$[\![C]\!](\underline{x}) = \underline{y} \quad \text{con} \quad \underline{x}, \underline{y} \in \mathbb{N}^{21}$$
$$f_C(x) = y \quad \text{con} \quad x = [\underline{x}]\ ,\ y = [\underline{y}]$$

Si noti che per passare da $[\![C]\!](\underline{x})$ a $f_C(x)$ si usano operazioni ricorsive parziali:

$$f_C(x) = y$$
$$\text{Pro} \in \mathscr{P} \quad \downarrow\uparrow \quad [\,] \in \mathscr{P}$$
$$[\![C]\!](\text{Pro}(0,x), \ldots, \text{Pro}(20,x)) = (\text{Pro}(0,y), \ldots, \text{Pro}(20,y))$$

Quindi si ha che f_C si comporta come $[\![C]\!](\underline{x})$ sullo stato prossimo. Basterà ora dimostrare che f_C effettivamente è una funzione ricorsiva parziale:

$$f_C \in \mathscr{P} \quad \Leftrightarrow \quad [\![C]\!](\underline{x}) \in \mathscr{P}$$

Vista la definizione induttiva di $[\![\,]\!]()$ si usi l'induzione strutturale:

- BASE:
  - $C \equiv \boxed{x_k := 0}$

    $$f_C(x) = [\text{Pro}(0,x), \ldots, 0, \ldots, \text{Pro}(20,x)]$$
    $$\uparrow$$
    $$\text{posizione } k$$

    Viene usata una composizione di funzioni $\in \mathscr{P} \Rightarrow f_{x_k:=0} \in \mathscr{P}$
  - $C \equiv \boxed{x_k := x_j + / \dot{-} 1}$

    $$f_C(x) = [\text{Pro}(0,x), \ldots, \text{Pro}(j,x) +/ \dot{-} 1, \ldots, \text{Pro}(20,x)]$$
    $$\uparrow$$
    $$\text{posizione } k$$

    Viene usata una composizione di funzioni $\in \mathscr{P} \Rightarrow f_{x_k:=x_j \pm 1} \in \mathscr{P}$
- PASSO:
  - $C \equiv \boxed{\texttt{begin } C_1;\ C_2;\ \ldots;\ C_m;\ \texttt{end}}$

    $$f_C(x) = f_{C_m}(\ldots f_{C_1}(x) \ldots)$$

    Viene usata una composizione di $f_{C_i} \in \mathscr{P}$ per ipotesi induttiva $\Rightarrow f_C \in \mathscr{P}$

– $C' \equiv \boxed{\texttt{while } x_k \neq 0 \texttt{ do } C}$

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

Sorge qui un problema: $e(x)$ non è costante; non basta quindi la composizione in quanto può essere applicata solo su un numero costante di funzioni.

Si dovrà allora definire una funzione $T \in \mathscr{P}$:

$$T(x, y) = f_C^y(x)$$

È facile farlo usando l'operatore di ricorsione primitiva RP:

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

Siccome:

1. $f_C \in \mathscr{P}$ per ipotesi induttiva
2. RP $\in \mathscr{P}$

$\Rightarrow \; T(x, y) \in \mathscr{P}$

$$\begin{aligned} e(x) &= \mu y(\text{Pro}(k, f_c^y(x)) = 0) \\ &= \mu y(\text{Pro}(k, T(x, y)) = 0) \end{aligned}$$

$e(x)$ è una minimalizzazione di $T(x, y) \in \mathscr{P}$, quindi $e(x) \in \mathscr{P}$

Infine:

$$f_{C'}(x) = f_C^{e(x)}(x) = T(x, e(x))$$

$f_{C'}$ è formato da una composizione di funzioni $\in \mathscr{P} \;\Rightarrow\; f_{C'} \in \mathscr{P}$.

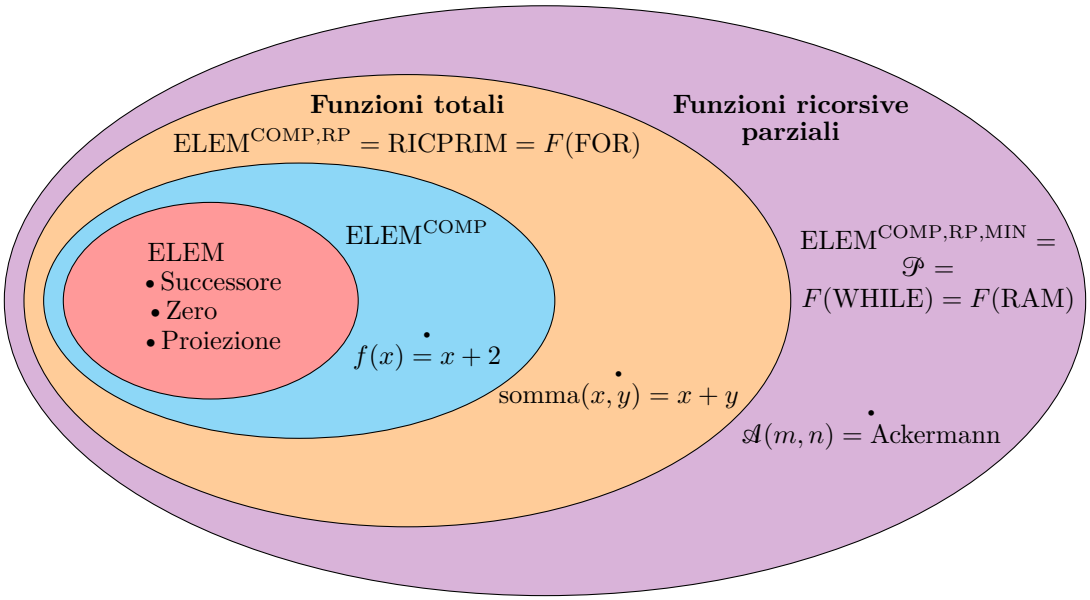
Per induzione strutturale si conclude che $F(\text{WHILE}) \subseteq \mathscr{P}$. □

Il precedente teorema unito al teorema 6 permettono di affermare che:

$$F(\text{WHILE}) = \mathscr{P}$$

2.7.6 Funzioni ricorsive totali

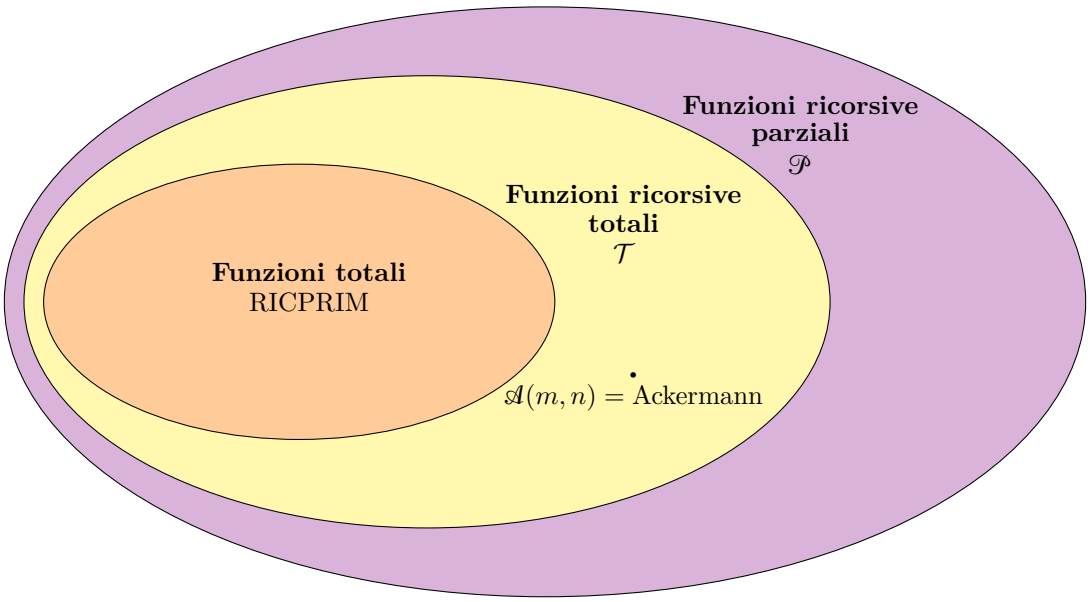
Il seguente schema mostra il quadro generale dell'idea intuitiva di calcolabilità:



L'insieme $\mathscr{P}$ delle funzioni ricorsive parziali contiene tutte le funzioni calcolabili, incluse quelle parziali. Questo vuol dire che in $\mathscr{P}$ ci sono anche programmi che non garantiscono una risposta in quanto potrebbero andare il loop.

Per delinare tutte le funzioni totali calcolabili, ovvero che possono essere calcolate da programmi che garantiscono una risposta, viene usato l'insieme delle funzioni ricorsive totali $\mathcal{T}$:

$$\mathcal{T} = \{\text{funzioni ricorsive totali}\}$$



2.7.7 Tesi di Church-Turing

Numerosi studiosi, a partire dagli anni 30, hanno provato a formalizzare il concetto di calcolabilità cercando quindi di definire quali siano le funzioni che possono essere calcolate da un qualsiasi programma in un qualsiasi sistema di calcolo.

Per farlo, sono stati introdotti innumerovoli e variegati sistemi di calcolo e, per ognuno di questi, è stato studiato che tipo di funzioni sono in grado di calcolare: **in tutti i casi si è arrivati alla stessa conclusione, ovvero che i sistemi di calcolo visti sono in gado di calcolare funzioni ricorsive primite.**

Da qui deriva la tesi di Church-Turing:

> Le classi delle funzioni intuitivamente calcolabili coincide con la classe $\mathscr{P}$ delle funzioni ricorsive parziali;

Per intuitivamente calcolabile si intende calcolabile da un modello di calcolo "ragionevole" come quelli mostrati nel corso degli ultimi cent'anni.

È importante sottolineare che la tesi di Church-Turing è una congettura, un'opinione. Non c'è infatti nessuna evidenza che non possa esistere un sistema di calcolo in grado di calcolare funzioni $f \notin \mathscr{P}$.

2.8 Assiomatizzazione dei sistemi di calcolo

Si vuole ora cercare di individuare alcune proprietà "buone", auspicabili da un sistema di calcolo, dette assiomi. Tutti i sistemi di programmazione che rispettano questi assiomi si diranno sistemi di programmazione accettabili.

In questo modo si potranno dimostrare determinate proprietà non sul singolo sistema ma su tutti i sistemi accettabili.

Per indicare un sistema di programmazione si userà la seguente notazione:

$$\{\varphi_i\}_{i\in\mathbb{N}}$$

dove φ_i è la funzione calcolata dal programma la cui codifica è $i \in \mathbb{N}$.

2.8.1 Assiomi di Rogers

Gli assiomi di Rogers vogliono essere quelle "buone" proprietà ricercate in un sistema di programmazione; in particolare si afferma che: un sistema di programmazione $\{\varphi_i\}$ si dice **accettabile (o spa)** se:

1. Aderisce alla tesi di Church-Turing;
2. Esiste un interprete universale;
3. Rispetta il teorema S_n^m.

Si vedranno ora nello specifico gli assiomi e si vedrà anche che il sistema RAM li rispetta.

Primo assioma

$\{\varphi_i\}$ rispetta il primo assioma se rispetta la tesi di Church-Turing ovvero:

$$\{\varphi_i\} = \mathscr{P}$$

Il sistema RAM, come ampiamente dimostrato nelle sezioni precedenti, rispetta il primo assioma:

$$F(\text{RAM}) = \mathscr{P}$$

Si vogliono considerare i sistemi di calcolo ne troppo poco potenti (sotto $\mathscr{P}$), ne troppo potenti (oltre $\mathscr{P}$).

Secondo assioma

$\{\varphi_i\}$ rispetta il secondo assioma se esiste un interpete universale ovvero:

$$\exists u \in \mathbb{N} : \forall x, n \in \mathbb{N} : \varphi_u(\langle x, n\rangle) = \varphi_n(x)$$

Vedi sezione 2.6.8 per il sistema RAM.

La presenza di un interprete universale permette di avere "un'algebra sui programmi".

Terzo assioma

$\{\varphi_i\}$ rispetta il terzo assioma se rispetta il teorema S_n^m.

Prima di mostrare il teorema S_n^m si vedrà un teorema più specifico, quello S_1^1.

Teorema 8 (S_1^1)**.** *Dato $\{\varphi_i\}$ RAM, esiste una funzione $S_1^1 \in \mathcal{T}$ tale che:*

$$\forall n, x, y \in \mathbb{N} \quad \varphi_n(\langle x, y\rangle) = \varphi_{S_1^1(n,y)}(x)$$

In altre parole è possibile costruire automaticamente programmi più specifici da programmi più generali, ottenuti prefissando alcuni input.

Dimostrazione. In generale il programma S_1^1 implementa la funzione:

$$S_1^1(n, y) = \bar{n} \;\text{ tale che }\; \varphi_{\bar{n}}(x) = \varphi_n(x, y)$$

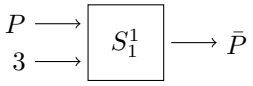
dove:

- n è la codifica del programma P a due variabili di input x, y
- $\bar{n}$ è la codifica del programma $\bar{P}$ ad una variabile la cui semantica è identica a quella di P a cui fisso a y il secondo input

Analizzando il sistema RAM si prenda un programma P a due variabili:

$$\begin{aligned} P \equiv\ & \texttt{// input } \langle x, y\rangle \texttt{ in } R_1 \\ & R_2 \leftarrow \sin(R_1) \\ & R_3 \leftarrow \mathrm{des}(R_1) \\ & R_0 \leftarrow R_2 + R_3 \end{aligned} \qquad \varphi_P(\langle x, y\rangle) = x + y$$

Si vuole produrre un programma $\bar{P}$ con un solo input x che restituisca $x + 3$; tutto questo partendo dal programma P. Si fissa quindi la seconda variabile di P a 3. La funzione S_1^1 dovrà restituire $\bar{P}$:



$$\begin{aligned} \bar{P} \equiv\ & \texttt{// input } x \texttt{ in } R_1 & \\ & R_0 \leftarrow R_0 + 1 & \Big\} \text{ fissa } y \text{ a } 3 \\ & R_0 \leftarrow R_0 + 1 & \\ & R_0 \leftarrow R_0 + 1 & \\ & R_1 \leftarrow \langle R_1, R_0\rangle & \} \text{ input di } P \\ & R_0 \leftarrow 0 & \} \text{ pulisci } R_0 \\ & R_0 \leftarrow R_2 + R_3 & \} \text{ richiama } P \end{aligned}$$

Si consideri il programma $\bar{P}$ nel caso generale, fissando la seconda variabile a un valore y:

$$\begin{array}{lll} \bar{P} \equiv \texttt{// input } x \texttt{ in } R_1 & & \text{codifica} \\ R_0 \leftarrow R_0 + 1 & \Big\} y \text{ volte} & \longmapsto \quad 0 \\ \quad \vdots & & \quad \vdots \\ R_0 \leftarrow R_0 + 1 & & \longmapsto \quad 0 \\ R_1 \leftarrow \langle R_1, R_0\rangle & \} \text{ input di } P & \longmapsto \quad \mathrm{s} \\ R_0 \leftarrow 0 & \} \text{ pulisci } R_0 & \longmapsto \quad \mathrm{t} \\ R_0 \leftarrow R_2 + R_3 & \} \text{ richiama } P & \longmapsto \quad \mathrm{n} \end{array}$$

La codifica del programma $\bar{P}$ è:

$$cod(\bar{P}) = \bar{n} = \langle \underbrace{0, \ldots, 0}_{y \text{ volte}}, s, t, n\rangle = S_1^1$$

Si può quindi notare che S_1^1 è:

1. Una funzione totale;
2. Programmabile.

$\Rightarrow \quad S_1^1 \in \mathcal{T}$ □

Teorema 9 (S_n^m)**.** *Dato $\{\varphi_i\}$ RAM, esiste una funzione $S_n^m \in \mathcal{T}$ tale che:*

$$\forall k \in \mathbb{N}, \underline{x} \in \mathbb{N}^m, \underline{y} \in \mathbb{N}^n \quad \varphi_k(\langle \underline{x}, \underline{y}\rangle) = \varphi_{S_n^m(k, \underline{y})}(\langle \underline{x}\rangle)$$

Il teorema è una generalizzazione di S_1^1; S_n^m tratta programmi a $m + n$ input in cui si prefissano n input e se ne lasciano liberi m. Anche la dimostrazione è una semplice generalizzazione di quella di S_1^1.

2.8.2 Compilatori tra spa

Dati due sistemi di programmazione accettabili (spa) $\{\varphi_i\}$ e $\{\psi_j\}$, un compilatore dal primo al secondo spa è una funzione $t : \mathbb{N} \to \mathbb{N}$ che possiede le seguenti proprietà:

1. Correttezza: $\forall i \in \mathbb{N} \quad \varphi_i = \psi_{t(i)}$;
2. Completezza: compila ogni programma $i \in \mathbb{N}$;
3. Programmabilità: esiste un programma che implementa t.

$\Big\} t \in \mathcal{T}$

Teorema 10. *Dati due spa, esiste un compilatore tra essi.*

Dimostrazione. Si considerino due spa $\{\varphi_i\}$ e $\{\psi_j\}$. Essendo spa rispettano gli assiomi:

1. $\{\varphi_i\} = \mathscr{P}$
2. $\exists u : \varphi_u(\langle x, n\rangle) = \varphi_n(x)$
3. $\exists S_1^1 \in \mathcal{T} : \varphi_n(\langle x, y\rangle) = \varphi_{S_1^1(n,y)}(x)$

$$\varphi_i(x) \overset{(2)}{=} \varphi_u(\langle x, i\rangle) \overset{(1)}{=} \psi_e(\langle x, i\rangle) \overset{(3)}{=} \psi_{S_1^1(e,i)}(x) \tag{8}$$

(2) Applicazione diretta dell'assioma 2;

(1) Entrambi gli spa possono calcolare le funzioni in $\mathscr{P}$ (assioma 1): esiste in $\{\psi_j\}$ un programma e con la stessa semantica del programma u;

(3) Tramite la funzione S_1^1 (assioma 3) ottengo il programma che fissa la seconda variabile del programma e ad i.

Si è quindi trovata la funzione $t(i) = S_1^1(e, i)$ valida per ogni $i \in \mathbb{N}$. t è un compilatore poichè:

1. $t \in \mathcal{T}$ in quanto $S_1^1 \in \mathcal{T}$(assioma 3);
2. T è corretto in quanto $\varphi_i = \psi_{t(i)}$ (8).

□

Corollario 1. *Dati gli spa A, B, C esiste sempre un compilatore da A a B scritto in C.*

Dimostrazione.

- Per il teorema 10 esiste un compilatore $t \in \mathcal{T}$ tra A e B;
- Per il primo assioma C contiene tutte le funzioni ricorsive parziali tra cui è presente anche t (in quanto $t \in \mathcal{T} \subset \mathscr{P} = C$)

□

Teorema 11 (Isomorfismo tra spa (Rogers))**.** *Dati due spa $\{\varphi_i\}$ e $\{\psi_j\}$, esiste un compilatore $t : \mathbb{N} \to \mathbb{N}$ tra i due sistemi tale che t è invertibile (t^{-1} è un decompilatore).*

Si noti che la decompilazione di un programma non restituisce semplicemente un programma con la stessa semantica, ma restituisce esattamente lo stesso programma che è stato originariamente compilato.

2.8.3 Teorema di ricorsione

Teorema 12 (Teorema di Ricorsione)**.** *Dato un spa* $\{\varphi_i\}$*, per ogni* $t : \mathbb{N} \to \mathbb{N}$ *ricorsiva totale vale:*

$$\exists n \in \mathbb{N} : \varphi_n = \varphi_{t(n)}$$

A parole, sia t un programma che trasforma programmi con lo scopo di cambiare totalmente la semantica del programma (rompendolo su tutti gli input). Qualunque sia la manipolazione effettuata da t, esisterà sempre un programma n che non viene rotto dalla manipolazione di t.

Dimostrazione.

D'ora in avanti, per semplicità, si scriverà $\varphi_i(x, y)$ al posto di $\varphi_i(\langle x, y\rangle)$.

Si applichino i (3) assiomi:

$$\varphi_{\varphi_i(i)}(x) \overset{(2)}{=} \varphi_{\varphi_u(i,i)}(x) \overset{(2)}{=} \varphi_u(x, \varphi_u(i, i)) = f(x, i) \tag{9}$$

$\varphi_u(x, \varphi_u(i, i))$ è la composizione di $\varphi_u \in \mathscr{P}$ con se stesso al variare delle variabili x e i. $f(x, i)$ è quindi ricorsiva parziale:

$$f(x, i) \in \mathscr{P}$$

$$f(x, i) \overset{(1)}{=} \varphi_e(x, i) \overset{(3)}{=} \varphi_{S_1^1(e,i)}(x) \tag{10}$$

Si consideri una qualsiasi traduzione $t \in \mathcal{T}$ e la si applichi a $S_1^1(e, i)$:

$$t(S_1^1(e, i))$$

$$t \in \mathcal{T} \ , \ S_1^1 \in \mathcal{T}$$

$t(S_1^1(e, i))$ è ricorsiva totale e grazie al primo assioma si può dire che esiste un programma m che la calcola:

$$\exists m \in \mathbb{N} : \varphi_m(i) = t(S_1^1(e, i)) \tag{11}$$

Sia:

$$n = S_1^1(e, m)$$

$$(9), (10) \ \Rightarrow \ \varphi_{\varphi_i(i)}(x) = \varphi_{S_1^1(e,i)}(x) \tag{12}$$

$$\left.\begin{aligned} \varphi_n(x) = \varphi_{S_1^1(e,m)}(x) &\overset{(12)}{=} \varphi_{\varphi_m(m)}(x) \\ \varphi_{t(n)}(x) = \varphi_{t(S_1^1(e,m))}(x) &\overset{(11)}{=} \varphi_{\varphi_m(m)}(x) \end{aligned}\right\} \quad \Rightarrow \quad \varphi_n = \varphi_{t(n)}$$

□

Quine

Dato un spa, esiste al suo interno un programma che restituisce se stesso (la sua codifica)? Formalmente:

$$\exists j \in \mathbb{N} : \forall x \in \mathbb{N} \quad \varphi_j(x) = j?$$

Questi programmi sono dette **Quine**.

Si prenda RAM e il seguente programma:

$$P \equiv \left.\begin{array}{l} R_0 \leftarrow R_0 + 1 \\ R_0 \leftarrow R_0 + 1 \\ \quad \vdots \\ R_0 \leftarrow R_0 + 1 \end{array}\right\} j \text{ volte} \ \Rightarrow \ cod(P) = \underbrace{\langle 0, 0, \ldots, 0\rangle}_{j \text{ volte}} = Z(j)$$

Alla fine dell'esecuzione di P si avrà $R_0 = j$ a prescindere dall'input x:

$$\forall x \in \mathbb{N} \quad \varphi_{Z(j)}(x) = j \tag{13}$$

Si definisca la codifica del programma P con $Z(j)$ che è uguale alla codifica di Cantor di una lista di j zeri in quanto l'unica istruzione usata ha codifica 0 (vedi sezione 2.6.1). È evidente che per qualsiasi valore j, la funzione $Z(j)$ esiste ed è quindi totale; inoltre è facilmente implementabile. Si può quindi dire che:

$$Z(j) \in \mathcal{T}$$

Il teorema 12 afferma che:

$$\exists j \in \mathbb{N} : \varphi_j = \varphi_{Z(j)} \tag{14}$$

Da (13) e (14) si ottiene che esiste un programma $j \in \mathbb{N}$ tale che:

$$\varphi_j(x) \overset{(14)}{=} \varphi_{Z(j)}(x) \overset{(13)}{=} j$$
$$\Downarrow$$
$$\varphi_j(x) = j$$

Quindi, sì, **nel sistema RAM esiste una Quine. Questo vale anche in generale per tutti gli spa che ammettono una codifica di programma.**

Compilatori completamente errati

Dati due spa $\{\varphi_i\}$ e $\{\psi_j\}$, esiste un compilatore dal primo al secondo completamente errato?

Con compilatore completamente errato si intende una funzione $t : \mathbb{N} \to \mathbb{N}$ tale che:

- $t \in \mathcal{T}$
- $\forall i \in \mathbb{N} \quad \varphi_i \neq \psi_{t(i)}$

ovvero un compilatore che sbaglia sempre.

Si consideri una qualunque traduzione $t \in \mathcal{T}$:

Con i (3) assiomi si può dire che:

$$\psi_{t(i)}(x) \overset{(2)}{=} \psi_u(x, t(i)) \overset{(1)}{=} \varphi_e(x, t(i)) \overset{(3)}{=} \varphi_{S_1^1(e,t(i))}(x) = \varphi_{g(i)}(x)$$
$$\psi_{t(i)}(x) = \varphi_{g(i)}(x) \tag{15}$$

$$g(i) = S_1^1(e, t(i)) \qquad S_1^1, t \in \mathcal{T} \;\Rightarrow\; g \in \mathcal{T}$$

Si applichi il teorema di ricorsione:

$$g \in \mathcal{T} \;\Rightarrow\; \exists i \in \mathbb{N} : \varphi_i = \varphi_{g(i)} \tag{16}$$

$$(15), (16) \;\Rightarrow\; \exists i \in \mathbb{N} : \varphi_i = \psi_{t(i)}(x)$$

Quindi, no, **non può esistere un compilatore sempre errato**.

2.9 Problemi di decisione

Un problema di decisione è un problema che richiede una soluzione "booleana": o sì o no. Formalmente, un problema si può definire come:

Π		← nome del problema
Istanza:	$x \in D$	← input del problema
Domanda:	$p(x)$?	← $p(x)$ è vera?

2.9.1 Decidibilità

Π	
Istanza:	$x \in D$
Domanda:	$p(x)$?

Π è decidibile sse esiste un programma P_Π che restituisce 1 se la proprietà p è verificata su ogni input, o restituisce 0 se la proprietà p non è verificata su ogni input.

Una formulazione più matematica del concetto di decidibilità parte dalla definizione della **funzione soluzione** Φ associata al problema Π:

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

Π è decidibile sse $\Phi_\Pi \in \mathcal{T}$.

Chiaramente, le due definizioni si equivalgono:

- Il programma P_Π calcola Φ_Π;
- Se $\Phi_\Pi \in \mathcal{T}$ allora esiste un programma che la calcola con la stessa semantica di P_Π.

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

1. O esibendo un algoritmo di soluzione;
2. O mostrando che Φ_Π è ricorsiva totale.

Esempi

PARITÀ (PR)	
Istanza:	$n \in \mathbb{N}$
Domanda:	n è pari ?

Funzione soluzione

$\Phi_{\text{PR}}(n) = 1 \dot{-} (n \bmod 2)$

$\Phi_{\text{PR}} \in \mathcal{T} \Rightarrow$ PR decidibile

EQ. DIOFANTEA (ED)	
Istanza:	$a, b, c \in \mathbb{N}^+$
Domanda:	$\exists x, y \in \mathbb{Z} : ax + by = c$?

Funzione soluzione

$\Phi_{\text{ED}}(a, b, c) = 1 \dot{-} (c \bmod mcd(a, b))$

$\Phi_{\text{ED}} \in \mathcal{T} \Rightarrow$ ED decidibile

FERMAT (FR)	
Istanza:	$n \in \mathbb{N}^+$
Domanda:	$\exists x, y, z \in \mathbb{N}^+ : x^n + y^n = z^n$?

Funzione soluzione

$\Phi_{\text{FR}}(n) = 1 \dot{-} (n \dot{-} 2)$ [A.Wiles 1994]

$\Phi_{\text{FR}} \in \mathcal{T} \Rightarrow$ FR decidibile

RAGGIUNGIBILITÀ (RG)	
Istanza:	Grafo $G = (\{1, \ldots, n\}, E)$
Domanda:	$\exists$ un cammino dal nodo 1 al nodo n ?

Algoritmi di decisione

Si effettua una visita in ampiezza o profondità, si usano algoritmi per cammini minimi etc...

2.9.2 Problema dell'arresto

ARRESTO SUL PROGRAMMA P (AR_P)	
Istanza:	$x \in \mathbb{N}$
Domanda:	$\varphi_P(x)\downarrow$?

Si noti che AR_P è una classe di problemi in cui viene fissato P e si chiede se il programma P termina oppure no.

La decidibilità di AR_P dipende dal programma P; per alcuni programmi si può trovare una risposta ed è quindi decidibile. Tuttavia esistono dei programmi dove AR_P è indecidibile.

Si prenda il seguente programma $\hat{P}$:

$$\begin{aligned}\hat{P} \equiv\ & \text{input}(x)\\ & Z := U(x,x)\\ & \text{output}(Z)\end{aligned}$$

Dove U è l'interprete universale.

E si definisca il problema $\text{AR}_{\hat{P}}$:

$$\varphi_{\hat{P}}(x) = \varphi_U(x,x) = \varphi_x(x)$$

$\text{AR}_{\hat{P}}$	
Istanza:	$x \in \mathbb{N}$
Domanda:	$\varphi_{\hat{P}}(x) = \varphi_x(x)\downarrow$?

Teorema 13 (Indecibilità di $\text{AR}_{\hat{P}}$)**.** *$AR_{\hat{P}}$ è indecidibile.*

Dimostrazione. Si ipotizzi per assurdo che $\text{AR}_{\hat{P}}$ sia decidibile. Si avrebbe quindi una funzione soluzione ricorsiva totale:

$$\Phi_{\text{AR}_{\hat{P}}}(x) = \begin{cases} 1 & \varphi_{\hat{P}}(x) = \varphi_x(x)\downarrow \\ 0 & \varphi_{\hat{P}}(x) = \varphi_x(x)\uparrow \end{cases}$$

Partendo da $\Phi_{\text{AR}_{\hat{P}}}(x) \in \mathcal{T}$ posso creare la seguente funzione sempre ricorsiva totale:

$$f(x) = \begin{cases} 0 & \Phi_{\text{AR}_{\hat{P}}}(x) = 0 \\ \varphi_x(x) + 1 & \Phi_{\text{AR}_{\hat{P}}}(x) = 1 \end{cases}$$

Che è uguale a:

$$f(x) = \begin{cases} 0 & \varphi_x(x)\uparrow \\ \varphi_x(x) + 1 & \varphi_x(x)\downarrow \end{cases}$$

Essendo $f \in \mathcal{T}$ esiste un programma α che la calcola:

$$f(x) = \varphi_\alpha(x)$$

Si valuti φ_α in α:

$$\varphi_\alpha(\alpha) = \begin{cases} 0 & \varphi_\alpha(\alpha)\uparrow \\ \varphi_\alpha(\alpha) + 1 & \varphi_\alpha(\alpha)\downarrow \end{cases} \qquad \text{(ASSURDO)}$$

$\varphi_\alpha(\alpha)$ non può esistere in quanto:

1. Nel primo caso ($\varphi_\alpha(\alpha)\uparrow$) se $\varphi_\alpha(\alpha)$ non termina allora $\varphi_\alpha(\alpha) = 0$;
2. Nel secondo caso ($\varphi_\alpha(\alpha)\downarrow$) se $\varphi_\alpha(\alpha)$ termina si ha che $\varphi_\alpha(\alpha) = \varphi_\alpha(\alpha) + 1$.

□

Il problema generale dell'arresto

La versione più generale del problema dell'arresto è stato data da Turing nel 1936 ed è:

ARRESTO (AR)	
Istanza:	$x, y \in \mathbb{N}$
Domanda:	$\varphi_y(x)\downarrow$?

dove x è il dato e y il programma.

Teorema 14 (Indecibilità di AR)**.** *AR è indecidibile.*

Dimostrazione. Si assuma, per assurdo, che AR sia decidibile; esiste quindi una funzione soluzione:

$$\Phi_{\mathrm{AR}}(x,y) = \begin{cases} 0 & \varphi_y(x)\uparrow \\ 1 & \varphi_y(x)\downarrow \end{cases}$$

Si valuti il caso in cui $x = y$:

$$\Phi_{\mathrm{AR}}(x,x) = \begin{cases} 0 & \varphi_x(x)\uparrow \\ 1 & \varphi_x(x)\downarrow \end{cases}$$

Si noti che $\Phi_{\mathrm{AR}}(x,x) = \Phi_{\mathrm{AR}_{\hat{P}}}(x,x)$; quindi nel caso $x = y$ il problema AR diventa uguale al problema $\mathrm{AR}_{\hat{P}}$. L'assunzione iniziale della dimostrazione afferma che esiste un programma in grado di decidere $\mathrm{AR}_{\hat{P}}$, cosa che il teorema 13 dimostra non essere vera. Si ha quindi l'assurdo. □

2.10 Riconoscibilità automatica di insiemi

L'insieme $A \subseteq \mathbb{N}$ è riconoscibile automaticamente se esiste un programma P_A che classifica correttamente **ogni** elemento di $\mathbb{N}$ come appartenente o meno ad A;

$P_A(x)$ restituisce 1 se $x \in A$ mentre restituisce 0 se $x \notin A$.

$P_A(x)$ è corretto e termina su ogni input.

Funzione caratteristica

Preso un insieme $A \subseteq \mathbb{N}$, la sua funzione caratteristica $\mathcal{X}$ è la funzione $\mathcal{X}_A : \mathbb{N} \to \{0,1\}$ tale che;

$$\mathcal{X}_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

2.10.1 Insiemi ricorsivi

L'insieme $A \subseteq \mathbb{N}$ è un insieme ricorsivo sse esiste un programma P_A che si arresta su ogni input classificando correttamente gli elementi di $\mathbb{N}$ in base alla loro appartenenza ad A.

Equivalentemente, A è ricorsivo sse $\mathcal{X}_A \in \mathcal{T}$.

Si dirà, con un abuso di notazione, che gli insiemi ricorsivi sono decidibili. Questo perchè ad ogni insieme $A \subseteq \mathbb{N}$ si può associare il corrispondente problema RIC_A di riconoscimento, che punta a decidere se un elemento x appartiene o meno ad A. Essendo A ricorsivo si ha che la funzione soluzione Φ_{RIC_A} coincide con la funzione caratteristica $\mathcal{X}_A$ e quindi il problema RIC_A è decidibile.

Allo stesso modo, sempre con un abuso di notazione, si dice che un problema di decisione decidibile è ricorsivo. Questo perchè ad ogni problema di decisione posso associare l'insieme delle istanze a risposta positiva.

2.10.2 Insiemi non ricorsivi

Un esempio di insieme non ricorsivo è l'insieme A delle variabili con le quali il programma $\hat{P}$ termina.

$\mathrm{AR}_{\hat{P}}$	
Istanza:	$x \in \mathbb{N}$
Domanda:	$\varphi_{\hat{P}}(x) = \varphi_x(x)\downarrow$?

$$A = \{x \in \mathbb{N} : \varphi_x(x)\downarrow\}$$

Se A fosse ricorsivo, vorrebbe dire che esisterebbe un programma P_A in grado di dire se un elemento sta al suo interno o meno. P_A però, renderebbe la risoluzione del problema dell'arresto $\mathrm{AR}_{\hat{P}}$ immediata, in quanto per decidere se, data una variabile x, il programma $\hat{P}$ termina, basterebbe verificare tramite P_A che x stia o meno in A. Tutto questo però è impossibile (teorema 13).

2.10.3 Relazioni ricorsive

$R \subseteq \mathbb{N} \times \mathbb{N}$ è una relazione ricorsiva sse R è ricorsivo ovvero $\mathcal{X}_R \in \mathcal{T}$; equivalentemente R è una relazione ricorsiva se esiste un programma P_R che, presi due valori $x, y \in \mathbb{N}$, restituisce 1 se $x \ R \ y$, o 0 se $x \not R \ y$.

Un'importante relazione ricorsiva è:

$$R_P = \{(x, y) \in \mathbb{N}^2 : P \text{ su input } x \text{ termina in } y \text{ passi}\}$$

Si può facilmente costruire un programma che implementa R_P aggiungendo una variabile che conta i passi e, nel momento in cui supera gli y passi restituisce 0, altrimenti se termina prima restituisce 1.

2.10.4 Insiemi ricorsivamente numerabili

$A \subset \mathbb{N}$ è ricorsivamente numerabile se è automaticamente listabile, ovvero esiste una routine F che su input $i \in \mathbb{N}$ restituisce l'i-esimo elemento di A.

```
i := 0;
while True
    output(F(i));
    i := i + 1;
```

Per alcuni insiemi è il meglio che si può ottenere, non si riesce a riconoscere esattamente l'insieme ma si riesce a listarlo fino a un certo punto.

A questo punto si può scrivere un algoritmo che "tenta" di riconoscere A:

```
input(x);
i := 0;
while F(i) ≠ x
    i := i + 1;
output(1);
```

Per alcuni insiemi è il meglio che si può ottenere, non si riesce a riconoscere esattamente l'insieme ma si riesce a listarlo fino a un certo punto. Preso in input x, il programma restituisce 1 se $x \in A$ o va in loop se $x \notin A$.

L'insieme $A \subset \mathbb{N}$ è ricorsivamente numerabile sse:

- $A = \emptyset$

- $A = \text{Im}_f$ con $f : \mathbb{N} \to \mathbb{N} \in \mathcal{T}$ $\qquad\qquad$ $(A = \{f(0), f(1), \dots\})$

Teorema 15. *Le seguenti definizioni sono equivalenti:*

1. *A è ricorsivamente numerabile ($A = Im_{f \in \mathcal{T}}$)*
2. *$A = Dom_f$ con $f \in \mathcal{P}$*
3. *Esiste una relazione $R \subseteq \mathbb{N}^2$ ricorsiva tale che:*

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

Dimostrazione. Si dimostrerà che (1) $\Rightarrow$ (2) $\Rightarrow$ (3) $\Rightarrow$ (1).

- (1) $\Rightarrow$ (2)

  Si consideri l'algoritmo di riconoscimento parziale di A:

P

input(x);
$i := 0$;
`while` $F(i) \neq x$
  $i := i + 1$;
output(1);

  $$\varphi_P(x) = \begin{cases} 1 & x \in A \\ \perp & x \notin A \end{cases} \Rightarrow A = \text{Dom}\varphi_P$$

  Siccome φ_P viene calcolato da P è ricorsiva parziale.

- (2) $\Rightarrow$ (3)

  $A = \text{Dom}_{f \in \mathcal{P}} \Rightarrow \exists R \subseteq \mathbb{N}^2$ ricorsiva tale che: $A = \{x \mid \exists y : (x, y) \in R\}$

  Se $f \in \mathcal{P}$ allora esiste un programma P tale che $f = \varphi_P$.

  $$R_P = \{(x, y) \in \mathbb{N}^2 : P \text{ su input } x \text{ termina in } y \text{ passi}\}$$

  Si definisca il seguente insieme:

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

  - $A \subseteq B$: se $x \in A = \text{Dom}\varphi_P$ allora P termina in y passi, quindi $(x, y) \in R_P$ e quindi $x \in B$.
  - $B \subseteq A$: se $x \in B$ allora il programma P su input x termina in un numero finito di passi e quindi $x \in \text{Dom}\varphi_P \Rightarrow x \in A$

  Quindi $A = B = \{x \mid \exists y : (x, y) \in R_P\}$ con R_P relazione ricorsiva.

- (3) $\Rightarrow$ (1)

  $A = \{x \mid \exists y : (x, y) \in R\}$ con r ricorsivo $\Rightarrow$ $A = \text{Im}_{f \in \mathcal{T}}$

  Sia $a \in A$ con $A \neq \varnothing$ e sia $t : \mathbb{N} \to \mathbb{N}$:

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

  Poichè R ricorsiva esiste un programma P_R che ne riconosce gli elementi.

```
input(n);
x := sin(n);
y := des(n);
if P_R(x, y) = 1
    output(x);
else
    output(a);
```

- $A \subseteq \mathrm{Im}_t$: $x \in A \Rightarrow (x,y) \in R \Rightarrow t(\langle x,y\rangle) = x \Rightarrow x \in \mathrm{Im}_t$
- $\mathrm{Im}_t \subseteq A$:
  - $x \in \mathrm{Im}_t \Rightarrow x = a \in A$
  - $x = \sin(n)$ con $n = \langle x,y\rangle$ per qualche y tale che $(x,y) \in R \Rightarrow x \in A$

Quindi $A = \mathrm{Im}_t$ con $t \in \mathcal{T}$.

□

2.10.5 Confronto tra insiemi

$\mathrm{AR}_{\hat{P}}$	
Istanza:	$x \in \mathbb{N}$
Domanda:	$\varphi_{\hat{P}}(x) = \varphi_x(x)\downarrow$?

$\Rightarrow A = \{x : \varphi_x(x)\downarrow\}$

A non è ricorsivo ma è ricorsivamente numerabile:

P

input(x);
$U(x,x)$;
output(1);

$$\varphi_P(x) = \begin{cases} 1 & \varphi_x(x)\downarrow \quad x \in A \\ 0 & \text{altrimenti} \quad x \notin A \end{cases}$$

Infatti $A = \mathrm{Dom}_{\varphi_P}$ con $\varphi_P \in \mathcal{P}$.

Teorema 16. *$A \subseteq \mathbb{N}$ ricorsivo $\Rightarrow$ A ricorsivamente numerabile.*

Dimostrazione. A ricorsivo $\Rightarrow$ esiste un programma P_A che classifica ogni elemento di A.

```
P
input(x);
if P_A(x) = 1
    output(1);
else
    while(1 > 0);
```

$$\varphi_P(x) = \begin{cases} 1 & x \in A \\ \perp & x \notin A \end{cases} \Rightarrow A = \mathrm{Dom}_{\varphi_P}$$

□

Teorema 17 (Chiusura degli insiemi ricorsivi)**.** *La classe degli insiemi ricorsivi è un'Algebra di Boole; è quindi chiusa per complemento, intersezione e unione.*

Dimostrazione. Siano A, B ricorsivi; esistono quindi due programmi P_A e P_B che li riconoscono. Si vedrà l'esistenza di programmi per riconoscere:

- A^c:

```
input(x);
output(1 ∸ P_A(x));
```

- $A \cap B$:

```
input(x);
output(min {P_A(x), P_B(x)});
```

- $A \cup B$:

```
input(x);
output(max {P_A(x), P_B(x)});
```

□

Teorema 18. *$A^c = \{x : \varphi_x(x)\uparrow\}$ non è ricorsivo.*

Dimostrazione. Se A^c fosse ricorsivo, per il teorema 17 anche il suo complemento A sarebbe ricorsivo, il che è assurdo. □

Teorema 19. *A ric. numerabile $\wedge$ A^c ric. numerabile $\Rightarrow$ A ricorsivo.*

Dimostrazione. Se sia A che A^c sono ric. numerabili vuol dire che entrambi hanno un programma che è in grado di listarli. Si crei un programma che chiama, alternandoli, i due programmi, listando un pò per volta sia A che A^c. Sia se $x \in A$ che $x \notin A$ il programma termina. □

Teorema 20. *La classe degli insiemi ricorsivamente numerabili è chiusa per unione e intersezione ma non per il complemento.*

Dimostrazione.

- Complemento: $A = \{x : \varphi_x(x)\downarrow\}$ è ricorsivamente numerabile mentre A^c non lo è;
- Intersezione: siano F e G le funzioni in grado di listare i due insiemi ric. numer. A e B:

```
P'
input(x);
i := 0;
while F(i) ≠ x
    i := i + 1;
while G(i) ≠ x
    i := i + 1;
output(1);
```

$$\varphi_{P'}(x) = \begin{cases} 1 & x \in A \cap B \\ \perp & \text{altrimenti} \end{cases}$$

$$\mathrm{Dom}_{\varphi_{P'}} = A \cap B$$

$$\varphi_{P'} \in \mathscr{P}$$

- Unione: siano F e G le funzioni in grado di listare i due insiemi ric. numer. A e B:

```
P''
input(x);
i := 0;
while True
    if F(i) = x
        output(1);
    if G(i) = x
        output(1);
    i := i + 1;
```

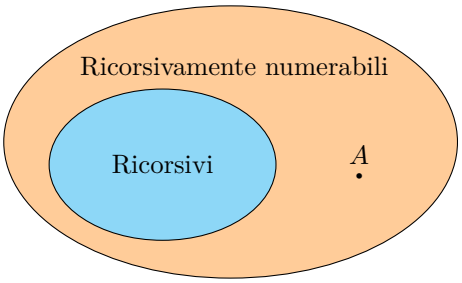
$$\varphi_{P''}(x) = \begin{cases} 1 & x \in A \cup B \\ \perp & \text{altrimenti} \end{cases}$$

$$\mathrm{Dom}_{\varphi_{P''}} = A \cup B$$

$$\varphi_{P''} \in \mathscr{P}$$

□

Situazione insiemistica



- $A = \{x : \varphi_x(x)\downarrow\}$
- $A^c = \{x : \varphi_x(x)\uparrow\}$

Teorema di Rice

L'insieme $I \subseteq \mathbb{N}$ rispetta le funzioni sse $(a \in I \ \wedge \ \varphi_a = \varphi_b) \ \Rightarrow \ b \in I$

Teorema 21 (Teorema di Rice)**.** *Sia $I \subseteq \mathbb{N}$ un insieme che rispetta le funzioni. Allora I è ricorsivo solo se $I = \emptyset$ oppure $I = \mathbb{N}$.*

Dimostrazione. Sia I un insieme che rispetta le funzioni con $I \neq \emptyset$ e $I \neq \mathbb{N}$. Per assurdo, si ipotizzi I ricorsivo e che rispetta le funzioni.

Si definisca la funzione $t : \mathbb{N} \to \mathbb{N}$ come:

$$t(x) = \begin{cases} \bar{a} & n \in I \\ a & n \notin I \end{cases} \qquad\qquad t \in \mathcal{T}$$

dove $a \in I$ e $\bar{a} \notin I$.

Essendo $t \in \mathcal{T}$ il teorema di ricorsione (12) garantisce per gli spa l'esistenza di un valore $d \in \mathbb{N}$:

$$\varphi_d(x) = \varphi_{t(d)}$$

Si prenda d:

- $d \in I$: poichè I rispetta le funzioni e $\varphi_d = \varphi_{t(d)}$ allora deve valere che $t(d) \in I$. Ma $t(d) = \bar{a} \notin I \ \Rightarrow$ contraddizione.
- $d \notin I$: $t(d) = a \in I$ e poichè I rispetta le funzioni e $\varphi_d = \varphi_{t(d)}$ si ha $d = t(d) = a \in I \ \Rightarrow$ contraddizione.

□

Il teorema di Rice è uno strumento molto utile per mostrare che un insieme $a \in \mathbb{N}$ non è ricorsivo; per verificarlo si può infatti:

1. Mostrare che A rispetta le funzioni;
2. Mostrare che $A \neq \emptyset$ e $A \neq \mathbb{N}$

$\overset{\text{Rice}}{\Longrightarrow}$ A non è ricorsivo

Verifica automatica de correttezza del software

Problema: si può scrivere un programma V che testa automaticamente se un programma sia corretto o meno?

Sia PC l'insieme dei programmi corretti:

$$PC = \{P \in \mathbb{N} : P \text{ è corretto}\}$$

PC può essere riconosciuto da V? PC è ricorsivo?

- PC **rispetta le funzioni** in quanto:

$$P \in PC \ \wedge \ \varphi_P = \varphi_{P'} \ \Rightarrow \ P' \in PC$$

  Se il programma P è corretto e la semantica del programma P' è la stessa di P, allora anche P' è corretto.

- $PC \neq \emptyset \wedge PC \neq \mathbb{N}$: sarebbe insensato avere delle specifiche con le quali nessun programma è corretto o delle specifiche dove tutti i programmi sono corretti.

Per il teorema di Rice si può affermare che PC non è ricorsivo. Non esiste quindi un programma V che lo identifica correttamente.