

Informatica Teorica

Manuel Pagliuca

27 novembre 2021

Indice

| | |
|--|-----------|
| 1 Introduzione | 4 |
| 1.1 Cosa studia l'informatica? | 4 |
| 1.2 Come l'informatica risolve i problemi? | 4 |
| 2 Syllabus | 5 |
| 3 Il nostro linguaggio: la matematica | 6 |
| 3.1 Funzione | 6 |
| 3.2 Classi di funzioni | 6 |
| 3.2.1 Funzioni iniettive | 6 |
| 3.2.2 Funzioni suriettive | 7 |
| 3.3 Insieme immagine di una funzione | 7 |
| 3.4 Funzioni biettive | 8 |
| 3.5 Inversa di una funzione | 8 |
| 3.6 Composizione di funzioni | 8 |
| 3.6.1 Funzione identità | 9 |
| 3.6.2 Definizione alternativa di funzione inversa | 9 |
| 3.6.3 Funzioni totali e parziali | 9 |
| 3.6.4 Totalizzazione di una funzione parziale | 10 |
| 3.6.5 Prodotto cartesiano | 10 |
| 3.6.6 Insieme di funzioni | 11 |
| 3.6.7 Funzione di valutazione | 12 |
| 3.7 Modellare teoricamente un sistema di calcolo | 12 |
| 3.7.1 Potenza computazionale | 13 |
| 3.7.2 Importanza del calcolo di funzione | 14 |
| 3.7.3 Cardinalità degli insiemi - I° | 14 |
| 3.7.4 Relazione | 15 |
| 3.7.5 Relazioni di equivalenza e partizioni | 16 |
| 3.7.6 Cardinalità degli insiemi - II° | 18 |
| 3.7.7 Insiemi numerabili | 19 |
| 3.8 Insiemi non numerabili | 20 |
| 3.8.1 Insieme delle parti di \mathbb{N} | 22 |
| 3.8.2 Insieme $\mathbb{N}^{\mathbb{N}}$ | 23 |
| 4 Calcolabilità | 23 |
| 4.1 Cosa è calcolabile? | 23 |
| 4.1.1 Dimostrazione $DATI \sim \mathbb{N}$ | 24 |
| 4.1.2 Prefazione alla dimostrazione $PROG \sim \mathbb{N}$ | 31 |
| 4.1.3 Sistema di calcolo RAM | 32 |
| 4.1.4 Dimostrazione $PROG \sim \mathbb{N}$ | 38 |
| 4.1.5 Il sistema di calcolo While | 42 |
| 4.1.6 Definizione formale di semantica di un programma WHILE | 46 |
| 4.1.7 Definizione induttiva della semantica while | 47 |
| 4.1.8 Concetto di traduzione | 50 |

| | | |
|--------|--|----|
| 4.1.9 | Dimostrazione $F(WHILE) \subseteq F(RAM)$ | 51 |
| 4.1.10 | Dimostrazione $F(RAM) \subseteq F(WHILE)$ | 54 |
| 4.2 | Concetto di calcolabilità | 61 |
| 4.2.1 | Strumenti matematici (chiusure di insiemi) | 61 |
| 4.2.2 | Il nucleo di funzioni calcolabili | 66 |
| 4.2.3 | $RICPRIM$ vs $F(WHILE)$ | 69 |
| 4.2.4 | $RICPRIM \subseteq F(WHILE)$ | 69 |
| 4.2.5 | $\mathcal{P} \subseteq F(WHILE)$ | 73 |
| 4.2.6 | Dimostrazione $F(WHILE) \subseteq \mathcal{P}$ | 75 |
| 4.2.7 | Tesi di Church-Turing | 78 |

1 Introduzione

Nei corsi di informatica applicata come quelli di: Sistemi Operativi, Basi di dati, ecc... l'oggetto di studio è definito dal corso, e l'informatica è lo strumento per studiare questo oggetto.

Nel corso di informatica teorica l'oggetto di studio è l'informatica stessa, si studiano i fondamenti dell'informatica (come un corso di sistemi operativi effettua uno studio sui fondamenti dei sistemi operativi).

Per eseguire lo studio di questa disciplina ci si pone le due domande fondamentali nei confronti dell'informatica *cosa* e *come*.

1.1 Cosa studia l'informatica?

L'informatica è una disciplina che studia l'informazione e l'elaborazione automatica mediante sistemi di calcolo che eseguono programmi.

Ma sappiamo che tutti i problemi sono risolubili per via automatica, e questo porta proprio alla nostra domanda, quali problemi sono in grado di risolvere *automaticamente*?

Ciò che studia il *cosa* dell'informatica si chiama **teoria della calcolabilità**, mostreremo nelle successive lezioni che non è una domanda così facile da rispondere, poichè esistono delle cose che non sono calcolabili. Quindi dato che non ci sono cose calcolabili, la teoria della calcolabilità si domanda *che cosa è calcolabile*?

Nella teoria della calcolabilità vogliamo una risposta generale, non cerchiamo dei casi particolari per dire questo è calcolabile o meno, esistono delle proprietà che accomunano tutto ciò che è calcolabile ? La risposta è sì, la nozione di calcolabilità è denotabile attraverso la matematica e quindi posso affrontare l'insieme di cose fattibili dell'informatica con gli strumenti della matematica.

1.2 Come l'informatica risolve i problemi?

La branca dell'informatica che si chiama **teoria della complessità** risponde alla domanda *come è risolubile questo problema* ?. Essa studia la quantità di risorse computazionali richieste dalla *soluzione automatica* dei problemi. Una *risorsa computazionale* è qualsiasi cosa venga sprecata per l'esecuzione dei programmi:

Le principali risorse su cui ci concentreremo sono il **tempo** e **spazio di memoria**, dovremo quindi dare una definizione rigorosa di queste risorse computazionali. Successivamente si potrà porre delle domande ovvie : *quale è la classe di problemi che vengono risolti efficientemente in termini di tempo e di memoria* ?. Notare che nella teoria della complessità non si parla solo di **risolubilità** (come nella teoria della calcolabilità) ma anche dell'**efficienza** con cui risolvo questo problema.

Esistono problemi che si trovano in una zona grigia, che non sappiamo se hanno una soluzione efficiente, ma sono problemi molto importanti, nessuno è riuscito a dimostrare se avranno soluzioni efficienti ma nemmeno il contrario, ovvero nessuno è riuscito a dimostrare che saranno risolubili efficientemente.

Questa classe di problemi è la classe di problemi NP, vedremo poi di cosa si tratta.

2 Syllabus

- Teoria della calcolabilità: individuare la qualità della calcolabilità dei problemi, quali sono le categorie di problemi calcolabili e distinguerla da quella dei problemi non calcolabili.
- Teoria della complessità: studio quantitativo dei problemi, dopo aver delimitato il confini di ciò che è calcolabile cercare un sotto insieme di problemi **efficientemente calcolabili**.

3 Il nostro linguaggio: la matematica

3.1 Funzione

Una funzione dall'insieme A all'insieme B è una **legge**, che chiamiamo solitamente f , che spiega come associare ad ogni elemento di A un elemento di B .

Dal punto di vista formale l'espressione della funzione viene definita **globalmente**:

$$f : A \rightarrow B$$

Dove A viene chiamato **dominio** e B **codominio**, questa notazione dice che ogni elemento del dominio è associato attraverso una legge f ad un elemento del codominio. Esiste anche un'notazione che permette di stabilire localmente l'operato della funzione, essa rappresenta l'operato della legge f sull'elemento a che porta all'elemento b .

$$a \xrightarrow{f} b$$

La notazione comunemente più utilizzata (in particolare nei libri di testo, ma anche nei corsi di matematica), è la seguente:

$$f(a) = b$$

Solitamente b è l'**immagine** di a secondo f , e meno usualmente si dice che a è la **controimmagine** di b (sempre secondo f).

Per esempio:

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

Dove \mathbb{N} è l'insieme dei numeri naturali $0, 1, 2, 3, \dots$, e per utilizzi futuri denotiamo con \mathbb{N}^+ l'insieme dei numeri naturali positivi (zero escluso) $1, 2, 3, 4, \dots$

Ora vediamo la specifica della funzione f

$$f(n) = \lfloor \sqrt{n} \rfloor$$

Considerando $n = 5$ l'immagine di quest'ultima sarà $f(5) = \lfloor \sqrt{5} \rfloor = 2$.
Quindi possiamo dedurre che per più elementi del dominio una **funzione** effettua un mapping ad uno ed un solo valore del codominio (nel caso in cui un valore del dominio venga mappato a più valori del codominio allora si parla di relazione, ma non più di funzione).

3.2 Classi di funzioni

3.2.1 Funzioni iniettive

Una funzione è **iniettiva** se e solo se elementi diversi del dominio vengono mappati in elementi diversi del codominio.

$$f : A \rightarrow B \text{ è iniettiva sse } \forall a_1, a_2 \in A \text{ dove } a_1 \neq a_2 \implies f(a_1) \neq f(a_2)$$

Esempio 1

$$f(n) = \lfloor \sqrt{n} \rfloor$$

Abbiamo visto precedentemente che questa funzione non è iniettiva, perché avviene un mapping per più elementi del dominio ad un unico elemento del codominio.

Esempio 2

$$f(n) = [n]_2$$

Questa funzione è fortemente non iniettiva, le due metà dell'insieme dei numeri naturali vengono mappate solamente su due numeri $f(2k) = 0$ e $f(2k+1) = 1$.

Esempio 3

$$f(n) = n^2$$

Questa è una funzione iniettiva, poiché ad ogni controimmagine corrisponde una immagine distinta.

3.2.2 Funzioni suriettive

Una funzione è suriettiva quando tutti gli elementi del codominio hanno una corrispondenza con un elemento del dominio.

$$f : A \implies B \text{ sse } \forall b \in B, \exists a \in A : f(a) = b$$

Esempio 1

$f(n) = \lfloor \sqrt{n} \rfloor$ è suriettiva, questo perchè $\forall m \in \mathbb{N}, m = \lfloor \sqrt{m^2} \rfloor = f(m^2)$. Sostanzialmente, posso tornare con facilità al dominio perchè mi basta elevare al quadrato l'immagine, e questo è fattibile per tutto l'insieme dei numeri naturali.

Esempio 2

$f(n) = [n]_2$ non è una funzione suriettiva, questo perchè per esempio 3 non è immagine di niente rispetto a f (il codominio è tutto).

3.3 Insieme immagine di una funzione

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

Data f definiamo l'**insieme immagine di f** come gli elementi del codominio $\in B$ che sono immagine di un elemento del dominio A .

La relazione tra questo insieme Im_f ed il codominio stesso di f quale è B , consiste in:

$$Im_f \subseteq B$$

Allora possiamo dire che una funzione è suriettiva quando:

$$Im_f = B$$

Esempi

$$Im_{\lfloor \sqrt{n} \rfloor} = \mathbb{N} \implies f(n) = \lfloor \sqrt{n} \rfloor \text{ è suriettiva}$$

$$Im_{[n]_2} = 0, 1 \subseteq \mathbb{N} \implies f(n) = [n]_2 \text{ non è suriettiva}$$

3.4 Funzioni biettive

Una funzione si dice biettiva quando è sia suriettiva che iniettiva, devono valere entrambe le due condizioni (questo due condizioni è possibile fonderle in un'unica condizione).

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

$$\forall a_1, a_2 \in A, a_1 \neq a_2 \implies f(a_1) \neq f(a_2) \wedge \forall b \in B, \exists a \in A : f(a) = b$$

Che converge in un'unica definizione:

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

Per esempio: $f(n) = n$, oppure considerando gli insiemi dei numeri reali $f(x) = x^3$. Solo per questa tipologia di funzioni esiste il concetto di **funzione inversa**.

3.5 Inversa di una funzione

Data una funzione f biettiva si definisce l'inversa come f^{-1} la funzione tale che crei un mapping tra l'immagine del codominio rispetto alla controimmagine del dominio.

$$f : A \rightarrow B$$

$$f^{-1} : B \rightarrow A \text{ tale che } f^{-1}(b) = a \iff (a) = b$$

Per esempio l'inversa di $f(n)$ è $f^{-1} = n$, oppure l'inversa di $f(x) = x^3$ è $f^{-1} = \sqrt[3]{x}$ (considerando l'insieme dei numeri reali).

3.6 Composizione di funzioni

Date due funzioni $f : A \rightarrow B$ e $g : B \rightarrow C$, notiamo che queste funzioni hanno una caratteristica in comune, ovvero che il codominio di una è il dominio dell'altra. Definiamo la composizione di funzione $g \circ f : A \rightarrow C$ come la funzione che va da dal dominio di f al codominio di g , definita come $g(f(a))$.

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

Per esempio $f(n) = n + 1$ e $g(n) = n^2$:

- f composto $g : g \circ f(n) = (n + 1)^2$
- g composto $f : f \circ g(n) = n^2 + 1$

N.B. L'operazione di composizione restituisce una funzione, e l'operatore \circ non è commutativo, però quando dominio e codominio lo permettono è **associativo**: $(f \circ g) \circ h = f \circ (g \circ h)$.

3.6.1 Funzione identità

La funzione identità sull'insieme A è una funzione che effettua un mappaggio ricorsivo sullo stesso elemento.

$$i_A : A \rightarrow A : i_A(a) = a \quad \forall a \in A$$

Per esempio la funzione identità sull'insieme \mathbb{N} è $i_{\mathbb{N}}(n) = n$.

3.6.2 Definizione alternativa di funzione inversa

Data una funzione $f : A \rightarrow B$ biettiva, la sua inversa è l'unica funzione $f^{-1} : B \rightarrow A$ che soddisfa:

$$f^{-1}(b) = a \longleftrightarrow f(a) = b$$

oppure

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

Infatti considerando $f^{-1} \circ f(x) = \sqrt[3]{x^3} = x = i_{\mathbb{N}}(x)$ e $f \circ f^{-1}(x) = (\sqrt[3]{x})^3 = x = i_{\mathbb{N}}(x)$

3.6.3 Funzioni totali e parziali

Considerando una funzione $f : A \rightarrow B$ essa è una legge che ad **ogni** elemento di A si associa un elemento di B , questo significa che ogni immagine $f(a)$ è definita per ogni elemento $a \in A$. Esiste un apposita notazione:

$$f(a) \downarrow \forall a \in A$$

Una funzione di questo tipo viene chiamata **totale** poiché risulta definita sulla totalità del suo dominio.

Certe funzioni potrebbero *non essere definita* per quale che elemento di $a \in A$, e quindi non avere delle immagini corrispondenti, la notazione:

$$f(a) \uparrow$$

Ovvero, che per un elemento a non esiste immagine nell'insieme B tramite la funzione f .

Consideriamo il seguente esempio:

$$f : \mathbb{N} \rightarrow \mathbb{N}$$

$$f(n) = \lfloor \frac{1}{n} \rfloor \text{ non è definita su } n = 0 \implies f(0) \uparrow \forall n \in \mathbb{N} \setminus 0, f(n) \downarrow$$

Una funzione viene definita **parziale** se a *qualche* elemento di A si associa un elemento di B . Si amplia un nuovo concetto che è quello del **dominio** (o campo di esistenza) della funzione, ovvero quell'insieme costituito da tutti gli elementi di A tali per cui è definita una immagine appartenente a B .

$$\text{Dom}_f = \{a \in A : f(a) \downarrow\} \subseteq A$$

Allora vale precisare le due seguenti regole:

$$\text{Dom}_f \not\subseteq A \implies f \text{ parziale}$$

$$\text{Dom}_f \equiv A \implies f \text{ totale}$$

Alcuni esempi:

$$f(n) = \left\{ \frac{1}{n} + \frac{1}{(n-1)(n-2)} \implies \text{Dom}_f = \mathbb{N} \setminus \{0, 1, 2\} \right\}$$

$$f(n) = \lfloor \log n \rfloor \implies \text{Dom}_f = \mathbb{N} \setminus \{0\}$$

$$f(n) = \lfloor \sqrt{-n} \rfloor \implies \text{Dom}_f = \{0\}$$

3.6.4 Totalizzazione di una funzione parziale

Teniamo conto di una cosa, possiamo convenzionalmente rendere totale una funzione parziale, basta estendere il codominio con un **simbolo convenzionale** \perp che buttiamo fuori tutte le volte che la funzione non è definita.

$$f : A \rightarrow B \text{ parziale} \implies \tilde{f} : A \rightarrow B \cap \{\perp\}$$

La totalizzazione di f viene raggiunta con l'aggiunta di questo simbolo.

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

Quindi per i punti dove il campo di esistenza non è definito verrà restituito \perp , per convenzione quando una funzione parziale viene totalizzata ovvero $B \cap \perp$ possiamo utilizzare la seguente notazione B_{\perp} .

3.6.5 Prodotto cartesiano

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

Il **prodotto cartesiano** di due insiemi è l'insieme di coppie dove il primo elemento della coppia appartiene al primo insieme, ed il secondo elemento della coppia appartiene al secondo insieme.

Il prodotto cartesiano è un'operazione che non commuta.

$$A \times B \neq B \times A$$

Ovviamente l'unico caso dove un prodotto cartesiano è commutativo è quando $A \equiv B$. Il prodotto cartesiano può essere esteso al prodotto di ennuple di più insiemi cartesiani, dove questa volta il risultato è costituito da un insieme ordinato (non più di coppie) delle ennuple rispettive agli insiemi di provenienza:

$$A_1 \times A_2 \times \dots \times A_n = \{(a_1, a_2, \dots, a_n) : a_i \in A_i\}$$

Associato alla definizione di prodotto cartesiano abbiamo anche quella di **proiettore i-esimo**, essa è una funzione che agisce su un prodotto cartesiano. Ha come dominio l'insieme i-esimo di questo prodotto data una tupla del prodotto cartesiano non fa altro che estrarre la componente i-esima di quella tupla (*destrutta la tupla*).

$$\pi_i : A_1 \times \dots \times A_n \rightarrow A_i$$

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

Utilizzeremo la seguente notazione esponenziale per calcolare il prodotto cartesiano di un insieme cartesiano con se stesso:

$$A_1 \times A_2 \times A_3 \dots \times A_n = A^n$$

Alcuni esempi:

$$C = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 = 1\} \implies \text{punti che si trovano lungo la circonferenza}$$

$$I = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 < 1\} \implies \text{punti che si trovano all'interno della circonferenza}$$

$$E = \{(x, y) \in \mathbb{R}^2 : x^2 + y^2 > 1\} \implies \text{punti che si trovano all'esterno della circonferenza}$$

$$C \cap I \cap E = \mathbb{R}^2$$

3.6.6 Insieme di funzioni

L'insieme delle funzioni che vanno dall'insieme A a B viene indicato con:

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

L'insieme delle funzioni *parziali* che vanno da A a B :

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

3.6.7 Funzione di valutazione

Dati due insiemi A, B e B_{\perp}^A si definisce una funzione di valutazione come:

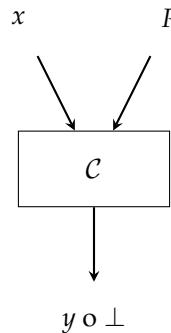
$$\omega : B_{\perp}^A \times A \rightarrow B \text{ con } \omega(f, a) = f(a)$$

Essa è una funzione che valuta il punto del codominio A in termini di B , ovvero restituisce un $f(a)$, quindi il suo compito è strettamente quello di valutare.

- Tenendo fisso a e facendo variare f , è come se a fosse un *benchmark* su cui testiamo una serie di funzioni.
- Tenendo fisso f e facendo variare a ottengo il grafico di f .

3.7 Modellare teoricamente un sistema di calcolo

Un sistema di calcolo è architettato come un architettura di Von Neumann, è un sistema che dato in input un *dato* x ed un *programma* P . L'output può essere indicato con y quando è definito, mentre con \perp quando va in *loop*.



$P \in PROG$: Un programma è una **sequenza di regole** scritte in un certo linguaggio che prende un input e lo trasforma in un output (o in un loop). Essenzialmente un programma è la definizione di una funzione scritta a mano, esso realizza una funzione che parte dai dati in input ed ottiene i dati in output.

$P \in DATI_{\perp}^D ATI$ è una funzione in un linguaggio di programmazione.

Quindi, ribadendo per l'ennesima volta un programma è la realizzazione di una funzione, che va da un insieme di dati ad un altro.

Il sistema di calcolo in se prende in input dei dati ed una funzione (il programma), che mi da un output, esso è definito come una **funzione di valutazione** \mathcal{C} .

$$\mathcal{C} : DATI_{\perp}^{DATI} \times DATI \rightarrow DATI_{\perp}$$

Quindi \mathcal{C} è una funzione di valutazione, e $\mathcal{C}(P, x)$ è la funzione di valutazione del programma P sul dato x .

Per esempio, consideriamo il seguente programma:

```
Input: x
if ⟨x⟩2 == 0 then
|   return x;
else
|   while 1 < 2 do ;
|   |   return 1;
end
```

Algorithm 1: Semantica di P

Voglio capire la semantica di questo programma è per capirlo voglio definire una funzione che mi rappresenti il legame input-output rispetto ad un dato elemento.

Il programma prende in ingresso un numero intero, se il numero è pari restituisce esattamente il numero intero, altrimenti entra in un loop (non termina).

$$\mathcal{C}(P, \cdot) : \mathbb{N} \rightarrow \mathbb{N}_\perp$$

La **semantica** è la seguente :

$$\mathcal{C}(P, n) = \begin{cases} n & \text{se } n \text{ è pari} \\ \perp & \text{altrimenti} \end{cases}$$

3.7.1 Potenza computazionale

I sistemi di calcolo servono per calcolare le funzioni, ogni programma che immetto nel mio sistema di calcolo mi viene calcolato. Indico con potenza computazionale del mio sistema di calcolo è *tutto quello che sa fare* il mio sistema di calcolo, o meglio, tutte quelle funzioni che il mio sistema di calcolo può calcolare.

Tutte le funzioni che può calcolare il mio sistema di calcolo è un sottoinsieme di tutte le funzioni immaginabili che possono andare da dati in dati. I programmi in generale sono delle funzioni da dati in dati, quindi il mio sistema di calcolo è ovvio che calcoli questo tipo di funzioni, è un sottoinsieme perchè non so quali funzioni è in grado di risolvere il mio sistema di calcolo (e questa è la domanda a *cosa* a cui risponde l'informatica teorica).

$$F(\mathcal{C}) = \{\mathcal{C}(P, \cdot) : P \in PROG\} \subseteq DATI_\perp^{DATI}$$

Questo significa che esistono delle funzioni che non possono essere risolte dal mio sistema di calcolo, e che quindi **non sono automatizzabili**.

$$F(\mathcal{C}) \not\subseteq DATI_\perp^{DATI}$$

Sono presenti funzioni che l'informatica può risolvere e fare tutto.

$$F(\mathcal{C}) = DATI_{\perp}^{DATI}$$

Quindi abbiamo ridotto il quesito *che cosa sono in grado di fare i programmi*, ad un quesito matematico di inclusione propria ed impropria.

3.7.2 Importanza del calcolo di funzione

Calcolare una funzione significa risolvere problemi in *generale*. Questo perché ad ogni problema posso associare una *funzione soluzione*, non è altro che quella funzione che ad un dato input associa una determinata *soluzione*. Per esempio, il *problema del calcolo del determinante*:

DET: **Input:** $M \in \mathbb{R}^{n \times n}$
Output: Determinante di M

La rispettiva funzione soluzione:

$$f_{DET} : \mathbb{N}^{n \times n} \rightarrow \mathbb{Z}$$

$$f_{DET}(M) = Det(M) :$$

Vediamo che risolvere il dato problema consiste nel calcolare la funzione soluzione, o risolvere il problema significa sostanzialmente avere un programma in grado di risolvere quella funzione.

Find-Replace: **Input:** Testo, parola da cercare, parola da sostituire
Output: Testo in cui ogni occorrenza della parola da cercare è sostituita dall'altra

La funzione soluzione:

$$f_{Find-Replace} : TESTI \times PAROLE \times PAROLE \rightarrow TESTI$$

$$f_{Find-Replace}(\text{La nostra vita,nostra,vostra}) = \text{La vostra vita}$$

3.7.3 Cardinalità degli insiemi - I°

Siamo arrivati al primo punto fondamentale, il *cosa* dell'informatica, ed abbiamo visto che si concretizza nell'interrogarsi sulla relazione tra l'insieme della potenza computazionale e quello di tutte le funzioni che vanno da dati in dati possibili.

$$F(\mathcal{C}) ? DATI_{\perp}^{DATI}$$

Per cercare di dare una dimostrazione sul carattere dell'inclusione, è molto utile fare riferimento al concetto matematico di **cardinalità**.

Dato un insieme A , la sua cardinalità si indica con $|A|$. Intuitivamente che la cardinalità di un insieme è il numero di elementi da cui è formato l'insieme. Questa idea intuitiva è abbastanza corretta quando si ha a che fare con insiemi finiti (la cardinalità mi può aiutare a capire quale insieme è più grande degli altri).

Purtroppo però quando tiriamo in ballo insiemi di cardinalità infinita, questo potrebbe portare a conclusioni più complicate da elaborare. Si potrebbe *erroneamente* pensare $|\mathbb{N}| = \infty = |\mathbb{B}|$, ma questo sappiamo che non è assolutamente vero, perché l'insieme dei numeri reali è molto più fitto di quello dei numeri interi.

Quindi dobbiamo aggiungere dei nuovi concetti, visto che "*l'infinito di \mathbb{R}* " è diverso da quello di \mathbb{N} . Il concetto da introdurre è quello di **relazione**.

3.7.4 Relazione

Consideriamo un insieme A , si definisce **relazione binaria** R su A , il sottoinsieme del prodotto cartesiano di A su se stesso.

$$R \subseteq A^2$$

Gli elementi $a, b \in A$ stanno in una relazione R se e solo se $(a, b) \in R$. Sono presenti due notazioni infisse per denotare l'esistenza della relazione tra i due elementi:

$$a R b \text{ oppure } a \mathcal{R} b$$

Consideriamo per esempio la relazione R come la relazione che agisce sui numeri naturali, tale che un numero divida l'altro.

$$R \equiv \text{divide} : 3 R 6, 5 R 45, \dots, 3 \mathcal{R} 10$$

$$R = \{(a, b) \in \mathbb{N}^2 : \langle b \rangle_a = 0\}$$

Introduciamo anche il concetto di *equivalenza in modulo k*, la quale mi descrive una relazione del tipo:

$$a \equiv_k b \text{ sse } \langle a \rangle_k = \langle b \rangle_k$$

Per esempio: $5 \equiv_2 7, 4 \equiv_2 16, \dots$ (quando il resto dato dal divisore k è uguale su entrambi gli operandi).

Parliamo di **relazione di equivalenza** se e solo se soddisfa le seguenti proprietà:

- **Riflessiva:** $\forall a \in A, a R a$
- **Simmetrica:** $\forall a, b, a R b \Leftrightarrow b R a$
- **Transitiva:** $\forall a, b, c, a R b \wedge b R c \implies a R c$

Ora considerando le precedenti relazioni, vediamo che la relazione $R \equiv$ "divide" non è una relazione di equivalenza:

- È riflessiva.
- **Non è simmetrica:** $3 R 6$ ma $6 \mathcal{R} 3$.
- È transitiva.

Mentre per la seconda relazione \equiv_k possiamo dire che essa è una relazione di equivalenza:

- È riflessiva.
- È simmetrica, vengono valutati i moduli non direttamente gli operandi.
- è transitiva (per lo stesso motivo ancora).

Con una relazione di equivalenza è possibile effettuare un **partizionamento delle classi di equivalenza**.

3.7.5 Relazioni di equivalenza e partizioni

Considerando una relazione di equivalenza $R \subseteq A^2$ induce una **partizione** su A . Le partizioni sono dei sottoinsiemi $A_1, A_2, A_3, \dots \subseteq A$ tali che:

- $A_i \neq \emptyset$ (non sono vuoti).
- $i \neq j \implies A_i \cap A_j = \emptyset$ (non sono sovrapponibili).
- $\bigcup_{i=1} A_i = A$ (la loro unione ricompone A).

Una **classe di equivalenza** di un elemento $a \in A$ è l'insieme di tutti gli elementi che sono in relazione con a , notazione:

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

Si dimostra facilmente che :

- Non esistono classi di equivalenza vuote, questo per via della proprietà *riflessiva* delle relazioni di equivalenza.
- Se prendo due elementi diversi del dominio le classi di equivalenza relative o sono disgiunti o sono la stessa classe di equivalenza: $a, b \in A$ vale che $[a]_R \cap [b]_R = \emptyset$ o $[a]_R = [b]_R$.
- $\bigcup_{a \in A} [a]_R = A$

Quindi la partizione indotta da una relazione di equivalenza non è altro che l'insieme delle classi di equivalenza relative a quella relazione.

L'insieme delle classi di equivalenza di R è la partizione indotta da R su A .



Figura 3.1: Insieme A partizionato

L'insieme A partizionato rispetto alla relazione R è detto **insieme quoziente**:

$$A/R$$

Quindi il quoziente di A rispetto a R è lo "spezzettamento" di A in classi di equivalenza.

Consideriamo il seguente esempio di insieme quoziente, consideriamo la relazione di equivalenza $\equiv_4 \subseteq \mathbb{N}$. *Quali classi di equivalenza ammette questa relazione?*

$$[0]_4 = 4k \text{ tutti i multipli di 4 hanno lo stesso resto di 1}$$

$$[1]_4 = 4k + 1 \text{ tutti i multipli di 4 aumentati di 1 hanno lo stesso resto 1}$$

$$[2]_4 = 4k + 2; [3]_4 = 4k + 3; \dots$$

Ne esistono altre? No, non esistono altre classi di equivalenza perché ogni caso successivo a quelli "base" si riconduce a quelli "base".

$$\langle n \rangle_2 \in \{0, 1, 2, 3\}$$

Voglio vedere se queste classi di equivalenza può rappresentare una partizione:

- Nessuna classe è vuota.
 - Sono mutuamente disgiunte, poiché se prendo un numero esso ricadrà solamente in una di queste classi di equivalenza.
 - L'unione di queste 3 classi di equivalenza restituisce l'insieme originale
- $$\bigcup_{i=0}^3 [i]_4 = \mathbb{N}.$$

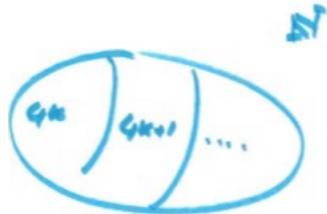


Figura 3.2: Insieme quoziente di \equiv_4

$\{[i]_4 : 0 \leq i \leq 3\}$ è una partizione di \mathbb{N} indotta dalla relazione \equiv_4 , tale per cui mi dia il rispettivo insieme quoziente (a volte impropriamente chiamato \mathbb{Z}_4):

$$N/\equiv_4 = \{[i]_4 : 0 \leq i \leq 3\} = \mathbb{Z}_4$$

Adesso abbiamo in mano gli strumenti per definire in maniera molto più fine il concetto di cardinalità.

3.7.6 Cardinalità degli insiemi - II°

Sia \mathcal{U} (insieme universo) la classe di tutti gli insiemi, definiamo la relazione $\sim \subseteq \mathcal{U}^2$ detta relazione di *equi numerosità* (hanno la stessa dimensione numerica), tra le coppie degli insiemi, se e solo se esiste una **biezione** tra A e B (ovvero, se riesco ad esibire una funzione iniettiva e suriettiva che va da A in B).

Questa relazione tra insiemi è una relazione di equivalenza, poiché:

- \sim è riflessiva, se utilizzo la funzione identità i_A .
- \sim è simmetrica, se esiste una biezione $A \rightarrow B$ allora esiste una biezione anche $B \rightarrow A$. Ovvero $A \sim B$ e $B \sim A$.
- \sim è transitiva, se compongo funzioni biettive ottengo ancora una funzione biettiva.

Due insiemi che stanno in questa relazione vengono detti *equi numerosi*. Ora considerando l'insieme quoziante del nostro universo \mathcal{U} rispetto alla relazione di equi numerosità \sim (quindi stesso numero di elementi in entrambi i due insiemi), esso mi rappresenta il concetto di **cardinalità** di un insieme.

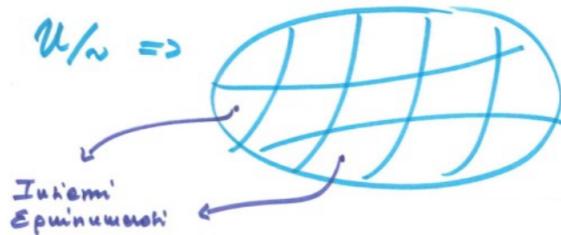


Figura 3.3: Insieme quoziante \mathcal{U}/\sim

Quindi spezzettando l'insieme \mathcal{U} in classi di equivalenza in questa classe di equivalenza ci sono tutti gli insiemi che sono equi numerosi (seppur diversi).

Questo concetto permette di parlare in maniera molto precisa anche di insiemi di cardinalità infinita. Questi insiemi possono avere lo stesso numero

Per esempio, consideriamo $n \in \mathbb{N}^+$, si consideri l'insieme $J_n = \{1, 2, \dots, n\}$. Per questo insieme è ovvio quale sia il concetto di cardinalità perché è finito.

Allora diciamo che un insieme A ha cardinalità **finita** se è equi numeroso con J_n (ovvero $A \sim J_n$) per un dato n ed in quel caso scriviamo che $|A| = n$.

Quindi nella classe di equivalenza J_1 troviamo tutti gli insiemi con un elemento, nella classe di equivalenza di J_2 troveremo tutti gli insiemi con due elementi, ecc.

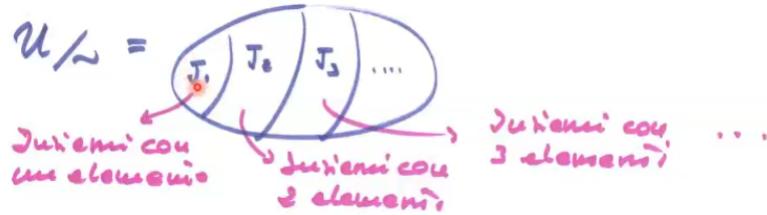


Figura 3.4: Esempio \mathcal{U}/\sim rispetto all'insieme J_n

Un insieme che non ha cardinalità si dice banalmente che ha cardinalità **infinità**. Fin qui abbiamo parlato ancora di cardinalità finita, ma adesso faremo un esempio con cardinalità infinita.

3.7.7 Insiemi numerabili

A si dice **numerabile** se è nella stessa classe di equivalenza dell'insieme dei numeri naturali \mathbb{N} , ovvero se esiste una *biezione* tra l'insieme A e l'insieme \mathbb{N} .

Siccome stanno in relazione di equi numerosità vuol dire che è presente una biezione fra i due elementi, quindi due elementi non possono collidere.

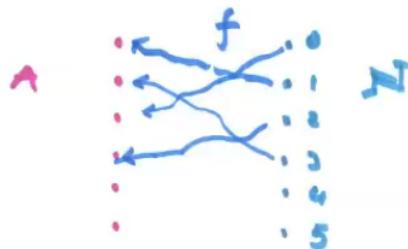


Figura 3.5: A è un insieme numerabile

La presenza di questa biezione significa che come A può essere **listato** con $f(0), f(1), f(2), \dots$ su \mathbb{N} senza perdere un elemento, anche l'insieme \mathbb{N} può essere listato a sua volta su A .

Alcuni esempi di insiemi numerabili:

- I numeri pari $f(n) = 2n$, essi sembrerebbero la metà dei numeri naturali ma sono tanti quanto i numeri naturali perché esiste questa biezione (vale anche per i dispari $f(n) = 2n + 1$).
- L'insieme \mathbb{Z} , possono essere presenti diverse funzioni, per esempio posso mappare i negativi sui numeri pari ed i positivi sui numeri dispari.
- L'insieme \mathbb{Q} , vedremo più avanti.

- L'insieme delle stringhe binarie che incominciano con 1, ovvero $1\{0,1\}^*$, dove una funzione $f(n) = bim(n)$ è in grado di associare un numero naturale (utilizzando le potenze in posizione) ad una stringa binaria.

3.8 Insiemi non numerabili

Esistono degli insiemi che non sono listabili, dette in altre parole sono più fitti di \mathbb{N} , è un altro tipo di categoria di infinito.

\mathbb{R} non è numerabile

Dimostrazione

L'idea consiste nel dimostra per assurdo che non esiste una biezione perché sono presenti dei "buchi" tra le associazioni. Ordine della dimostrazione:

- Dimostro che $\mathbb{R} \sim [0, 1]$ (si dice che è *isomorfo/equi numeroso* all'intervallo $[0, 1]$), ovvero che è fitto quanto l'intervallo citato.
- Dimostro che $\mathbb{N} \sim [0, 1]$
- $\mathbb{R} \sim [0, 1] \sim \mathbb{N} \implies \mathbb{R} \sim \mathbb{N}$

Dimostrazione $R \sim [0, 1]$

Significa riuscire a rappresentare una funzione che mappa gli elementi tra i due insiemi in maniera che sia suriettiva ed iniettiva.

Prima di tutto abbiamo una retta cartesiana con un origine fissata e che copre tutti i numeri reali. Pongo l'intervallo $[0, 1]$ in modo che si trovi in corrispondenza del punto mediano della retta 0, successivamente prendo un compasso punto il centro nel mediano dell'intervallo e traccio la semi circonferenza rispetto alla metà dell'intervallo.

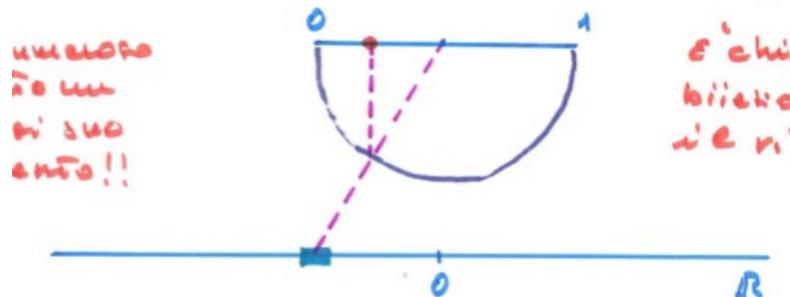


Figura 3.6: Dimostrazione $R \sim [0, 1]$

Gli elementi dell'insieme \mathbb{R} vengono mappati tracciando una retta in direzione del punto mediano di $[0, 1]$, nel punto di intersezione si traccia la retta

perpendicolare che interseca l'intervallo ed in quel punto si trova il valore corrispondente. Si può fare lo stesso in maniera opposta partendo da $[0, 1]$, trovando l'intersezione si parte dal punto mediano di quest'ultima e si attraversa l'intersezione toccando \mathbb{R} .

Questo dimostra che tutti i valori di \mathbb{R} sono associabili su $[0, 1]$, quindi $\mathbb{R} \sim [0, 1]$.

Dimostrazione $\mathbb{N} \not\sim [0, 1]$ (per diagonalizzazione)

Supponiamo per assurdo che $\mathbb{N} \sim [0, 1] \implies [0, 1]$, ovvero che sia listabile (*elencabile*) in maniera esaustiva così come lo è \mathbb{N} . Siccome sto ipotizzando che sia elencabile, allora creo una lista di tutti gli elementi in $[0, 1]$, i numeri all'interno di questo numero seguono il formato "0.", quindi:

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

Se riesco a costruire un numero che non fa parte di questa lista infinita (elusivo alla biezione), allora vuol dire che la lista non è elencabile, e quindi $[0, 1]$ non è numerabile.

Per costruire questo numero elusivo che non si trova in nessuno di questi elementi della lista, vado a guardare le cifre sulla *diagonale*.

Costruiamo il numero elusivo alla lista

$$0.c_1c_2c_3c_4$$

Tale che rispetti la seguente regola rispetto alla diagonale del elenco dei numeri $\in [0, 1]$

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

Ora usando questa regola non riuscirò mai a posizionare il mio nuovo numero tra quelli elencati, questo perchè ovviamente cambio le cifre (il perché questo accade probabilmente è collegato al fatto che \mathbb{R} è più fitto di \mathbb{N} , proprio quello che vogliamo dimostrare). Il numero $0.c_1c_2\dots \in [0, 1]$, ma non appartiene nelle liste poiché:

- Differisce dal primo perché $c_1 \neq a_{11}$
- Differisce dal secondo perché $c_2 \neq a_{22}$
- Differisce da qualunque numero presente sulla diagonale

Quindi la lista non è esaustiva $\implies \mathbb{N} \not\sim [0, 1]$, significa che non cattura tutto l'intervallo $[0, 1]$, quindi non può esistere una biezione tra questo ed \mathbb{N} , ovvero $[0, 1]$ **non è numerabile**.

Conclusione

$$\mathbb{R} \sim [0, 1] \not\sim \mathbb{N} \implies \mathbb{R} \not\sim \mathbb{N}$$

- \mathbb{R} non è numerabile.
- Esso è più "fitto" di \mathbb{N} .
- Qualsiasi tentativo di listare anche solo un segmento non è esaustivo.
- \mathbb{R} è un insieme **continuo**, e tutti gli insiemi equi numerosi \mathbb{R} si dicono continui.
- Altri esempi di insiemi non numerabili:

3.8.1 Insieme delle parti di \mathbb{N}

Un altro insieme non numerabile è quello costituito dalla famiglia di sottoinsiemi possibili di \mathbb{N} , quindi mettendo assieme uno dopo l'altro tutti i sottoinsiemi di differenti cardinalità (talvolta chiamato **insieme potenza o boleano di \mathbb{N}**). Per esempio su un insieme $S = \{a, b, c\}$, l'insieme delle parti è $\mathcal{P}(S) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, S\}$.

$$\mathcal{P}(\mathbb{N}) = 2^{\mathbb{N}} = \{\text{sottoinsiemi di } \mathbb{N}\} \not\sim \mathbb{N}$$

Anche questa dimostrazione avviene per diagonalizzazione, per assurdo suppongo che esista una biezione che mi permetta di elencare tutti i sottoinsiemi di \mathbb{N} .

Posso rappresentare i sottoinsiemi di \mathbb{N} utilizzando un **vettore caratteristico** $A \subseteq \mathbb{N}$, questo è un vettore dove metto il bit di appartenenza rispetto alla corrispondenza dell'elemento.

$$\begin{aligned} \mathbb{N} &\rightarrow 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ \dots \\ A &\rightarrow 0 \ 1 \ 1 \ 0 \ 1 \ 1 \ 0 \ \dots \end{aligned}$$

Allora se io suppongo per assurdo che $\mathcal{P}(\mathbb{N}) \sim \mathbb{N}$, ovvero che è numerabile rispetto all'insieme dei numeri naturali, allora posso elencare in maniera esaustiva i sottoinsiemi di \mathbb{N} , questo elencandone i vettori caratteristici.

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

Se riesco a costruire un sottoinsieme di \mathbb{N} (vettore caratteristico) che non fa parte da di questa lista, allora l'insieme delle parti non è un insieme numerabile poiché non è equi-numeroso con \mathbb{N} . Questo è possibile procedendo per *diagonalizzazione*, per esempio se considerassi il seguente vettore negato:

$$\overline{b_{01}} \ \ \overline{b_{12}} \ \ \overline{b_{23}} \ \ \overline{b_{34}} \ \ \dots$$

riesco a constatare che esso non appartiene alla lista nonostante sia un sottoinsieme di \mathbb{N} in quanto composto solo da valori booleani.

$$\mathcal{P}(\mathbb{N}) \not\sim \mathbb{N}$$

3.8.2 Insieme $\mathbb{N}^{\mathbb{N}}$

Consideriamo l'insieme di tutte le funzioni possibili funzioni che vanno da \mathbb{N} in \mathbb{N} .

$$\mathbb{N}^{\mathbb{N}} = \{f : \mathbb{N} \rightarrow \mathbb{N}\}$$

Ipotizzando che $\mathbb{N}^{\mathbb{N}} \sim \mathbb{N}$, consideriamo la seguente tabella dove nella prima colonna si trova l'elenco esaustivo di tutte le funzioni $\in \mathbb{N}^{\mathbb{N}}$, e nelle colonne successive tutto il dominio completo di \mathbb{N} . Significa che per ogni riga sarà presente l'insieme dei valori che fornisce il grafico di una funzione f_i .

| Elenco delle funzioni di \mathbb{N} | 0 | 1 | 2 | 3 | $\dots \in \mathbb{N}$ |
|---------------------------------------|----------|----------|----------|----------|------------------------|
| f_0 | $f_0(0)$ | $f_0(1)$ | $f_0(2)$ | $f_0(3)$ | \dots |
| f_1 | $f_1(0)$ | $f_1(1)$ | $f_1(2)$ | $f_1(3)$ | \dots |
| f_2 | $f_2(0)$ | $f_2(1)$ | $f_2(2)$ | $f_2(3)$ | \dots |
| f_3 | $f_3(0)$ | $f_3(1)$ | $f_3(2)$ | $f_3(3)$ | \dots |
| \dots | \dots | \dots | \dots | \dots | \dots |

Allora anche in questo caso per diagonalizzazione voglio costruire una funzione $\in \mathbb{N}^{\mathbb{N}}$ ma elusiva all'elenco delle funzioni della tabella:

$$\varphi : \mathbb{N} \rightarrow \mathbb{N} \text{ come } \varphi(n) = f_n(n) + 1$$

Siamo d'accordo che φ sia ben definita, visto che per ogni immagine corrisponderà un immagine numero naturale, ma notiamo che una qualsiasi $\varphi(n)$ presa sulla lista avrà per forza una immagine della diagonale discordante rispetto a quelle elencate nella tabella.

Quindi abbiamo una funzione elusiva all'elenco, e la nostre ipotesi si rivela assurda:

$$\varphi(0) = f_0(0) + 1 \neq f_0(0)$$

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

N.B.: Gli insiemi $\mathcal{P}(\mathbb{N})$ e $\mathbb{N}^{\mathbb{N}}$ sono detti **insiemi continui**, e sono equinumerosi a \mathbb{R} .

4 Calcolabilità

4.1 Cosa è calcolabile?

Sappiamo che la potenza computazionale di un sistema di calcolo \mathcal{C} corrisponde a:

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

Il nostro problema consiste nel comprendere se questa inclusione sia propria o impropria, per dimostrare ciò posso utilizzare il concetto di cardinalità precedentemente appreso.

Se riesco a dimostrare che il primo insieme ha una cardinalità numerabile ed il secondo ha una cardinalità continua allora il secondo insieme sarà molto più grande del primo.

Prendiamo alcune **assunzioni** che verranno dimostrate successivamente nel corso:

- $PROG \sim \mathbb{N}$, i programmi sono tanti quanti i numeri naturali, questo perché un programma viene digitalizzato in una fila di bit (finiti). Questo vuol dire che per ogni programma corrisponderà un numero che fa parte dei numeri naturali.
- $DATI \sim \mathbb{N}$, i dati sono tanti quanto i numeri, questo per lo stesso motivo di digitalizzazione (o traduzione) in binario delle informazioni.

Possiamo dire che la potenza computazionale sia equinumerosa rispetto al numero di programmi, al variare del programma cambio la funzione di potenza computazionale. A questo punto agganciamo la prima assunzione.

$$F(\mathcal{C}) \sim PROG \sim \mathbb{N}$$

Sapendo che l'insieme dei dati in dati definito come $DATI_{\perp}^{DATI}$, grazie alla seconda assunzione che abbiamo fatto allora possiamo asserire:

$$DATI_{\perp}^{DATI} \sim \mathbb{N}_{\perp}^{\mathbb{N}}$$

Abbiamo visto precedentemente che $\mathbb{N}^{\mathbb{N}} \sim \mathbb{N}$, allora unendo le precedenti dimostrazioni possiamo confermare:

$$F(\mathcal{C}) \sim PROG \sim \mathbb{N} \sim \mathbb{N}^{\mathbb{N}} \sim DATI_{\perp}^{DATI}$$

$$F(\mathcal{C}) \sim DATI_{\perp}^{DATI}$$

Ovvero, che l'insieme dei programmi (corrispondente all'insieme delle funzioni di potenza di calcolo) non è equinumeroso (o *isomorfo*) all'insieme delle funzioni che vanno da dati in dati. In parole povere ho un numero di programmi nettamente inferiore (fitto quanto i numeri naturali) rispetto al numero di funzioni (o problemi) calcolabili (fitto quanto i reali $\mathbb{N}_{\perp}^{\mathbb{N}}$).

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

Vogliamo stabilire una corrispondenza **biunivoca** tra dati e numeri naturali, ovvero che devo riuscire a "nascondere" un dato all'interno di un numero naturale, tale che se lo dovessi riportare all'insieme dei dati otterrei esattamente quel dato (e non un altro).

Questa biezione la voglia rendere effettiva e programmabile, ovvero costruirsi delle primitive che operino direttamente sulla codifica numerica dei

dati così che possiamo smaterializzare completamente il mondo dei dati in quello dei numeri.

Questa volta vogliamo dimostrare che l'**insieme delle coppie** $\mathbb{N} \times \mathbb{N}$ è numerabile. Questo avverrà per passi, prima dimostreremo che l'insieme delle coppie è isomorfo a \mathbb{N}^+ (quindi escluso lo 0) e poi di conseguenza che lo è per tutto \mathbb{N} . Questo per effetto collaterali che anche l'insieme dei razionali \mathbb{Q} è numerabile, questo perché sono una frazione, quindi una coppia di interi.

Dimostrazione $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+$

$$\begin{aligned}\langle , \rangle : \mathbb{N} \times \mathbb{N} &\rightarrow \mathbb{N}^+ \\ \langle x, y \rangle &= n\end{aligned}$$

La **funzione coppia di Cantor** è definita per mezzo delle parentesi angolari, prende due argomenti x, y e restituisce un numero n , questo in maniera iniettiva e suriettiva (biettività).

Ovvero che coppie diverse vengono mappate in numeri diversi, e se ho un numero devo essere in grado di tornare alla stessa coppia x, y . Quindi esibiremo sia l'andata della funzione coppia di Cantor che il ritorno, esibiremo anche i due proiettori *sin* e *des* tali che mi restituiscano i due argomenti in questione.

$$\begin{aligned}sin : \mathbb{N}^+ &\rightarrow \mathbb{N} \text{ e } des : \mathbb{N}^+ \rightarrow \mathbb{N} \\ sin(n) &= x \text{ e } des(n) = y\end{aligned}$$

Vogliamo dare una rappresentazione grafica di questa funzione, Cantor ha pensato di realizzare una tabella che ha come righe e colonne l'insieme \mathbb{N} . In questa tabella il valore della coppia si trova esattamente posizionato all'incrocio tra l' x -esima riga e della y -esima colonna.

| x/y | 0 | 1 | 2 | 3 | 4 |
|-----|----|-----|-----|-----|-----|
| 0 | 1 | 3 | 6 | 10 | 15 |
| 1 | 2 | 5 | 9 | 14 | ... |
| 2 | 4 | 8 | 13 | ... | ... |
| 3 | 7 | 12 | ... | ... | ... |
| 4 | 11 | ... | ... | ... | ... |

La disposizione ingegnosa della tabella evita di andare in un elenco infinito, nel senso che se dovessi elencare tutti i numeri naturali andando verso il basso (o destra) non riuscirei a finire mai, ed inoltre non sarebbe un buon approccio visuale per trovare la corrispondenza con l'altro insieme. I numeri sono disposti in maniera che il successivo si trovi sulla diagonale (andando verso l'alto).

$$\langle 2, 1 \rangle = 8$$

Con questa rappresentazione, coppie diverse non riusciranno mai a confluire nella stessa casella (coordinate di punti che individuano un punto univoco), e

viceversa (iniettiva), questo mi conferma che la funzione è biettiva.
Adesso mettiamo da parte la forma tabellare e consideriamo la **forma analitica** di $\langle \cdot, \cdot \rangle$.

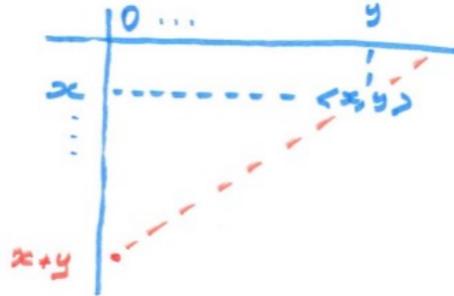


Figura 4.1: Diagonale che passa per $\langle x, y \rangle$

Notiamo che per il valore che voglio valutare attraverso la funzione di Cantor passa una diagonale, questa diagonale parte dalla colonna 0 e dalla riga $x + y$ (il punto $\langle x + y, 0 \rangle$). La funzione coppia nell'origine della diagonale assumerà un certo valore, e visto che i numeri sulla diagonale incrementano in maniera crescente di una sola unità, allora potrò raggiungere $\langle x, y \rangle$ sommando y .

A questo punto il mio problema si riduce a trovare delle coppie particolari ovvero, la cui seconda coordinata è 0, le chiamiamo $\langle z, 0 \rangle$. Notiamo nella prima colonna c'è una correlazione con gli elementi di x , e vediamo che l'elemento i -esimo della prima colonna corrisponde alla somma tra il corrispettivo elemento x e l'elemento $i - 1$ della colonna, allora riusciamo ad esprimere una **legge matematica** per descrivere questa *codifica*.

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

Allora (1) + (2):

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

Come tornare da \mathbb{N}^2 a \mathbb{N}^+ Ovvero, come trovare le funzioni *sin* e *des*.

$$\langle x, y \rangle = n \longrightarrow \text{sin}(n) = x \text{ e } \text{des}(n) = y$$

Partiamo da un elemento n presente nell'intersezione dei due insiemi definiti da x e y e vogliamo individuare le due relative componenti. Procedo trovando le coordinate dell'origine della diagonale che passa per n , ovvero $\langle \gamma, 0 \rangle$.

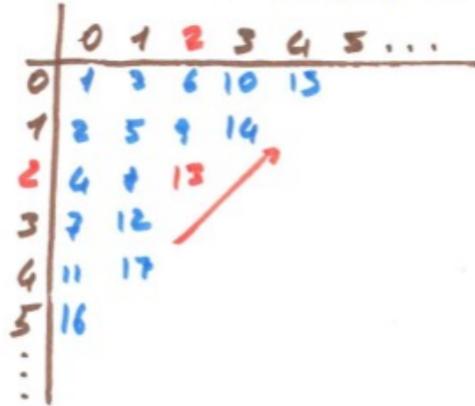


Figura 4.2: $n = 13$

Considerando la dimostrazione precedente possiamo dire che $y = n - \langle \gamma, 0 \rangle$, ovvero procedo a ritroso sulla diagonale partendo da n ed andando verso $\langle \gamma, 0 \rangle$.

Allora, sapendo che $\gamma = x + y$ posso trovare facilmente le parti destre e sinistre.

Il problema principale ora è trovare γ , questo non è altro che l'intero più grande valore tale per cui l'origine della diagonale sia minore di n .

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

Quindi dobbiamo risolvere la diseguaglianza la quale ci darà un range su n , all'interno di questo range $\langle z, 0 \rangle \leq n$ andiamo a prendere l'intero più grande che soddisfa questa diseguaglianza.

$$\begin{aligned} \langle z, 0 \rangle \leq n &\longrightarrow \frac{z(z+1)}{2} + 1 \leq n \longrightarrow z^2 + z + 2 - 2n \leq 0 \\ z_{1,2} &= \frac{-1 \pm \sqrt{8n-7}}{2} \end{aligned}$$

Quindi devo scoprire quale è l'intero più grande dati:

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

l'elemento a destra è il più grande valore, ma non so se è intero, quindi dovrò correggere ciò:

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

Questa è una formula che utilizza la variabile in input n , quindi riusciamo a ricavare il parametro sinistro e destro della funzione di Cantor e ritornare allo

stato iniziale:

$$des(n) = y = n - \langle \gamma, 0 \rangle \text{ e } sin(n) = x = \gamma - y$$

Esempio andata e ritorno tra \mathbb{N}^2 e \mathbb{N}^+

$$\text{Andata : } \mathbb{N}^2 \longrightarrow \mathbb{N}^+$$

$$\langle 10, 20 \rangle = \frac{30 \cdot 31}{2} + 20 + 1 = 15 \cdot 31 + 21 = 485 + 21 = 486$$

$$\text{Ritorno : } \mathbb{N}^+ \longrightarrow \mathbb{N}^2$$

$$\gamma = \left\lfloor \frac{-1 + \sqrt{8 \cdot 436 - 7}}{2} \right\rfloor = \lfloor 30.6448 \rfloor = 30$$

$$y = 486 - \langle 30, 0 \rangle = 486 - \left(\frac{30 \cdot 31}{2} + 1 \right) = 486 - 466 = 20 = des(486)$$

$$x = 30 - 20 = 10 = sin(486)$$

Adesso noi abbiamo dimostrato che $\mathbb{N}^2 \sim \mathbb{N}^+$.

Dimostrazione $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$ Definisco una nuova funzione:

$$[,] : \mathbb{N} \times \mathbb{N} \text{ tale che } [x, y] = \langle x, y \rangle - 1$$

Adesso ho una funzione esplicita per mappare \mathbb{N}^2 su \mathbb{N} .

Nota: A questo punto otteniamo che l'insieme dei razionali \mathbb{Q} sono coppie di interi (*num, den*). Dunque $[,]$ mostra che \mathbb{Q} è numerabile.

Dimostrazione rigorosa che DATI $\cong \mathbb{N}$ Mostriamo le nozioni appena apprese sulle principali strutture dati.

Liste d'interi

$$\text{codifichiamo } x_1, x_2, \dots, x_n \rightarrow \langle x_1, x_2, \dots, x_n \rangle$$

Quindi il numero che rappresenta la codifica è indicato con il numero all'interno delle parentesi angolari. L'idea consiste nell'incapsulare il risultato di una coppia di elementi come elemento destro della coppia più esterna. Non sapendo quanto sia lunga la lista dovrò porre un elemento che mi segnali il termine, e questo lo posso fare con $\langle x_n, 0 \rangle$.

$$\langle x_1, x_2, \dots, x_n \rangle = \langle x_1, \langle x_2, \langle \dots \langle x_n, 0 \rangle \dots \rangle \rangle \rangle$$

Per esempio:

$$\text{codifica } 1, 2, 5 \rightarrow \langle 1, 2, 5 \rangle = \langle 1, \langle 2, \langle 5, 0 \rangle \rangle \rangle =$$

$$\begin{aligned}\langle 1, \langle 2, 16 \rangle \rangle &= \langle 1, 188 \rangle = \\ &= 18144\end{aligned}$$

Per l'operazione di decodifica quello che facciamo è di considerare la codifica della mia lista M (non il risultato ma la lista di coppie incapsulate) come un albero binario. Allora dato questo albero il figlio sinistro è l'elemento sinistro (la testa della lista) ed il figlio destro è la parte restante della lista, allora continuando a considerare il figlio sinistro come una struttura dati lineare ottengo la lista codificata, questo attraversamento dell'albero in pre-ordine terminerà quando incontrerà un figlio destro 0 (l'ultimo elemento, c'è solo un elemento con 0).

Vogliamo effettuare delle implementazioni in pseudo-codice (simile C), assumiamo 0 come la lista nulla, e $\langle \rangle, sin, des$ come la funzione di Cantor $\langle \rangle$.

Implementazione codifica

```
int encode(x1, ..., xn){
    int k = 0;
    for (int i = n; i >= 1; i--)
        k = ⟨xi, k⟩;
    return k;
}
```

Implementazione decodifica

```
void decode(int n){
    if (n != 0){
        print(sin(n));
        decode(des(n));
    }
}
```

Altre operazioni utili possono essere quelle per calcolare la **lunghezza**:

```
int length(int n){
    return n == 0 ? 0 : 1 + length(des(n));
}
```

Ottiene per calcolare la **proiezione**:

$$proj(t, b) = \begin{cases} -1 & \text{se } t > length(n) \text{ o } t == 0 \\ x_t & \text{se } 1 \leq t \leq length(n) \text{ e } n = \langle x_1, \dots, x_t, \dots, x_m \rangle \end{cases}$$

```
int proj(int t, int n){
    if (t == 0 || t > length(n))
        return -1;
    else {
```

```

        if (t == 1)
            return sin(n);
        else
            return proj(t - 1, des(n));
    }
}

```

Esercizi - *incr, decr* (da implementare)

$$incr(t, n) = \begin{cases} -1 & \text{se } t > length(n) \text{ o } t == 0 \\ \langle x_1, \dots, x_t + 1, \dots, x_n \rangle & \text{se } 1 \leq t \leq length(n) \text{ e } n = \langle x_1, \dots, x_t, \dots, x_n \rangle \end{cases}$$

decr(*t, n*) = come sopra ma con $\langle x_1, \dots, x_{t-1}, \dots, x_m \rangle$

Corrispondente numerico : DATI $\sim \mathbb{N}$

Adesso sappiamo codificare liste di interi, questo ci dà un'idea del fatto che i dati siano isomorfi ad \mathbb{N} . Questo ci dà un modo per nascondere testi, suoni, immagini dietro ad un numero. Per esempio, per compattare un testo in un numero posso imporre una codifica (tipo ASCII), in questa maniera posso associare un numero per ogni carattere numerico, un insieme di caratteri numerici è codificabile utilizzando \langle, \rangle il quale mi darà un singolo numero.

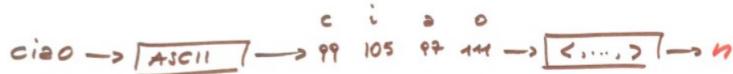


Figura 4.3: Codifica di un testo

Posso usare questo modo per comprimere i testi? In realtà questa tecnica non è un buon compressore, questo perché la crescita del numero è quadratica rispetto al numero di caratteri codificati (otteniamo un n troppo grande e si verifica un'espansione, altrettanto). *Perché non è un buon modo per crittografare i dati?* Il primo problema è che il testo crittografato sarebbe molto lungo (aumenta il rischio nella perdita di dati durante la trasmissione), secondo problema la copia di Cantor è una funzione biettiva, esiste un modo abbastanza diretto per tornare indietro.



Figura 4.4: Discretizzazione del suono in segnale digitale e poi codifica in numero naturale

Per i suoni il discorso è molto simile, grazie alla **campionatura** (o **discretizzazione**) dei segnali analogici in un segnale digitale, ovvero una grandezza discreta sotto forma di numeri naturali.

Per le immagini potrei usare la tecnica bitmap, quindi per ogni bit registro un colore che verrà codificato con un numero (e da qui come per gli altri casi sappiamo come muoverci).

Codifica di altre strutture dati

Codifica di un **array** a dimensione finita, proprio per questo non necessita di un elemento che mi ponga fine alla sequenza (so già dove fermarmi).

$$x_1, \dots, x_n \rightarrow [x_1, \dots, x_n]$$

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

Codifica di una **matrice**, esse sono array bidimensionali dove hanno anch'esse dimensione fissa e nota. Quindi mi basta codificare per riga le matrici e poi codificare le codifiche.

$$\begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} = [[x_{11}, x_{12}], [x_{21}, x_{22}]]$$

Sia per la codifica di array che per quello delle matrici sono facilmente implementabili le primitive numeriche ad esempio $\text{elem}(i, j, n)$ che restituisce l'elemento (i, j) -esimo di una matrice codificata dal numero n .

Codifica dei **grafi**, sappiamo che un grafo è rappresentabile con lista di adiacenza, matrice di adiacenza,... quindi possiamo codificare la lista o la matrice.

Conclusione

Abbiamo mostrato effettivamente che grazie alla corrispondenza effettiva di $\text{DATI} \sim \mathbb{N}$ che i dati possono essere "scartati" e tenuti solamente i numeri, questo perché ogni dato può essere rappresentato con un numero naturale, questo grazie a delle leggi matematica effettive ed implementabili, che mi permettono di codificare e decodificare il dato.

Quindi la nostra funzione $f : \text{DATI} \rightarrow \text{DATI}_\perp$ che rappresenta un qualsiasi problema può essere sostituita da una funzione $f : \mathbb{N} \rightarrow \mathbb{N}_\perp$, ed è questo quello che i nostri calcolatori calcoleranno, quindi possiamo dire che il nostro universo dei problemi $\text{DATI}_\perp^{\text{DATI}}$ non è altro che $\mathbb{N}_\perp^\mathbb{N}$

4.1.2 Prefazione alla dimostrazione $\text{PROG} \sim \mathbb{N}$

Per spiegare ciò utilizzeremo un linguaggio apposito detto "*linguaggio RAM*" e su un sistema formale apposito detto "*sistema RAM*". Grazie alla semplicità di questo potrò mostrare molto semplicemente:

- Che i programmi sono numerabili $\text{PROG} \sim \mathbb{N}$
- In maniera rigorosa la semantica dei programmi, $\mathcal{C}(P, _) \rightarrow \text{RAM}(P, _)$

- Ed altrettanto formalmente potrò definire la potenza computazionale $F(RAM)$, questo fornirà un'idea di *che cosa è calcolabile*.

Il linguaggio RAM è un *assembly* molto semplificato, per questo motivo l'idea di calcolabilità è altamente criticabile, poiché il modello potrebbe fare scappare qualcosa per via della banalità.

Si introduce un altro modello molto più sofisticato la macchina *while*(JVM), quello che è calcolabile è effettivamente calcolabile da questo.

Metteremo poi a confronto il modello formale $F(RAM)$ con quello sofisticato $F(while)$. Se queste due idee di calcolabilità risultassero diverse allora la cosa è un po' pericolosa, perché l'idea di calcolabilità dipenderebbe dal periodo storico. Se invece due idee così diametralmente opposte risultassero uguali, ovvero che calcolano lo stesso insieme di funzioni, allora incomincio a capire che l'idea di calcolabilità è intrinseca ai problemi. Ed è questo che ci porterà alla **tesi di Church** (spoiler).

Prima di arrivare a ciò dobbiamo dimostrare che $PROG \sim \mathbb{N}$, e per fare questo dobbiamo introdurre la macchina RAM.

4.1.3 Sistema di calcolo RAM

L'hardware della macchina RAM:

- Una **memoria** R costituita contigua di registri, ognuno di questi registri può memorizzare i numeri naturali arbitrariamente grandi (non hanno una capienza). Il registro R_1 è il registro di input; R_0 registro di output.
- Il **program counter** L , un registro contenente l'indirizzo dell'istruzione da eseguire.
- Un **programma** P , costituito da una sequenza di istruzioni, un'istruzione RAM può essere:
 - Incremento: $R_k \leftarrow R_k + 1$
 - Decremento: $R_k \leftarrow R_k - 1$ (non può mai andare sotto lo zero)

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

- Salto condizionato: IF $R_k = 0$ THEN GOTO $m, m \in 1, \dots, |P|$

Esecuzione su una macchina RAM

1. Fase di inizializzazione del programma P .
2. Si carica il dato m nel registro di output R_1 .
3. Si comincia ad eseguire l'istruzione al posto 1, e man mano il PC continua ad incrementare così da poi eseguire l'istruzione successiva (eccetto nel caso in cui non si incontri un'istruzione di salto).

4. Per convenzione la macchina si arresta quando il PC il numero 0

$$L = 0 \implies HALT$$

c'è possibilità per via dell'istruzione di salto di mandare in loop l'esecuzione della macchina.

5. L'output di questo programma dato un input x ha due possibilità

- Il programma si ferma ($HALT$) e l'output è presente all'interno del registro R_0 .
- Il programma non termina, allora in questo caso l'output per l'input sarà indefinito

Quello che fa il programma è calcolare una funzione:

$$\varphi(x) = cout(R_0) / \perp$$

Chiameremo **semantica del programma** P :

$$\varphi_p = \mathbb{N} \rightarrow \mathbb{N}_\perp$$

Questa è una definizione di semantica del programma molto intuitiva, quello che voglio fare però è utilizzare degli oggetti matematici con i quali specificare in maniera formale e corretta la semantica del programma.

Definizione formale di semantica di un programma RAM

Introduciamo il concetto di **semantica operazionale**, questo è il tipo di semantica più semplice. Significa specificare che cosa fa una data istruzione andando a vedere quale è l'effetto di quell'istruzione sulla data macchina.

Per descrivere l'effetto di un'istruzione, prendiamo una foto della macchina prima dell'esecuzione e dopo. La "foto" solitamente si chiama **stato** della macchina. Quindi spiego la semantica di una istruzione specificando il cambiamento di stato indotto dall'esecuzione di quell'istruzione. Questo significa dare la semantica operazionale di una macchina.

$$\textcolor{red}{STATO} \rightarrow \boxed{\textcolor{brown}{I\&Tr.}} \rightarrow \textcolor{blue}{STATO}$$

Esecuzione di un programma P e sua semantica L'esecuzione di un programma consiste nell'esecuzione di molteplici cambiamenti di stato a partire dallo stato iniziale S_{init} (partendo un input n) e giungendo a quello finale S_{final} . Si dice che la computazione di P induce una sequenza di stati.

$$S_{init} \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \dots \rightarrow S_{fin}$$

P induce una seq. di stati

Figura 4.5: Sequenza di stati indotta da P

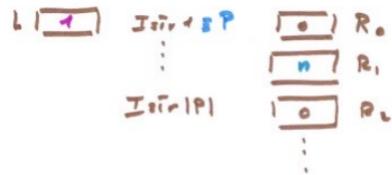


Figura 4.6: Situazione globale durante S_{init}

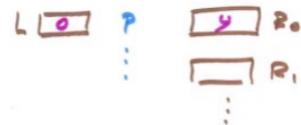


Figura 4.7: Situazione globale durante S_{final} (in caso di $HALT$)

Quindi la semantica del programma sarà:

$$\varphi_p(x) = \begin{cases} y \\ \perp \end{cases}$$

Dove \perp mi indica una sequenza infinita di stati in loop.

Ingredienti della definizione formale della semantica

- **Stato** di una macchina RAM, è lo stato dei registri di quella macchina. Matematicamente modella to come una funzione che mi restituisce il contenuto del risultato quando la macchina è nello stato S

$$S : \{L, R_i\} \implies \mathbb{N}$$

$$STATI = \mathbb{N}^{\{L, R_i\}} = \{\text{insieme di tutti i possibili stati della macchina}\}$$

$$S(R_j) = \text{contenuto di } R_j \text{ durante lo stato } S$$

- **Stato finale** della macchina RAM, uno stato tale per cui:

$$S(L) = 0 \implies \text{HALT}$$

- **Dato**, rappresentato da \mathbb{N} , questo perchè sappiamo che $\mathbb{N} \sim DATI$.
- **Inizializzazione**, dato il dato n prepara la macchina nell'ostato iniziale con input n . Quindi sarà una funzione in questa creerà uno stato S_{init} a partire da un input n :

$$in : DATI \rightarrow STATI \text{ t.c. } in(n) = S_{init}$$

Essa imposterà il contenuto del primo registro ad 1, e tutti gli altri a 0, e poi il caricherà nel program counter l'indirizzo della prima istruzione (contenuto del registro).

$$S_{init}(R_i) = \begin{cases} n & \text{se } i = 1 \\ 0 & \text{se } i = 0 \end{cases}$$

$$S_{init}(L) = 1$$

- **Programmi**, insieme dei programmi RAM $PROG = \{\text{programmi RAM}\}$, un singolo programma $P \in PROG$, tale per cui la sua cardinalità indica il numero di istruzioni contenute $|P| = \#istr$.
- **Esecuzione**, essa mi specifica la dinamica del programma che mi fa passare da uno stato al successivo. Questo è possibile con utilizzando la funzione *stato prossimo* δ . Tale funzione mi permette di spostarmi dallo stato attuale S a quello successivo S' .

$$\delta : STATI \times PROG \rightarrow STATI_\perp$$

$$\delta(S, P) = S'$$

Lo stato prossimo dipenderà dall'istruzione che in quel momento deve essere eseguita, e per conoscere tale istruzione devo andare a vedere il contenuto del program counter allo stato attuale $S(L)$ (quindi lo stato prossimo dipenderà da $S(L)$).

Come è definito lo stato prossimo in funzione dello stato attuale?

1. Se $S(L) = 0$ (ovvero in terminazione), allora lo stato prossimo non è definito $S' = \perp$.
2. Se $S(L) > |P|$, significa che abbiamo superato l'ultima istruzione e che il program counter è stato incrementato diventando così più grande del numero di istruzioni del programma. Quello che si fa è:

$$S'(L) = 0 \text{ HALT}$$

$$\forall i : S'(R_i) = S(R_i)$$

3. Se $1 \leq S(L) \leq |P|$, questo è il caso comune e considerando la $S(L)$ -esima istruzione:

- $R_k \leftarrow R_k \pm 1$, definita come:

$$\begin{aligned} S'(R_k) &= S(R_k) \pm 1 \\ S'(L) &= S(L) + 1 \\ S'(R_i) &= S(R_i) \text{ con } i \neq k \end{aligned}$$

- IF $R_k = 0$ THEN GOTO m , definita come:

$$\begin{aligned} S'(R_i) &= S(R_i) \\ \text{if } S(R_k) == 0 \text{ then} \\ &\quad S'(L) = m \\ \text{else} \\ &\quad S'(L) = S(L) + 1 \end{aligned}$$

Esecuzione del programma $P \in PROG$

Ora posso definire la sequenza di computazione di un programma $P \in PROG$ su un input $n \in \mathbb{N}$. Una computazione è una sequenza di stati indotta dalla dinamica:

$$in(n) = S_0, S_1, \dots, S_i, S_{i+1}, \dots$$

Eventualmente la sequenza può essere infinita (loop), o terminare se un S_m raggiunge un $S_m(L) = 0$ (halt).

Dobbiamo vincolare la sequenza di stati alla funzione $\delta(S, P)$, e dire che per ogni $\delta(S_i, P) = S_{i+1}$.

Quindi, la semantica di P , è definita come $\varphi_P : \mathbb{N} \rightarrow \mathbb{N}_\perp$:

$$\varphi_P(n) = \begin{cases} y & \text{se la computazione del programma termina in} \\ & \text{con } S(L)=0 \text{ ed il contenuto di } S_m(R_0) = y \\ \perp & \text{se la computazione del programma va in loop} \end{cases}$$

Ora posso definire formalmente la potenza computazionale del sistema RAM:

$$F(RAM) = \{f \in \mathbb{N}^{\mathbb{N}} : \exists P \in PROG, \varphi_P = f\} = \{\varphi_P : P \in PROG\} \not\subseteq \mathbb{N}_\perp^{\mathbb{N}}$$

Ovvero tutte le funzioni per cui esiste un programma P tale per cui $\varphi_P = f$ (o φ_P al variare di P).

Alcuni programmi RAM e le loro funzioni che vengono calcolate

$P \in$ IF $R_1 = 0$ THEN GOTO 6
 $R_0 \leftarrow R_0 + 1$
 $R_0 \leftarrow R_0 + 1$
 $R_1 \leftarrow R_1 + 1$
IF $R_2 = 0$ THEN GOTO 1
 $R_1 \leftarrow R_1 + 1$

$$\varphi_p(n) = 2n$$

Figura 4.8: n

$P \in$ $R_1 \leftarrow R_1 + 1$
 $R_1 \leftarrow R_1 + 1$
 $R_1 \leftarrow R_1 + 1$
IF $R_1 = 0$ THEN GOTO 1
IF $R_2 = 0$ THEN GOTO 9
 $R_0 \leftarrow R_0 + 1$
 $R_1 \leftarrow R_1 + 1$
IF $R_2 = 0$ THEN GOTO 5
 $R_0 \leftarrow R_0 + 1$

$$\varphi_p(n) = \begin{cases} 1 & n = 0 \\ n-2 & n > 0 \end{cases}$$

Figura 4.9: Funzione più complicata

Questi programmi possono essere spiegati formalmente con gli strumenti matematici precedentemente forniti, per esempio con il primo programma:

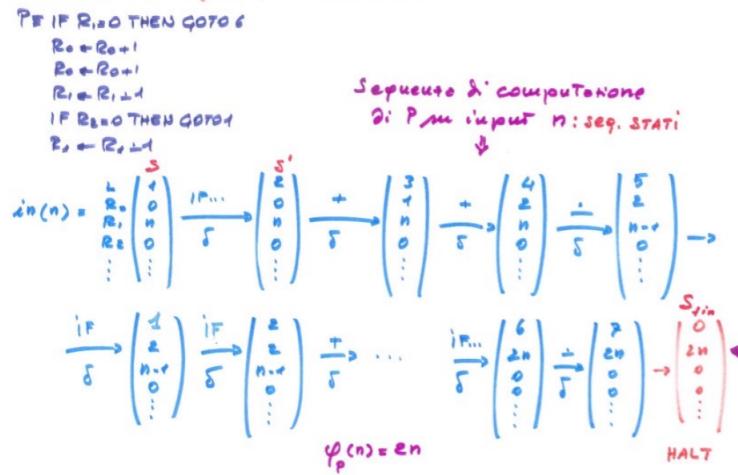


Figura 4.10: Rappresentazione formale di $\varphi_P(n) = 2n$

Questo è il modo corretto di procedere, e questo modo permette di definire formalmente la potenza computazionale di una macchina RAM, non lasciando nulla al caso.

Alcune considerazioni

- $F(\text{RAM})$ conterrà funzioni più complesse o solo quelle banali? In realtà si posso fare delle funzioni più complesse, questo comunque è un primo tentativo di fare qualcosa di ragionevole che mi permette la completa formalizzazione.
- Indubbiamente la semplicità di questo sistema di calcolo ci permette l'estrema formalizzazione, tale che vedremo che sarà possibile dimostrare $\text{PROG} \sim \mathbb{N}$

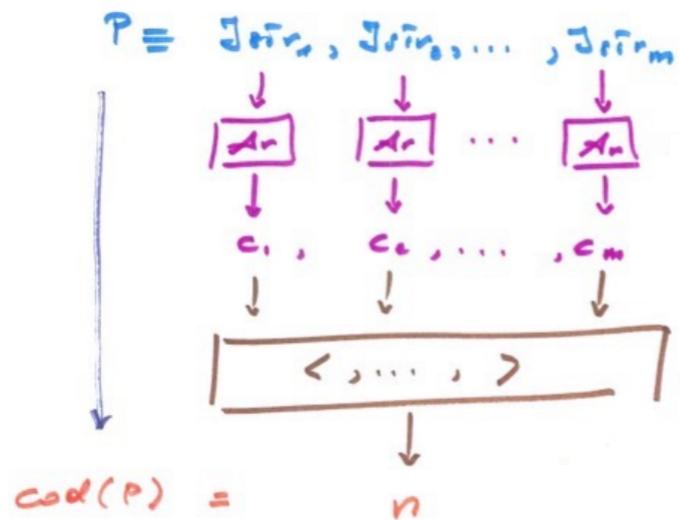
4.1.4 Dimostrazione $\text{PROG} \sim \mathbb{N}$

Dato un programma RAM vogliamo associare un numero a tale programma in maniera che partendo quel numero sia possibile tornare al sorgente.

Ovvero vogliamo dimostrare che anche i programmi possono essere in corrispondenza biunivoca con i numeri naturali (utilizzeremo la semplicità della macchina RAM).

$$P \equiv Istr_1, Istr_2, \dots, Istr_m$$

Il primo passo consiste nell'aritmetizzazione dell'istruzione, ovvero trasformare una lista di istruzioni in una lista di codici numerici. Successivamente utilizzando la funzione lista di Cantor possiamo trasformare questo elenco di codici in un singolo numero n .



Sappiamo che la funzione lista di Cantor è invertibile quindi possiamo ricostruire le codifiche associate alle istruzioni. Ora se l'operazione di aritmetizzazione del sorgente è invertibile, allora ci sarà possibile risalire al sorgente partendo da n .

In generale un procedimento che fa corrispondere ad una qualsiasi struttura matematica un numero, si chiama operazione di **aritmetizzazione** o **Godetizzazione**.

Il nostro scopo è quello di aritmetizzare ogni istruzione.

Aritmetizzare biunivocamente le istruzioni RAM

$$Ar : Istr \rightarrow \mathbb{N} \text{ e } Ar^{-1} : \mathbb{N} \rightarrow Istr \text{ t.c. } Ar(Istr) = n \Leftrightarrow Ar^{-1}(n) = Istr$$

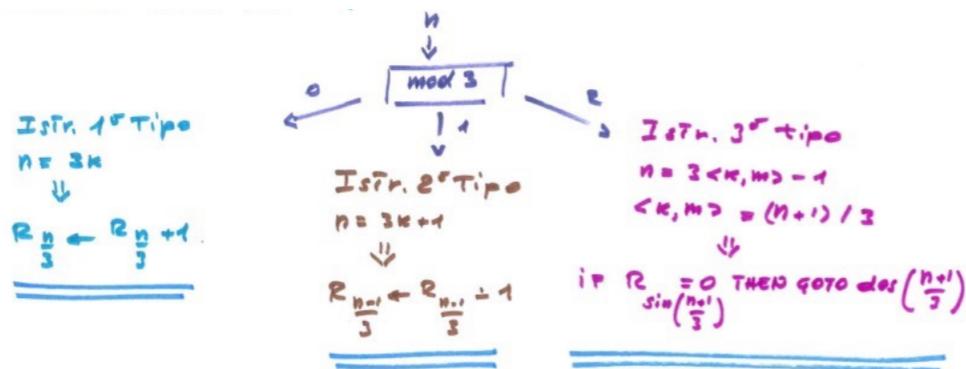
Nel caso di un programma RAM significa aritmetizzare tre tipi di istruzioni.

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

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

$$Ar(\text{If } R_k = 0 \text{ then goto } m) = 3\langle k, m \rangle - 1$$

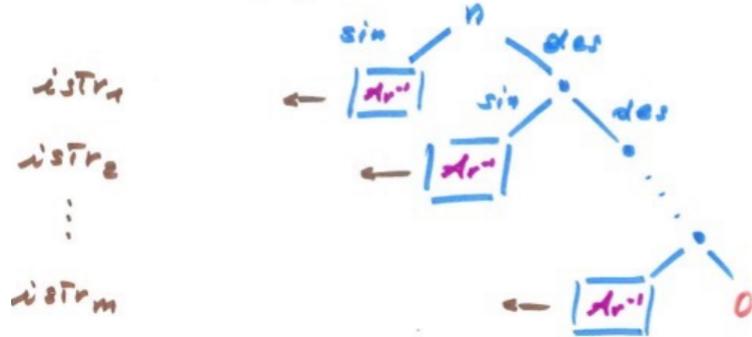
Come è fatta Ar^{-1} ? La funzione di decodifica è biunivoca? La funzione risulta sia iniettiva che suriettiva, l'applicazione di questa funzione è permessa attraverso l'operatore di modulo per 3.



Ricapitolando, il passaggio da programmi a numeri è molto semplice

$$cod(P) = \langle Ar(Istr_1), \dots, Ar(Istr_m) \rangle$$

mentre il passaggio da numeri a programmi, parto da n e trovo la parte sinistra e la parte destra. Vediamo passo per passo:



Al primo passo dalla parte sinistra trovo il codice della prima istruzione (aritmetizzato), quindi nel caso in cui io voglia il sorgente del programma quello che devo fare è applicare Ar^{-1} sulla parte sinistra. Questo finché non si incontra il terminatore sul figlio destro.

Se io volessi sempre scompattare il programma P e dato n io volessi sapere il numero di istruzioni del programma $|P|$, che cosa devo calcolare?

$$|P| = \text{length}(\text{cod}(P))$$

Abbiamo quindi dimostrato in maniera inequivocabile il secondo tassello, ovvero che i programmi sono equinumerosi rispetto ai numeri naturali.

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

Adesso che abbiamo acquisito gli strumenti, proviamo a vedere ad occhio qualche esempio.

Primo programma RAM

$P = \text{IF } R_0 = 0 \text{ THEN GOTO } 4$ $\varphi_P(n) = 0$
 $R_0 \leftarrow R_0 - 1$
 $\text{IF } R_1 = 0 \text{ THEN GOTO } 1$
 $R_0 \leftarrow R_0 - 1$

La corrispettiva aritmetizzazione delle istruzioni :

$$\Delta r(\text{IF } R_0 = 0 \text{ THEN GOTO 4}) = 3 \cdot \langle 0, 4 \rangle - 1 = 44$$

$$\Delta r(R_0 \leftarrow R_0 + 1) = 3 \cdot 0 + 1 = 1$$

$$\Delta r(\text{IF } R_1 = 0 \text{ THEN GOTO 1}) = 3 \cdot \langle 1, 1 \rangle - 1 = 14$$

$$\Delta r(R_0 \leftarrow R_0 + 1) = 3 \cdot 0 + 1 = 1$$

$$cod(P) = \langle 44, \langle 1, \langle 14, \langle 1, 0 \rangle \rangle \rangle \rangle = 50556496$$

Questo non è un ottimo modo per compattare i programmi ram, poiché cresce esponenzialmente rispetto al programma. Tuttavia questo ha scopi didattici.

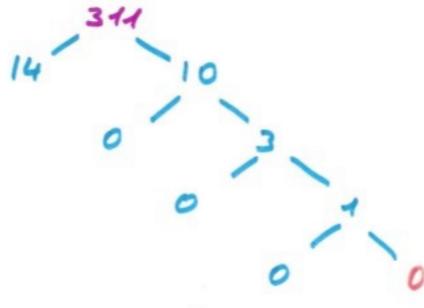
$$\varphi_{50556496}(n) = 0$$

Sorgente RAM da un numero

Il numero è il 311, ha come parte sinistra 14 e come parte destra 10. Facendo il modulo 3 del figlio sinistro otteniamo 2, quindi si tratta di un istruzione condizionale, in questo caso troviamo che il valore della coppia di Cantor $\langle k, m \rangle = 5$. Quindi dobbiamo trovare il figlio sinistro e destri di 5, che sono entrambi 1. Quindi la prima istruzione sarà:

If $R_1 = 0$ then goto 1

IF $R_1 = 0$ THEN GOTO 1
 $R_0 \leftarrow R_0 + 1$
 $R_0 \leftarrow R_0 + 1$
 $R_0 \leftarrow R_0 + 1$
Sorgente P



Quale è la semantica di questo programma? (ovvero $\varphi_P(n)$)

$$\varphi_P(n) = \varphi_{311}(n) = \begin{cases} \perp & \text{se } n = 0 \\ 3 & \text{se } n > 0 \end{cases}$$

Se l'input n è 0 vado in loop, poiché continuo a rieseguire la stessa istruzione, altrimenti parto dalla terza istruzione.

Riflessioni su $PROG \sim \mathbb{N}$

- Abbiamo dimostrato che i numeri sono un linguaggio di programmazione, questo perché un numero può rappresentare un linguaggio.
- $F(RAM) = \{\varphi_P : P \in PROG\}$, la potenza computazionale del sistema di programmazione RAM, adesso la posso scrivere come $F(RAM) = \{\varphi_i\}_{i \in \mathbb{N}}$ questo mostra in maniera inequivocabile che la potenza computazionale della macchina RAM è enumerabile.
- Per il sistema RAM si ha rigorosamente che $F(RAM) \sim \mathbb{N} \sim \mathbb{N}^{\mathbb{N}^{\mathbb{N}}}$,
- Questa è una prima idea di calcolabilità, esploriamo un sistema di calcolo \mathcal{C} più complesso e vediamo la sua potenza computazionale di questo, e vediamo se è più complessa. Questo va fatto è onestamente dire che ciò che è calcolabile sia $F(RAM)$ è troppo restrittivo.
- Possiamo avere che questo sistema di calcolo più avanzato ampli l'insieme $F(RAM)$.

4.1.5 Il sistema di calcolo While

Il linguaggio è basato su un linguaggio moderno, contrapposto all'assembly della macchina RAM (quindi anni 50'), adesso ci troviamo nell'utilizzo di un linguaggio strutturato.

Hardware:

- **Memoria** costituita da 21 registri. Siccome parliamo di linguaggio strutturato non si parla più con il termine registri ma ci si riferirà con *variabili*.

$$x_0, x_1, \dots, x_{20}$$

. La variabile x_1 ci si troverà l'input del programma, mentre l'output si troverà su x_0 .

- Program counter non presente, poiché si parla di linguaggio strutturato, ed in questo le istruzioni vengono eseguite una dopo l'altra (punti di inizio ciclo e finale sono ben definiti, ed il *goto* non è presente).

20 variabili potrebbero sembrare poche, in realtà la numerosità dei dati non è un problema poiché abbiamo primitive che mi permettono di condensare (coppia di Cantor) un infinità di variabili.

Il linguaggio while ha una sintassi induktiva (dove i costrutti dei linguaggi sono definiti su delle basi semplici, i mattoni, e man mano costruisco istruzioni più complesse):

- Comando di **assegnamento**: $x_k := 0$, $x_k := x_j + 1$, $x_k := x_j - 1$
- Comando **while**: while $x_k \neq 0$ do G , dove G ovvero il corpo del loop, può essere un comando di assegnamento, while o composto.

- Comando **composto**: begin $C_1; C_2; \dots; C_m$; end dove la sequenza è composta da comandi che possono essere di assegnamento, while o composto.

Come si può vedere sono strutture che sono in grado di nascondere altre strutture, di per sé un programma while è un comando composto.

Indicheremo con $W - PROG = \{\text{programmi WHILE}\}$ come l'insieme dei programmi WHILE, dove ciascuno è un programma costruito induttivamente.

Esempio di programma WHILE

```

w := begin
    x0 := x1 + 1;
    x1 := x2 - 1;
    while x1 ≠ 0 do
        begin
            x0 := x0 + 1;
            x1 := x1 - 1;
        end
    while x0 ≠ 0 do
        begin
            x0 := x0 + 1;
            x1 := x1 - 1;
        end
    end

```

Indicheremo con ψ_w le semantiche dei programmi WHILE (in questo caso il programma w).

$$\begin{aligned}\psi_w : \mathbb{N} &\rightarrow \mathbb{N}_\perp \\ \psi_w(x) &= 2x\end{aligned}$$

Le prime due istruzioni sono messe perché non è possibile effettuare una copia diretta, non stiamo utilizzando il linguaggio di programmazione RAM.

Il parsing del programma ne rivela la struttura induttiva, $W - PROG$ è un insieme definito induttivamente (ovvero partendo dalla base ed utilizzando dei passi induttivi): per dimostrare una proprietà P su $W - PROG$:

1. Dimostro che P vale sui comandi base (passo base).
2. Suppongo vero quella proprietà P sui comandi C base, e poi dimostro che vale sul comando più complesso while $x_n \neq 0$ do C (passo induttivo).

3. Suppongo vera P su C_1, \dots, C_m e la dimostro vera su $\begin{array}{l} \text{begin } \\ C_1; \dots; C_m \end{array}$

Quando abbiamo di fronte una struttura induttiva il modo migliore per dimostrare che una certa proprietà valga su tutti gli elementi della struttura dobbiamo farlo per induzione (in questo caso si parla di **induzione strutturale**).

Esempio

L'insieme degli alberi binari è un insieme definito induttivamente. Sappiamo che è un DAG.



Possiamo definire l'insieme degli alberi binari come l'insieme di tutti gli oggetti composti come mostrato in figura 4.1.5. Oppure, possiamo definirli induttivamente

- Base: Un nodo solo è considerabile un albero binario (con una singola foglia).
- Induzione: Se T_1 e T_2 sono alberi binari anche



Figura 4.11: Binary Tree (indicati i nodi interni con le frecce marroni)

- Nient'altro è un albero binario.

Allora vogliamo dimostrare per induzione la proprietà P , tale che:

$P \equiv$ "Su ogni albero binario, il numero dei nodi interni è minore di uno rispetto a quello delle foglie."

Allora dimostriamola per induzione:

- Base: è vero che se ho un solo nodo la proprietà P è vera? Si poiché nodi interni non sono presenti.

- Induzione: supponiamo vera P su T_1, T_2
 - T_1 : foglie f_1 foglie e $f_1 - 1$ nodi interni.
 - T_2 : foglie f_2 foglie e $f_2 - 1$ nodi interni.

Per esempio, prendiamo questo albero:



notiamo che il numero di foglie è pari a $f_1 + f_2$. Adesso la domanda fondamentale è la seguente *quanti nodi interni ha questo albero?* Il numero di nodi interni è pari alla somma tra i nodi interni di T_1 con quelli di T_2 . Ma questo riutilizzando i mattoncini basilari non è altro che

$$(f_1 - 1) + (f_2 - 1) + 1 = f_1 + f_2 - 1$$

P risulta dimostrata per induzione su tutta la struttura.

Esempio 2

Voglio definire una funzione $depth : \tau \rightarrow \mathbb{N}$ su un albero binario che dato un qualsiasi albero binario mi restituisce la **profondità**: ovvero, la lunghezza del cammino più lungo dalla radice ad una foglia dell'albero.

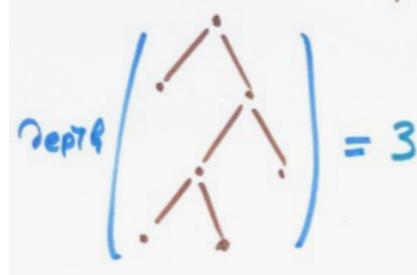
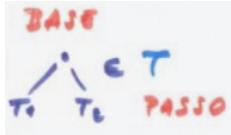


Figura 4.12: Definizione grafica

Questa è una possibile definizione, oppure potremmo definirla induttivamente:

1. Base: Dato un BT costituito da una sola radice darà $depth = 0$
2. Induzione: Dato questo albero binario



si prende la profondità maggiore due due sotto alberi più uno, ovvero il nuovo livello $depth = 1 + \max\{depth(T_1), depth(T_2)\}$

3. Nient'altro è in τ

Esecuzione su una macchina WHILE

Considerando un input n

- La prima è una fase di inizializzazione, quindi si avvia la macchina con il nostro programma w , dove tutti i registri della variabili sono inizializzati a 0, tranne per il registro in input x_1 che contiene n .
- Si comincia ad eseguire il programma w , sequenzialmente sulle istruzioni, in questo caso non c'è assolutamente bisogno di program counter (motivi già spiegati).
- Possiamo avere due casi, o l'esecuzione effettivamente termina o si incappa in un loop (while true). Nel primo caso l'output lo andiamo a leggere nella variabile x_0 (se HALT), altrimenti diciamo che è indefinito \perp . Definendo così la semantica del programma w , $\psi_w(n) = cont(x_0) / \perp$.

Vogliamo essere precisi anche in questo caso, ed espandere il discorso della sequenza di istruzioni durante l'esecuzione di un programma WHILE (introducendo anche qui il concetto di stato).

4.1.6 Definizione formale di semantica di un programma WHILE

- **Stato**, una foto dove compare in maniera completa tutto ciò che accade sulla macchina in quel dato istante. Rispetto alla macchina RAM lo stato viene definito in maniera differente, dal punto di vista matematico uno stato è una tupla di 21 elementi (c_0, \dots, c_{20}) . Detto ciò, l'insieme di tutti i possibili stati $w - stati$ è composto da \mathbb{N}^{21} ovvero tutte le possibili 21-tuple (tuple di lunghezza 21) infinite.
- **Dati**, sono l'insieme dei numeri naturali.
- **Inizializzazione**, sta volta la funzione può essere modellata su 21 elementi, la funzione prende un numero e restituisce uno stato

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

- **Semantica operazionale**

$$[] : w-com \times w-stati \rightarrow w-stati_{\perp}$$

Dato un comando while C e uno stato \underline{x} , allora

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

sarà la funzione stato prossimo dove \underline{y} è lo stato prossimo di \underline{x} a seguito dell'esecuzione del comando C . Ovviamente possiamo definire induttivamente $[C](\underline{x})$ sulla struttura induttiva del comando C .

4.1.7 Definizione induttiva della semantica while

- Base: gli **assegnamenti**

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

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

- Passo:

- Comando **composto**

$$[\underline{\text{begin}} \ C_1; \dots; C_m; \underline{\text{end}}](\underline{x})$$

induttivamente conosco la semantica di ciascuno di questi programmi, come essi agiscono. Allora posso definire la semantica di questo comando come:

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

Quindi l'applicazione del primo comando (che provoca cambiamento di stato) sullo stato iniziale, seguita iterativamente dall'applicazione degli m comandi sugli stati risultanti.

- Comando **while**

$$[\text{while } x_k \neq 0 \text{ do } C](\underline{x})$$

, anche qui conosco per ipotesi induttiva la semantica del comando C .

$$[C](\dots ([C]([C](\underline{x}))) \dots) = \underline{y} = [C_1] \text{ e } \dots \text{ e } [C_m](\underline{x})$$

Il numero di volte che applicherò il comando C quante volte serve per azzerare il contenuto di x_k

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

Ovvero il più piccolo numero di volte che mi serve per azzerare la componente k -esima, altrimenti vado in un loop interminabile.

- Semantica di W e $W - PROG$: a questo punto so definire in maniera formale la semantica di un programma while, prendiamo il programma while w (il quale è un comando composto).

Quindi è la semantica del comando composto che rappresenta w , quindi non è altro che preparare la macchina su input x .

$$\Psi_w(x) = Pro(0, [w](w - in(x)))$$

Ovvero la proiezione 0-esima dell'esecuzione del comando composto di w dato uno stato finale x (avviato da uno stato iniziale), significa che se il programma termina devo pescare il contenuto di x_0 (la proiezione di uno stato finale di un programma w). Nel caso in cui non avvenisse l'HALT allora avrò una semantica indefinita \perp .

Esempio di semantica di un piccolo programma while

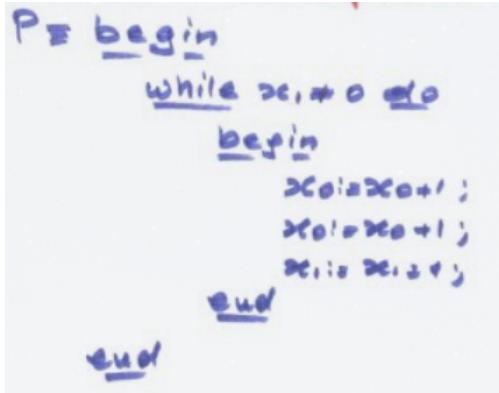


Figura 4.13: Programma che calcola $2n$

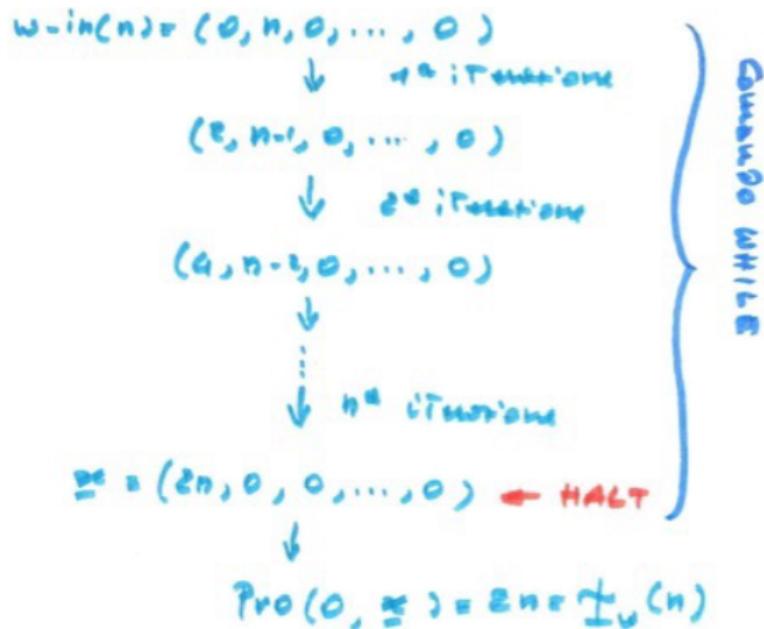


Figura 4.14: Deduzione della semantica Ψ_w

Potenza computazionale del sistema while

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

Definiamo la potenza computazionale di una macchina while come l'insieme di tutte le funzioni per cui esiste un programma while per cui si può calcolare quella funzione. Naturalmente la domanda è: *che relazione esiste tra $F(\text{RAM})$ e $-F(\text{WHILE})$?*, quindi che cosa è calcolabile di questi due sistemi? $\{\varphi_P : P \in \text{PROG}\}$? Magari la macchina WHILE mi fornisce più potenza della macchina RAM? L'unico modo di scoprirlo è confrontando le due idee. Quali situazioni si possono verificare:

1. Una situazione dove $F(\text{RAM}) \not\subseteq F(\text{WHILE})$ (inclusione propria) ovvero, che ci sono funzioni calcolabili con i programmi WHILE che non sono calcolabili sulle macchine RAM.
2. Le due idee di calcolabilità non sono confrontabili, ovvero un intersezione tra i due insiemi dove magari alcune funzioni sono calcolabili da entrambe le architetture, oppure sono due insiemi totalmente disgiunti. In entrambi i casi siamo preoccupati, perché l'idea di calcolabilità è dipendente dal modello di calcolo adottato.

3. Una situazione dove $F(WHILE) \not\subseteq F(RAM)$ questo sarebbe sorprendente, la macchina sofisticata non riesce a calcolare delle funzioni più semplici.
4. Una situazione dove $F(WHILE) = F(RAM)$, ovvero due cose filosoficamente diverse eppure entrambe riescono a calcolare la stessa classe di funzioni, questo sarebbe molto interessante, perché allora vuol dire che calcolabile non dipende dalla tecnologia-

Adesso dobbiamo cercare di capire quali tra i seguenti punti è quello veritiero.

Confrontiamo $F(RAM)$ e $F(WHILE)$

Iniziamo con il confrontare $F(RAM)$ e $F(WHILE)$ introducendo due sistemi di calcolo \mathcal{C}_1 e \mathcal{C}_2 con i loro relativi linguaggi di programmazione. Attraverso i quali riusciamo a scrivere l'insieme dei programmi scritti con i relativi linguaggi $\mathcal{C}_1 - PROG$ e $\mathcal{C}_2 - PROG$. Le relative potenze computazionali

$$F(\mathcal{C}_1) = \{f \in \mathbb{N}_{\perp}^{\mathbb{N}} : f = \Psi_{P_1} \text{ per qualche } P_1 \in \mathcal{C}_1 - PROG\} = \{\Psi_{P_1} : P_1 \in \mathcal{C}_1 - PROG\}$$

$$F(\mathcal{C}_2) = \{f \in \mathbb{N}_{\perp}^{\mathbb{N}} : f = \Psi_{P_2} \text{ per qualche } P_2 \in \mathcal{C}_2 - PROG\} = \{\Psi_{P_2} : P_2 \in \mathcal{C}_2 - PROG\}$$

Cominciamo a cercare degli strumenti per confrontare le potenze computazionali. Come faccio a dimostrare $F(\mathcal{C}_2) \subseteq F(\mathcal{C}_1)$? Per dimostrare che tale inclusione sia vera, mi basta dimostrare che un elemento di un insieme appartenga all'altro, in linguaggio matematico:

$$\forall f \in F(\mathcal{C}_1) \implies f \in F(\mathcal{C}_2)$$

Quando stiamo compilando un programma quello che stiamo dimostrando è che il linguaggio utilizzato come sorgente è potente uguale al linguaggio macchina/oggetto.

$$\exists P_1 \in \mathcal{C}_1 - PROG : f = \Psi_{P_1} \implies \exists P_2 \in \mathcal{C}_2 - PROG : f = \varphi_{P_2}$$

Ovvero vuol dire che esiste un programma P_1 tale per cui ne esiste uno equivalente nel secondo sistema (allora ho dimostrato che tutto ciò che è fattibile in \mathcal{C}_1 è fattibile in \mathcal{C}_2 , questo attraverso una *traduzione*).

4.1.8 Concetto di traduzione

Dati i sistemi $\mathcal{C}_1, \mathcal{C}_2$, una **traduzione** (controparte di *compilazione*) dal primo verso il secondo è una funzione:

$$T : \mathcal{C}_1 - PROG \rightarrow \mathcal{C}_2 - PROG$$

con le seguenti proprietà (che sono quelle che vogliamo nei nostri compilatori):

- deve essere **programmabile** effettivamente.

- deve essere **completa**, ovvero che traduce *ogni* programma scritto in \mathcal{C}_1 in uno scritto in \mathcal{C}_2 .
- deve essere **corretta**, ovvero che mantiene la semantica del programma

$$\forall P \in \mathcal{C}_1 - PROG : \varphi_{T(P)} = \Psi_P$$

questa è la formalizzazione del concetto di compilatore. Allora il **teorema** che posso scrivere è il seguente, se esiste $T : \mathcal{C}_1 - PROG \rightarrow \mathcal{C}_2 - PROG$, allora $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$ se esiste una traduzione da \mathcal{C}_1 a \mathcal{C}_2 , allora ho dimostrato che la potenza computazionale del primo linguaggio è contenuto (o uguale) a quella del secondo.

Dimostrazione

$$f \in F(\mathcal{C}_1) \implies \exists P \in \mathcal{C}_1 - PROG : f = \Psi_P$$

Applico T ed ottengo

$$T(P) \in \mathcal{C}_2 - PROG \text{ (completa) con } \varphi_{T(P)} = \Psi_P = f$$

allora esiste dunque un programma $T(P) \in \mathcal{C}_2 - PROG$ per f per cui $f \in F(\mathcal{C}_2)$.

Sostanzialmente, ipotizzo l'esistenza di una funzione che rientri nella potenza computazionale del primo linguaggio \mathcal{C}_1 , esiste un programma p scritto in questo linguaggio che è in grado di calcolare f . Siccome esiste la traduzione, prendo p e lo compilo, e questo funziona perché la traduzione è completa.

4.1.9 Dimostrazione $F(WHILE) \subseteq F(RAM)$

Da un punto di vista immediato sembrerebbe la cosa più difficile da dimostrare, ma se ci si pensa è proprio quello che accade durante la compilazione dei programmi (per esempio C, C++, ...). Quindi vogliamo a dimostrare che la macchina RAM non ha nulla da invidiare rispetto alla macchina WHILE.

Dimostrare questa inclusione mi spinge ad esibire che esiste una traduzione da un programma WHILE ad uno RAM

$$Comp : W - PROG \rightarrow PROG$$

Chiamerò questa traduzione *compilazione* (la quale è una funzione di traduzione che rispetta le tre proprietà). Per pura comodità, scrivere un compilatore da un programma WHILE ad uno RAM vuol dire prendere un programma del primo tipo e scrivere un frammento del secondo, per semplicità ammetterò di utilizzare un programma RAM che utilizza delle *label* (cosa che mi permette di fare salti nel codice).

| | |
|--|---|
| RAM PURO IF $R_1 = 0$ THEN GOTO <u>8</u> $R_0 \leftarrow R_0 + 1$ $R_0 \leftarrow R_0 + 1$ IF $R_0 = 0$ THEN GOTO <u>1</u> $R_1 \leftarrow R_1 + 1$ <i>Judicato</i> | RAM ETICHETATO <u>LOOP:</u> IF $R_1 = 0$ THEN GOTO <u>EXIT</u> $R_0 \leftarrow R_0 + 1$ $R_0 \leftarrow R_0 + 1$ IF $R_0 = 0$ THEN GOTO <u>LOOP</u> <u>EXIT:</u> $R_0 \leftarrow R_0 - 1$ <i>Etichetta</i> |
|--|---|

semplicemente al posto di utilizzare i riferimenti numerici alle righe utilizzeremo delle parole, con un semplice post-processing posso trasformare il "RAM etichettato" in "RAM puro" (l'aggiunta di etichette non aumenta la potenza del linguaggio RAM).

Forma del compilatore

Teniamo conto della natura degli oggetti su cui agirà il compilatore $W - PROG$ è un insieme definito induttivamente. Questo significa che anche $Comp$ può essere definito induttivamente.

1. mostriamo come compilare le basi di quel linguaggio, quindi gli assegnamenti (base).
2. posso usare l'ipotesi induttiva, quindi assumo di saper compilare il comando $Comp(C_1), \dots, Comp(C_m)$ e ti faccio vedere come compilare il comando composto

begin $C_1; \dots; C_m;$ end

3. posso usare l'ipotesi induttiva, quindi di saper compilare $Comp(C)$ e mostrarlo come compilare il comando *while* che ha come corpo C , while $x_n \neq 0$ do C

In tale maniera si saprà come compilare tutto $W - PROG$. Sia nota la seguente proprietà di compilazione, ogni volta che durante una traduzione incontreremo nel programma una variabile x_k ad essa verrà associata il registro R_k (abbiamo tranquillamente lo spazio per mappare tutte le variabili, visto che il numero di registri è infinito).

Base: assegnamenti

Vogliamo considerare i seguenti frammenti RAM per eseguire gli assegnamenti base dei programmi WHILE:

- $Comp(x_k := 0) =$

$\text{Comp}(x_n := 0) =$
 LP: IF $R_n = 0$ THEN GOTO EX
 $R_n \leftarrow R_n + 1$
 IF $R_{21} = 0$ THEN GOTO LP
 EX: $R_n \leftarrow R_n - 1$

controllo se un registro è azzerato, nel caso non fosse azzerato allora lo decremento e ripeto il controllo (R_{21} non sarà mai azzerato). Ad un certo punto uscirò dal ciclo, l'ultima operazione è praticamente nulla perché non danneggia l'operato precedente (è già zero). Tutti i registri da R_{21} in poi sono tutti azzerati, perché non vengono utilizzati dal programma WHILE, quindi possiamo utilizzarli come meglio crediamo.

- $\text{Comp}(x_k := x_j \pm 1) =$

notiamo che c'è un caso dove la compilazione è immediata, in un caso di quest'istruzione troviamo subito l'istruzione RAM che fa la stessa cosa. Il caso particolare accade quando $k = j$. Se $k == j \rightarrow R_k \leftarrow R_k \pm 1$, il problema allora è nell'altro caso, ovvero quando $k \neq j$.

$\text{Comp}(x_n := x_j + 1) =$
 $(k \neq j)$
 LP: IF $R_j = 0$ THEN GOTO EX1
 $R_j \leftarrow R_j + 1$
 $R_{22} \leftarrow R_{22} + 1$
 IF $R_{21} = 0$ THEN GOTO LP
 EX1: IF $R_n = 0$ THEN GOTO EX2
 $R_n \leftarrow R_n + 1$
 IF $R_{21} = 0$ THEN GOTO EX1
 EX2: IF $R_{22} = 0$ THEN GOTO EX3
 $R_{22} \leftarrow R_{22} + 1$
 $R_j \leftarrow R_j + 1$
 $R_n \leftarrow R_n + 1$
 IF $R_{21} = 0$ THEN GOTO EX2
 EX3: $R_n \leftarrow R_n + 1$

1. Salvo x_j in R_{22}
2. Azzero x_k , ovvero R_k

3. Rigenero x_j e x_k da R_{22}
4. Sommo/sottraggo 1 da x_k

Passo induttivo Consideriamo il comando composto $\begin{array}{l} C_1; \dots; C_m; \end{array}$, considerando che esiste l'ipotesi induttiva, quindi assumo noti $Comp(C_1), \dots, Comp(C_m)$ mostro che la compilazione $Comp(\begin{array}{l} \text{begin } C_1; \dots; C_m \text{ end} \end{array})$ sarà:

```

 $Comp(C_1);$ 
 $Comp(C_2);$ 
...
 $Comp(C_m);$ 

```

Consideriamo il comando while $x_k \neq 0$ do C , allora vogliamo mostrare che la compilazione $Comp(\text{while } x_k \neq 0 \text{ do } C)$ corrisponde al frammento RAM:

```

LP: IF R_k = 0 THEN GOTO EX
    comp(C)
    IF R_k = 0 THEN GOTO LP
EX: R_k ← R_k + 1

```

Ricapitolando

La funzione $Comp : W - PROG \rightarrow PROG$ che abbiamo definito induttivamente soddisfa:

1. è facilmente **programmabile**.
2. compila ogni sorgente WHILE (**completo**).
3. mantiene la semantica $\Psi_w = \varphi_{Comp(w)} \implies \text{corretta}$

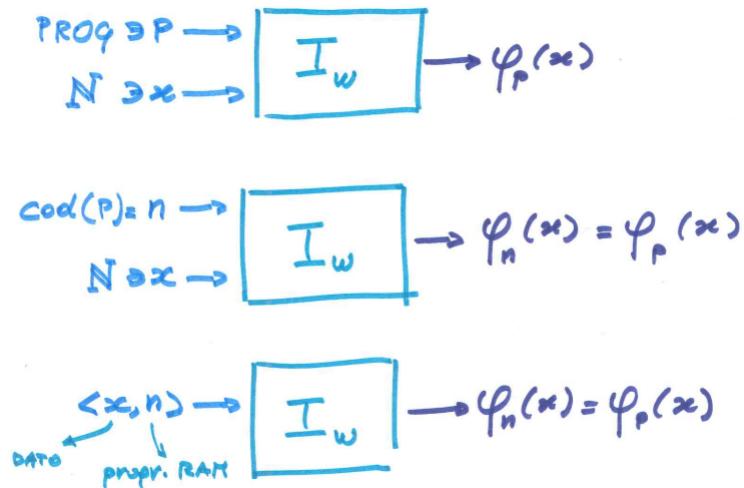
I punti 2 e 3 si dimostrano facilmente per induzione strutturale, quindi possiamo dimostrare che $Comp$ è una traduzione da WHILE a RAM, e che quindi $F(WHILE) \subseteq F(RAM)$

4.1.10 Dimostrazione $F(RAM) \subseteq F(WHILE)$

Il problema in questo caso è dato dal fatto che di deve gestire il "go to", che non è presente nel linguaggio WHILE (ed è proprio per questo motivo che utilizzare goto nei linguaggi strutturati è solo un problema). Dimostreremo come è possibile evitare l'utilizzo delle istruzioni "go to" nella programmazione strutturata.

Introduciamo il concetto di **interprete**, per quasi tutti i linguaggi di programmazione ormai c'è la possibilità di avere la versione compilata che interpretata (con differenze prestazionali). La compilazione produce un programma equivalente che sta in piedi da solo che può essere eseguito direttamente dalla macchina, l'interpretazione simula l'esecuzione di un programma attraverso un interprete.

Vediamo come scrivere un interprete di programmi RAM scritto in WHILE, che chiameremo I_W .



Esso prenderà in ingresso un programma scritto in RAM ed un input, e produrrà in uscita non un codice oggetto ma la semantica del programma P sul dato x . Notiamo che in ingresso viene dato un programma P che è listato, ma sappiamo che i nostri programmi WHILE non lavorano sulle stringhe. Un programma WHILE ha una sola variabile di input x_1 che può solo contenere numeri. *Come facciamo a fornire il listato di P al programma WHILE?* Il mio I_W prenderà in ingresso la codifica del programma p che sarà fornito sotto forma di numero.

Ma il nostro programma può veramente prendere due dati in questa maniera? Notiamo bene che nel linguaggio WHILE x_1 non può prendere una coppia di numeri distinti, ma allora dobbiamo passare un ulteriore coppia di Cantor di questi due numeri.

Il risultato sarà il funzionamento del programma con codice n che in realtà è P .

Quindi la semantica di I_W :

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

Macro-WHILE

Siccome devo scrivere un interprete in WHILE, voglio facilitarmi il lavoro uti-

lizzando una versione modificata del linguaggio (sempre permettendo di tornare al WHILE puro e senza avere guadagni o perdite prestazionali). Questo per permettere di scrivere un codice più semplice.

Per esempio:

```

 $x_K := x_g + x_s \rightarrow x_K = \langle x_g, x_s \rangle \rightarrow x_K = \langle a_1, a_2, \dots, a_n \rangle$ 
 $x_K := Pro(x_g, x_s) \quad // mette in x_K l'elemento x_g-esimo della$ 
 $\text{lista codificata in } x_s$ 
 $x_K := iner(x_g, x_s) \quad // mette in x_K la codifica della lista codificata in x_g$ 
 $\text{ove la } x_g\text{-esima comp. è incrementato di 1}$ 
 $x_K := decr(x_g, x_s) \quad // come sopra ma la } x_g\text{-esima comp. è decrementata di 1}$ 
 $x_K := sin(x_g) \rightarrow x_K = \text{dec}(x_g) \rightarrow \text{if ... Then ... else}$ 

```

Come vediamo queste non sono istruzioni di WHILE puro, dovremmo scrivere molte più istruzioni per ottenere ciò. Si ha anche la possibilità di rappresentare la coppia di cantor di due numeri, visto che si ottengono con somme/-sottrazioni, ed anche la possibilità di rappresentare la lista di Cantor.

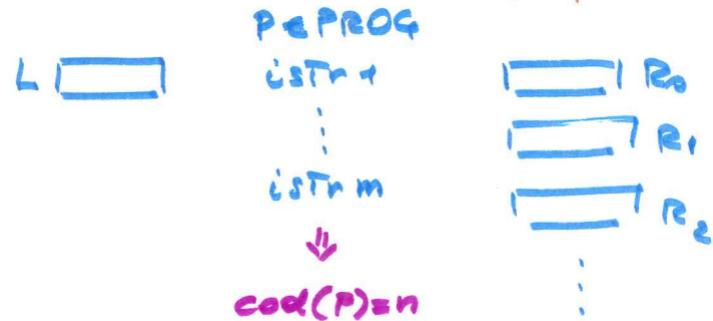
Ciascuna di queste MACRO può essere sostituita da un frammento di WHILE puro (seppur risultati più comodo da utilizzare)

$$F(\text{MACRO} - \text{WHILE}) = F(\text{WHILE})$$

Interprete WHILE di programmi RAM

Lo scopo di questo interprete è quello di ricreare esattamente al suo interno (scritto in WHILE) utilizzando le variabili la situazione della macchina RAM, entro cui fare eseguire il programma.

Sappiamo che la macchina RAM riceve un programma P che è un listato delle istruzioni di un programma RAM. Sappiamo che la memoria della macchina RAM ha un numero infinito di registri (a destra) ed un registro L che è il program counter.



Questo interprete scritto in WHILE si trova davanti ad un problema iniziale, si ha un numero di registri *infinito* sulla macchina RAM (mi servono quindi un numero infinito di variabili, ma purtroppo sono solo 21). Però sappiamo che il nostro programma P ha come codifica il numero n , allora sicuramente non userà mai un registro il cui registro sarà mai superiore di n , sicuramente $R_j < n$. Significa che posso limitarmi a maneggiare un numero di registri che va da R_0, \dots, R_{n+2} , quindi potrò sviluppare questa dinamica in una lista.

Utilizzo delle variabili da parte dell'interprete:

- x_0 per indicare la porzione di **memoria della macchina** RAM che identifica i registri del programma, quindi lo stato della memoria istante per istante del programma che devo simulare.
- x_1 giocherà il ruolo del **program counter** L .
- x_2 metterò il dato su cui deve girare il programma (**input**).
- x_3 ci metterò il **listato del programma** ovvero n (ovvero $\text{cod}(P)$).
- x_4 di volta in volta verrà caricata l'istruzione che il program counter deve eseguire (gioca il ruolo del registro di istruzione corrente *instruction register*, IR).

è opportuno ricordare che inizialmente I_W troverà il suo input nella variabile x_1 , ovvero $\langle x, n \rangle$, nella variabile input x_1 :

$$x_0 := \langle x, n \rangle \rightarrow \boxed{I_W} \rightarrow \varphi_n(x)$$

Composizione dell'interprete:

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

```

 input(<x,n>)           // in x,
x2:=sin(x1);             // dato x
x3:=des(x1);            // listato di P (i.e. n)

```

Rispettivamente i primi due registri x_2 ed x_3 vengono definiti facilmente, poiché l'input del programma ed il listato si trovano sono rispettivamente il figlio sinistro ed il figlio destro.

```

x0:=<0,x1,0,...,0>;   // mem. iniziale di punteggio
x1:=1;                  // Program Counter sullo 1° istru.

```

La variabile x_0 deve contenere la memoria del programma P , sappiamo che sarà costituita dalla lista di tutti i registri che vanno da R_0 a R_{n+2} , dove in R_1 sarà presente l'input del programma P (che andremo a prelevare da x_2), tutti i restanti registri saranno azzerati (la lista è ovviamente rappresentata da una coppia di Cantor). Il program counter viene impostato all'inizio del programma ad 1, in maniera che punti alla prima istruzione del programma.

```

while ( $x_1 \neq 0$ ) do           //  $x_1 == 0 \Rightarrow HALT$ 
    if ( $x_1 > \text{length}(x_3)$ ) Then   // Se ho Tanti istn.
         $x_4 := 0$ ;                   // HALT
    else
         $x_6 := \text{Pro}(x_1, x_3)$ ;    // fetch
        if ( $x_4 \bmod 3 == 0$ ) Then   //  $R_n \leftarrow R_n + 1$ 
             $x_5 := x_4 \div 3$ ;          // K
             $x_6 := \text{incr}(x_5, x_6)$ ;
             $x_1 := x_4 + 1$ ;
        if ( $x_4 \bmod 3 == 1$ ) Then   //  $R_n \leftarrow R_n + 1$ 
             $x_5 := (x_4 - 1) \div 3$ ;  // K
             $x_6 := \text{dec}(x_5, x_6)$ ;
             $x_1 := x_4 + 1$ ;
        if ( $x_4 \bmod 3 == 2$ ) Then   //  $IFR_n = 0 \dots m$ 
             $x_5 := \sin((x_4 + 1) \div 3)$ ; // K
             $x_6 := \cos((x_4 + 1) \div 3)$ ; // m
            if ( $\text{Pro}(x_5, x_6) == 0$ ) Then
                 $x_1 := x_6$ ;
            else
                 $x_4 := x_4 + 1$ ;
         $x_0 := \sin(x_0)$ ;           //  $q_n(w)$ 
    
```

Figura 4.15: Fetch - Decode - Execute

- L'esecuzione di un programma RAM si ferma quando il program counter $L = 0$ (HALT), l'interprete non è altro che un ciclo che esegue sempre le operazioni di fetch,decode e execute del processore.
- Sappiamo che l'interpretazione di un programma termina nel caso in cui abbiamo terminato la simulazione ci dobbiamo fermare, sappiamo che questo accade quando il program counter supera il numero di istruzioni. Il numero di istruzioni è condensato nella variabile x_3 , al superamento della dimensione di quest'ultima da parte di x_1 , impostiamo l'HALT (altrimenti continuiamo con l'esecuzione del programma).
- Usiamo la macro $\text{Pro}()$ per prendere la codifica dell'istruzione del programma caricato in x_3 che viene puntato da x_1 e la carichiamo in x_4 (fetch).

- Si deve capire che tipo di istruzione stiamo trattando, questo lo possiamo fare considerando il resto della divisione per 3:
 - Resto = 0, si tratta di un incremento di un registro R_k , dove $k = x_4/3$. Questo significa andare nella memoria simulata in x_0 , posizionarsi nel registro $x_4/3$ ed incrementarlo di 1. Per sapere la posizione di k utilizzo una delle variabili disponibili, x_5 , utilizzo la macro *Incr()* per incrementare l'elemento in posizione x_5 della lista x_0 (memoria del programma). Al termine avanzo il program counter perché ho eseguito l'istruzione.
 - Resto = 1, si tratta di un decremento di un registro R_k , in questo caso il valore di $k = x_4 - 1/3$, la macro che utilizzo è *Decr()* (per il resto come prima).
 - Resto = 2, si tratta di un condizionale, allora devo trovare sia il R_k (la condizione) che m (l'istruzione da eseguire). Una volta individuate si va a cercare l'istruzione m nella memoria della macchina, se presente si sposta il program counter a quell'istruzione, altrimenti si continua linearmente con l'istruzione successiva.
- Si carica nella variabile di output x_0 l'output del programma simulato che è il primo registro del programma (quindi il figlio sinistro della coppia di Cantor).

Conseguenze di I_W Una volta che ho l'interprete io posso creare un compilatore da programmi WHILE a programmi RAM.

$$\Psi_{I_W}(\langle x, n \rangle) = \varphi_W(x) = \varphi_P(x)$$

$$Comp : \text{Prog} \rightarrow W - \text{PROG}$$

Comp($P \in \text{PROG}$)
 $n \leftarrow \text{cod}(P);$
 Comp(P)
 $x_1 := \langle x_0, n \rangle;$
 $I_W;$

Prima di tutto calcolo la codifica di P , poi la posso cablare nella variabile x_1 ed infine chiamo l'interprete (il listato precedente descritto).

- **programmabile:** nel listato di un programma RAM si può calcolare tranquillamente la codifica.
- **completo:** in nessuno dei 3 passaggi rischiamo di andare in loop.

- **corretto:** *mantiene la semantica?* Vediamo che la semantica del programma compilato non è altro che la semantica dell'interprete chiamata sulla coppia, ma questa sappiamo essere l'esecuzione del programma n su x , che ha sua volta è esattamente l'esecuzione del programma P su x :

$$\Psi_{Comp(P)}(x) = \Psi_{I_W}(\langle x, n \rangle) = \varphi_n(x) = \varphi_P(x)$$

$$P \in PROG \rightarrow Comp(P) \in W - PROG$$

quindi è corretto.

Quindi è un compilatore da programma RAM a WHILE funzionante e corretto.

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

La presenza di questo compilatore ci presenta in maniera formale che i *goto* possono non essere utilizzati nella programmazione strutturata.

Teorema di Böhm Jacopini - 1970

Per ogni programma (RAM) con *goto* ne esiste uno equivalente in linguaggio strutturato (WHILE).

- Significa che la programmazione a basso livello può essere tranquillamente sostituita da quello ad alto livello.
- Quindi il *goto* va eliminato.

Conseguenza 1 $F(RAM) = F(WHILE)$

Sapevamo già che $F(WHILE) \supseteq F(RAM)$ (o viceversa) grazie alla presenza di un apposito compilatore, ed adesso sappiamo che grazie ad un interprete (che viene trasformato "rozzamente" in un compilatore) $F(WHILE) \subseteq F(RAM)$. Quindi quello che abbiamo ottenuto:

Che la potenza computazionale di una macchina piccola come quella RAM è esattamente identica a quella di una macchina WHILE molto più complessa, entrambe le macchine sono in grado di calcolare la stessa cosa.

$$F(RAM) = F(WHILE)$$

Utilizzando una concatenazione logica, sappiamo che grazie alla Gödelizzazione che la potenza computazionale di una macchina RAM è equinumerosa al numero di programmi, che a sua volta sono tanti quanti i numeri naturali (che non sono equinumerosi a tutte le funzioni calcolabili dai naturali ai naturali):

$$\mathbb{N}_\perp^{\mathbb{N}} \not\sim \mathbb{N} \sim PROG \sim F(RAM) = F(WHILE)$$

A questo punto la nozione di calcolabilità si capisce che è intrinseca ai problemi, questo perché ho provato sia a catturarla con la potenza computazionale di RAM che di WHILE, ma in entrambi i casi ho catturato la stessa cosa (\mathbb{N}). *Riesco a catturare il concetto di calcolabilità da un punto più astratto?*

Conseguenza 2: interprete universale

Un'altra conseguenza importante derivata dal compilatore e dall'interprete, sappiamo anche che il nostro interprete I_W è un programma RAM. Se io do in pasto questo interprete al compilatore di programmi WHILE, quello che ottengo è un programma RAM U .

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

La semantica di tale programma:

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

abbiamo scoperto che in RAM esiste un programma molto particolare che è in grado di prendere un qualsiasi programma RAM e di simularlo, e non è un caso che questo programma si chiami **interprete universale** RAM. Ovvero un programma scritto in quel linguaggio di programmazione che è in grado di simulare un qualsiasi altro programma scritto in quel linguaggio di programmazione.

Questo programma U rappresenta tutta la potenza di tutti i programmi RAM, perché è in grado di simularli tutti.

Riflessione sul concetto di calcolabilità

Sappiamo che esistono delle funzioni calcolabili e non calcolabili, siccome esistono delle funzioni non calcolabili è naturale cercare di caratterizzare ciò che è calcolabile, questo lo abbiamo fatto attraverso la macchina RAM e WHILE. Entrambe però catturano la stessa classe di funzioni calcolabili, *il concetto di calcolabile non è intrinseco ai problemi? quindi un concetto matematico non informatico*. Quindi a prescindere dallo strumento di calcolo, se riuscirò in questo intento allora risponderò una volta per tutte a questo quesito (non dipenderà mai dallo strumento di calcolo utilizzato). Ne verremo a conoscenza con il formalismo introdotto da S. Kleene.

4.2 Concetto di calcolabilità

Iniziamo a sospettare che questa idea di calcolabilità sorga dai problemi (*come caratteristica intrinseca dei problemi*) più che dal tipo di tecnologia che stiamo utilizzando. Quindi si cerca di astrarre il concetto che si vuole trattare, ovvero quello di **calcolabilità**. Quello che vogliamo fare è definire il concetto in modo matematico e assiomatico, in modo che ci siano solo riferimenti matematici (e non informatici), in modo che si possa utilizzare tutti gli strumenti matematici (che possono essere spostati in maniera diretta nel mondo informatico). Dobbiamo ripassare degli strumenti matematici e fissare un linguaggio comune:

4.2.1 Strumenti matematici (chiusure di insiemi)

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

$$op : U \times \cdots \times \Rightarrow U$$

che va da k -volte il cartesiano di U ad U . Il numero k di operazioni cartesiane non è altro che il numero di argomenti, detto anche **arietà**.

Per esempio

L'insieme universo è l'insieme dei numeri naturali, per esempio l'operazione somma è un esempio di operazione binaria, la somma è una funzione definita su \mathbb{N}^2 , poiché prende in ingresso due numeri e ne restituisce uno.

$$+ : \mathbb{N} \times \mathbb{N} \implies \mathbb{N} \text{ (operazione binaria)}$$

Anche la radice è un operazione (unaria)

$$\lfloor \sqrt{} \rfloor : \mathbb{N} \implies \mathbb{N} \text{ (operazione unaria)}$$

Un esempio di operazione n -aria è quella del proiettore su di t , non fa altro che restituirà la componente t -esima.

$$Proj_t^n : \mathbb{N} \times \cdots \times \mathbb{N} \implies \mathbb{N} \text{ (operazione n-aria)}$$

Tornando al nostro discorso, definiamo con **chiusura** di un insieme $A \subseteq U$ rispetto ad un operazione $op : U^k \implies U$, quando il risultato dell'operazione è ancora interno all'insieme A .

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

Esempio

L'insieme dei numeri pari è chiuso rispetto alla somma ($2+2=4$), la controparte invece no, l'insieme dei numeri dispari non è chiuso rispetto alla somma ($3+3=6$).

In generale questo concetto di chiusura può essere esteso ad un insieme di operazioni Ω , un insieme viene considerato chiuso se è chiuso per ciascuna operazione $\in \Omega$.

Esempio

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

L'insieme dei numeri pari è chiuso rispetto ad Ω ($2*4=8$ e $2+4=6$). Mentre l'insieme dei numeri dispari non è chiuso rispetto ad Ω poiché non è chiuso rispetto alla somma.

Consideriamo il seguente problema, sia $A \subseteq U$ e $op : U^k \implies U$. Quale è il più piccolo sottoinsieme di U che contiene A , che sia chiuso rispetto all'operazione op ? (Notare che sono tre condizioni e non due). Voglio allargare A il meno possibile in maniera che contenga A ma garantisca la chiusura rispetto ad op .

Alcune risposte ovvie:

- Se A è chiuso rispetto ad op , allora l'insieme cercato è A stesso.
- Sicuramente U soddisfa 1 e 2. Però U potrebbe avere il difetto di non essere il più piccolo (non è detto che sia il più piccolo).

Consideriamo il seguente esempio: sia $A = \{2, 3\} \subseteq \mathbb{N}$ come operazione abbiamo la somma $+ : \mathbb{N}^2 \rightarrow \mathbb{N}$, banalmente vediamo che A è un insieme che non può essere chiuso rispetto alla somma. Quale è il più piccolo sottoinsieme di \mathbb{N} che contiene A ed è chiuso rispetto a $+$? Sappiamo che sicuramente \mathbb{N} :

1. contiene A
2. è chiuso rispetto a $+$
3. non è il più piccolo che soddisfa le precedenti due condizioni (1 e 2).

Il problema è dato dal fatto che il risultato dell'operazione da un risultato che sarà al di fuori di A , ma se io stessi costruendo questo questo insieme man mano che eseguo l'operazione ed aggiungo il nuovo numero al sottoinsieme, al termine delle operazioni quello che avrò è il più piccolo sottoinsieme (approccio euristico costruttivo).

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

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

La definizione induttiva dice che nella base dell'induzione sono presenti tutti gli elementi originari in A^{op} . Ed ogni elemento a cui si applica l'operazione è un elemento presente all'interno dell'insieme. Questa definizione è identica alla definizione "più operativa" di A^{op} :

1. Metti in A^{op} tutti gli elementi di A .
2. Applica op a una k -tupla di elementi in A^{op}
3. Trovi un risultato che non è già in A^{op} allora aggiungilo ad A^{op}
4. Ripeti i punti 2 e 3 finché A^{op} cresce.
5. Output A^{op}

Esempi di chiusure

Voglio il più piccolo sottoinsieme di \mathbb{N} contenente $A = \{2, 3\}$ e chiuso rispetto a $+$ $\implies A^+ = ?$, come ottenerlo "operativamente"?

1. $A^+ \leftarrow A$
2. $(2 + 3) = 5 \notin A^+ \rightarrow A^+ = \{2, 3\}$
3. $(2 + 2), (3 + 3), (5 + 5), (2 + 5), (3 + 5), \dots \notin A^+ \implies A^+ = \{2, 3, 4, 5, 6, 7, 8, 10\}$
4. ...

Ottenendo in output $A^+ = \mathbb{N} \setminus \{0, 1\}$. Questo è anche dimostrabile banalmente con i risultati di un **equazione diofantea** che utilizza gli elementi iniziali dell'insieme di A .

Un altro esempio, voglio il più piccolo sottoinsieme di \mathbb{N} contenente $A = \{7, 10\}$ e chiuso rispetto a $-$ (**sottrazione troncata**).

$$A^- = \{0, 1, 2, \dots, 10\} = \mathbb{Z}_{11}$$

Tutti i numeri da 0 a 10 sono in grado di generare un numero all'interno dello stesso insieme, risultando chiuso rispetto all'operazione di sottrazione troncata.

Un altro esempio, voglio il più piccolo sottoinsieme di \mathbb{N} contenente $A = \{0\}$ che sia chiuso rispetto a $+$.

$$A^+ = \{0\} = A$$

L'insieme A è già chiuso rispetto alla somma.

Un'altro esempio, voglio il più piccolo sottoinsieme di \mathbb{N} contenente $A = \{0\}$ e chiuso per $\text{succ}(n) = n + 1$

$$A^{\text{succ}} = \mathbb{N} \text{ se } A = \{1\} \implies A^{\text{succ}} = \mathbb{N}^+$$

Lo stesso concetto di chiusura si può estendere rispetto ad un insieme di operazioni, $\Omega = \{op_1, \dots, op_t\}$, ogni operazione ha una proprietà arietà k_1, \dots, k_t di argomenti. Vogliamo il più piccolo sottoinsieme di U contenente A tale che sia chiuso rispetto ad Ω . Questo viene chiamato come **chiusura di A rispetto ad Ω** , ciò è l'insieme A^Ω definito induttivamente (induzione strutturale) come:

- $\forall a \in A \implies a \in A^\Omega$ (base)
- $\forall i \in \{1, \dots, t\}, \forall a_1, \dots, a_{k_i} \in A^\Omega \implies op_i(a_1, \dots, a_{k_i}) \in A^\Omega$
- Nient'altro è presente in A^Ω

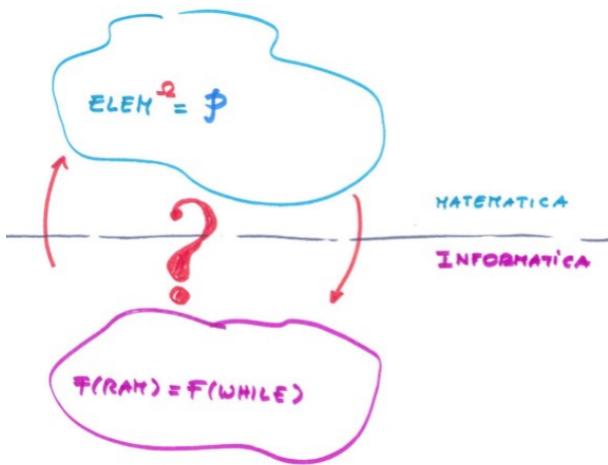
Anche qui ovviamente essendo una definizione induttiva di un insieme induttivo se devo dimostrare una proprietà su A^Ω la dimostro per induzione strutturale. Analogamente possiamo dare una definizione più "operativa" (o costruttiva), al caso dove $|\Omega| = 1$, quindi si effettua tutte le operazioni possibili

finché non sarà più possibile estendere l'insieme che si sta costruendo si avrà il più piccolo sottoinsieme di U che è chiuso rispetto all'insieme di operazioni Ω .

Questi sono esattamente i concetti che ci serviranno per definire in maniera astratta il concetto di calcolabilità (quindi che astrae da qualunque connotato informatico). Il nostro approccio sarà così strutturato:

1. Partiremo da un nucleo di tre funzioni $ELEM$ che saranno così facili che "qualunque" idea di calcolabile si voglia proporre le deve considerare calcolabili. Si vedrà subito che quelle tre funzioni non possono essere tutte le funzioni calcolabili, quindi sarà obbligatorio ampliare il nucleo di sicura calcolabilità.
2. Per l'ampliamento del nucleo introdurremo un insieme Ω di operazioni su funzioni, ovvero operazioni che prendono funzioni e restituiscono funzioni (per esempio la composizione di funzioni). Usiamo delle funzioni che mi generano altre funzioni, questo insieme Ω è fatto in maniera che sia calcolabile, sono banalmente implementabili in qualunque linguaggio di programmazione. Sono operatori che agiscono su cose calcolabili ($ELEM$) e sicuramente generano cose calcolabili data la loro semplice implementazione.
3. Calcolerò la chiusura di $ELEM^\Omega = \mathcal{P}$, il risultato sarà una classe di funzioni che sarà la nostra idea di calcolabilità data da un punto di vista puramente matematico, si chiamerà **classe delle funzioni ricorsive parziali**.

(approccio di Kleene).



Abbiamo due idee di calcolabilità una data dal punto di vista informatico e matematico, la domanda che ci dovremo porre è *l'approccio matematico è in grado di catturare interamente l'approccio informatico?* Dobbiamo trovare un punto

dove le due idee coincidono, quindi ritrovare ciò che abbiamo definito in maniera informatica con gli strumenti matematici (che saranno sempre quelli per l'eternità).

4.2.2 Il nucleo di funzioni calcolabili

L'insieme delle funzioni elementari (intuitivamente calcolabili) è composto da tre funzioni.

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

- La funzione successore è in grado di effettuare una somma atomica ad una certa quantità.
- La funzione zero che restituisce sempre 0.
- La funzione proiezione è in grado di restituire un k fissato da una n -upla.

La nostra idea di calcolabilità si origina da qui poiché sono funzioni banalmente calcolabili (un programma while è banalmente scrivibile). Questa classe delle funzioni elementari deve essere un punto da cui partire, poiché sono presenti delle funzioni che sono estremamente semplici da calcolare ma non presenti nel nucleo iniziale (es. predecessore, una funzione che sommi 2 anziché 1 ...).

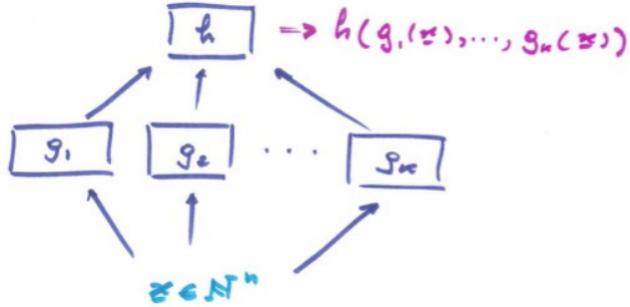
Quindi cosa devo fare? Devo ampliare la classe delle funzioni elementari. Quindi si pensa di mettere in campo degli operatori che prendono in ingresso delle funzioni e restituiscono delle funzioni. *Quale è il primo operatore che prende funzioni e restituisce funzioni?* La **composizione** di funzione, indicata con \circ prende in ingresso due funzioni e restituisce la composizione di due funzioni passata come argomento. Quella che vogliamo implementare è una composizione su più funzioni non solo 2.

L'operatore di composizione di funzioni

Sia $h : \mathbb{N}^k \rightarrow \mathbb{N}$ e $g_1, \dots, g_k : \mathbb{N}^n \rightarrow \mathbb{N}$; denoteremo $\underline{x} \in \mathbb{N}^n$. Definisco la composizione degli argomenti come $\text{COMP}(h, g_1, \dots, g_k) : \mathbb{N}^n \rightarrow \mathbb{N}$ definito come

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

Si parte dalla ennupla \underline{x} la si da in pasto alla prima fino alla k -esima, il risultato di ogni funzione viene dato in pasto alla funzione h per ottenere la composizione. Notiamo che il caso di una composizione con due funzioni è un caso particolare di questo che è quello generale.



L'operazione di composizione suona come qualcosa di implementabile, come potrebbe essere fatto un programma che calcola la composizione di funzioni. La mia idea di calcolabilità se viene espansa con questo operatore è una cosa corretta, perché parte da cose calcolabili per generare cose calcolabili.

Ampliamo ELEM chiudendo rispetto a COMP
 $ELEM^{COMP}$ effettivamente amplia $ELEM$, infatti:

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

Non fosse altro che la mia funzione $x+2$, è banalmente dimostrabile che $f(x)$ (la funzione che somma 2 ad x) appartiene ad $ELEM^{COMP}$, questo perchè basta comporre la funzione successore con se stessa. Infatti

$$f(x) = COMP(s, s)(x) = s(s(x)) = s(x + 1) = x + 2 = f(x)$$

Abbiamo visto che qualcosa in più riusciamo a calcolare, ma la domanda è *questa nuova classe di funzione è considerabile come la nuova classe di tutte le funzioni calcolabili oppure manca qualcosa?* Purtroppo questo primo ampliamento non basta a definire un intera classe di funzioni calcolabili. Per esempio la somma di due numeri non è calcolabile, ed è facile anche da dimostrare la non appartenenza ad $ELEM^{COMP}$. Devo ampliare ancora il mio insieme, perché sono presenti ancora funzioni troppo banali che devono essere considerate calcolabili.

L'operatore di ricorsione primitiva

A tutti è nota la definizione induttiva di fattoriale di n , il concetto di questo operatore matematico è descrivibile induttivamente con la seguente espressione

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

Allora introduciamo l'operatore di ricorsione primitiva RP . Per definire tale operatore necessito di due funzioni argomento g, h (una con una singola

ennupla come argomenti, l'altra con una ennupla più altri due argomenti).

$$g : \mathbb{N}^n \rightarrow \mathbb{N}$$

$$h : \mathbb{N}^{n+2} \rightarrow \mathbb{N}$$

rispettivamente $g(x)$ e $h(z, y, \underline{x})$ con $\underline{x} \in \mathbb{N}^n$. L'operatore di ricorsione primitiva prende in ingresso due funzioni e restituisce una terza funzione ottenuta per **ricorsione** delle altre due. La ricorsione primitiva applicata ad h e g restituisce una nuova funzione f che dipende da una sola ennupla \underline{x} e da un solo parametro y che è definita

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

Notiamo che nel caso in cui il parametro y sia 0, la funzione $g(x)$ non è altro che il caso base, ma ricordiamo che effettivamente g può essere una qualsiasi funzione. Mentre quando il parametro y è positivo notiamo che la funzione restituita da RP è la composizione della funzione h su f su valori y minori (**passo induttivo**). In particolare si vede che è una **induzione** che generalizza il concetto di ricorsione.

Questa generalizzazione ci permette di definire funzioni in termini di se stesse, le definizioni ricorsive a cui siamo abituati sono casi particolari di questo caso generale. Sia risaputo che tutti i metodi per la risoluzione di equazioni differenziali utilizzano questa forma tabella come risoluzione.

Ora ampliamo ulteriormente *ELEM*

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

la chiusura comincia a creare una classe abbastanza consistente di funzioni, che contiene anche molte funzioni complicate (anche quelle differenziali). Prende il nome come **classe delle funzioni ricorsive primitive**. Questa classe contiene la funzione somma tra due numeri.

$$somma(x, y) = \begin{cases} x = Pro_1^2(x, y) & y = 0 \\ s(somma(x, y - 1)) & y > 0 \end{cases}$$

questo utilizzando il successore ed il proiettore, ma è anche possibile definire il prodotto riutilizzando la somma (e via via anche l'esponenziale).

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

possiamo definire l'operazione di $\dot{-}$ definendo a priori l'operatore di predecessore

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

Si può facilmente notare che la classe delle funzioni ricorsive primitive è una classe importante che ricopre molte funzioni.

4.2.3 $RICPRIM$ vs $F(WHILE)$

$RICPRIM$ contiene molte funzioni e potrei già chiedermi se ho raggiunto $F(WHILE)$. Mostreremo che $RICPRIM \subseteq F(WHILE)$ per induzione strutturale. Vogliamo capire che tipo di relazione c'è tra la mia idea di calcolabilità matematica e informatica. Quello che mostreremo è che questa idea di calcolabilità matematica è inclusa all'interno di quella informatica.

Ovvero che ogni funzione presente in $RICPRIM$ sia $WHILE$ -programmabile, ovvero che sia possibile scrivere un programma $WHILE$ che la calcoli. Significa che siamo sulla strada giusta che ci stiamo muovendo su qualcosa di significativo.

Possiamo notare che però in $RICPRIM$ sono presenti funzioni totali, ovvero che terminano, le funzioni non possono andare in loop, quindi l'inclusione dovrà essere per forza propria se dimostrata. La dimostrazione seguirà i seguenti punti:

- Definizione induttiva di $RICPRIM = ELEM^{COMP,RP}$
 1. Le funzioni in $ELEM$ sono in $RICPRIM$
 2. Se prendo delle funzioni $RICPRIM$ allora la loro composizione è $RICPRIM$ anch'essa.
 3. Se prendo delle funzioni $RICPRIM$ allora la loro composizione mediante l'operatore di ricorsione primitiva genera una funzione RP .
 4. Nient'altro è in $RICPRIM$

Quindi data questa definizione induttiva abbiamo dimostrato che $ELEM \subseteq F(WHILE)$, proseguiò sempre per induzione sui seguenti punti:

- Assumo per induzione che $h, g_1, \dots, g_k \in RICPRIM$ siano in $F(WHILE)$ e dimostro che $COMP(h, g_1, \dots, g_k) \in F(WHILE)$
- Assumo per ipotesi induttiva che $g, h \in RICPRIM$ siano in $F(WHILE)$ e dimostro che $RP(g, h) \in F(WHILE)$.

allora per induzione avrò dimostrato che le ricorsive primitive sono un sottoinsieme dell'insieme $WHILE$.

4.2.4 $RICPRIM \subseteq F(WHILE)$

- passo base: $ELEM \subseteq F(WHILE)$ (sappiamo calcolare funzione successore, zero e proiezione k-esima con un programma WHILE).
- passi induttivi:
 - Consideriamo delle funzioni h, g_1, \dots, g_k per ipotesi induttiva assumo che siano $WHILE$ -programmabili ($\in F(WHILE)$). Ovvero che per ciascuna funzione esisti un rispettivo programma $H, G_1, \dots, G_k \in$

$W - PROG$ che le calcola. Ovvero la cui semantica sia corrisponda esattamente alle funzioni originali.

$$\Psi_H = h, \Psi_{G_1}, \dots, \Psi_{G_k} = g_k$$

allora devo mostrarti un programma WHILE che calcoli la loro composizione.

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

```

W ∈ begin
  x₀ := G₁(x₁);
  x₀ := [x₀, G₂(x₁)];
  :
  x₀ := [x₀, Gₖ(x₁)];
  x₀ := H(x₀);
end

```

Figura 4.16: Programma WHILE che calcola la composizione

Supponiamo che in x_1 ci sia il nostro input del programma, in x_0 calcolo il primo argomento. In realtà $G_1(x_1)$ è una macro del WHILE che è in grado di calcolare la sua corrispettiva funzione g_1 che esistere per ipotesi induttiva. Poi in x_0 ci condenso il secondo argomento G_2 , continuo in questa maniera fino al k -esimo argomento. In x_0 ottengo la condensazione di tutti gli argomenti su un solo numero, seguendo lo schema di Cantor (non è un problema se abbiamo un numero finito di variabili, i motivi sono già stati spiegati).

adesso il primo passo induttivo è stato dimostrato, ovvero che la composizione di funzioni è WHILE-programmabile.

- passo induttivo sull'operatore di ricorsione primitiva: per ipotesi induttiva assumo che le due rispettive funzioni g, h siano WHILE-programmabili. Che quindi esistano $G, H \in W - PROG$ con $\Psi_G = g$ e $\Psi_h = h$. Allora riesco a scrivere un programma WHILE che riesce a calcolare l'operazione di ricorsione primitiva

```

W := input (x, y)
begin
  t := G(x);
  k := 1;
  while k <= y do
    begin
      t := H(t, k-1, x);
      k := k + 1;
    end
end

```

questo è un programma perfettamente legale che calcola la ricorsione primitiva di due funzioni che sono WHILE-programmabili.

Allora ho dimostrato che per induzione strutturale che la classe delle funzioni ricorsive primitive è all'interno delle funzioni WHILE

$$RICPRIM \subseteq F(WHILE)$$

A sto punto ci chiediamo se questo sotto insieme sia proprio oppure no. Questo però è facile da dimostrare, tutte le RICPRIM sono funzioni totali:

- tutte le funzioni elementari sono totali.
- la composizione delle funzioni totali è ancora una funzione totale

Chiaramente $F(WHILE)$ contiene anche funzioni **parziali** (che non terminano), questo grazie ad i loop che possono non far terminare i programmi.

$$RICPRIM \not\subseteq F(WHILE)$$

Significa che devo ampliare ulteriormente RICPRIM per raggiungere $F(WHILE)$

Ulteriori considerazioni su RICPRIM

Consideriamo questo while, notiamo che è possibile riscrivere la stessa cosa con una sintassi semplificata quella del for. Le RICPRIM caratterizzano la potenza computazionale con un linguaggio più semplice di quello WHILE, ovvero il FOR language, però come un for del PASCAL dove è presente una variabile di controllo che scandisce le iterazioni, che parte da un certo valore e termina in un altro valore e che non può essere toccata nel FOR. Prendiamo tale FOR-language, e vediamo che le RICPRIM sono esattamente la classe delle funzioni calcolate da questo linguaggio che ha un numero di iterazioni che è fissato e non potrà mai andare all'infinito (non stiamo considerando for loop del C/Java ma quello del PASCAL).

$$RICPRIM = F(FOR) \not\subseteq F(WHILE)$$

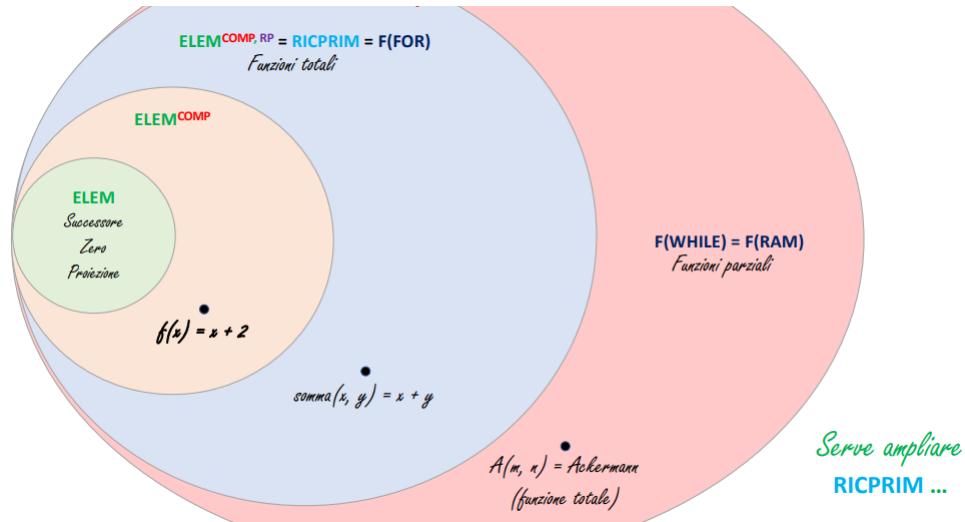
Questa è una inclusione propria.

Ora effettuiamo un ulteriore considerazione, ipotizziamo di usare un linguaggio while che non possa andare in loop, quindi mi restringo ad una classe $\tilde{F}(\text{WHILE})$ di funzioni while che non possono andare in loop. *Adesso come sarà il confronto tra il FOR language ed il WHILE indebolito?* L'inclusione è propria?

Le RICPRIM sono incluse propriamente nella classe delle funzioni $\tilde{F}(\text{WHILE})$, questo è dimostrabile sia per diagonalizzazione che utilizzando la funzione di Ackermann. Una funzione di Ackermann (1928) è una funzione che non è ricorsiva primitiva ma può essere calcolata in un programma WHILE da un programma che termina. Questa funzione sfrutta una doppia induzione, perché lo schema ricorsivo della funzione di Ackermann non è lo stesso di quello delle RICPRIM.

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

Se usiamo solamente la composizione di RICPRIM non siamo in grado di esprimere tale classe di funzioni. L'idea è che le RICPRIM hanno un determinato tasso di crescita che è nettamente inferiore rispetto alle funzioni di Ackermann. N.B.: Già nel 1927 Sudau aveva proposto una funzione con le stesse caratteristiche di quella di Ackermann... Occorre ampliare RICPRIM...



Operatore di minimalizzazione di funzioni

Vogliamo ampliare ancora di più l'attuale classe di funzioni, introduciamo il nuovo operatore di minimalizzazione. Sia $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$, $f(\underline{x}, y)$ con $\underline{x} \in \mathbb{N}^n$. La minimalizzazione di una funzione ad $n + 1$ genera una funzione ad n

argomenti, definita come:

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

Nella prima condizione y è tale da essere il più piccolo valore che azzeri la funzione $f(\underline{x}, y)$. Vediamo alcuni esempi di minimalizzazione di funzioni:

$$\begin{array}{c|c} \hline x + y + 1 & \perp \\ \hline x - y & x \\ \hline y - x & 0 \\ \hline x - y^2 & \lceil \sqrt{x} \rceil \\ \hline \end{array}$$

- La minimalizzazione di $x + y + 1$ è una funzione $g(x)$ che dato un certo x mi deve trovare il più piccolo y tale che $x + y + 1 = 0$ (attenzione che è una funzione che va da naturali a naturali, quindi i numeri negativi non possiamo considerarli). Qualsiasi sia questo numero nella parte sinistra avremo sempre 1, perché con qualunque valore di x sia fissato ed y (naturali) non azzererà la funzione.
- nel caso della sottrazione troncata il più piccolo y che azzerà x è x stesso, notare che lo si potrebbe azzerare anche con un $x + 1, +2, \dots$ ma questo non sarebbe il più piccolo.
- invertendo l'ordine degli operandi il più piccolo y che azzerà la funzione è 0.
- nell'ultimo caso è la parte superiore della radice di x (la radice quadrata è una operazione presente in RICPRIM).

Vediamo che con l'operatore di minimalizzazione otteniamo delle funzioni interessanti, adesso proviamo a chiudere rispetto a queste funzioni.

$$\text{RICPRIM}^{\text{MIN}} = \text{ELEM}^{\text{COMP,RP,MIN}} = \mathcal{P} = \text{funzioni ricorsive parziali}$$

Questa nuova classe di funzioni è detta **classe delle funzioni ricorsive parziali** \mathcal{P} . Sicuramente \mathcal{P} , che grazie a MIN contiene anche funzioni parziali, amplia RICPRIM fatto solo da funzioni totali. Ma come si pone rispetto a $F(\text{WHILE})$?

4.2.5 $\mathcal{P} \subseteq F(\text{WHILE})$

Riesco a dimostrare che sia un sottoinsieme (non so se sia tanto quanto $F(\text{WHILE})$), possiamo dimostrare tenendo conto della natura induttiva delle ricorsive parziali che sono la chiusura delle ricorsive primitive rispetto all'operatore di minimalizzazione.

1. Le funzioni in $RICPRIM$ sono tutte in \mathcal{P}
2. Se f è una \mathcal{P} allora la minimalizzazione di f è una ricorsiva parziale.
3. Nient'altro è in \mathcal{P}

Allora cosa faccio per dimostrare che $P \subseteq F(WHILE)$? Dimostro che le ricorsive primitive che sono ricorsive parziali sono $WHILE$ -programmabili (ma questo lo abbiamo già dimostrato precedentemente).

L'unico punto veramente da dimostrare è il passo induttivo, ovvero che il nuovo oggetto che andiamo a costruire sia $WHILE$ -programmabile.

1. Dimostrare che le $RICPRIM$ sono $WHILE$ -programmabili (fatto).
2. Sia $f \in \mathcal{P}$ una funzione $WHILE$ -programmabile per ipotesi induttiva.
Allora, esiste un programma $WHILE$ che calcola la minimalizzazione di quella funzione $MIN(f)$ (da dimostrare).

Dimostrazione che $MIN(f)$ è $WHILE$ -programmabile

Sia $f(\underline{x}, y) \in \mathcal{P}$; per ipotesi induttiva, sia f calcolata dal programma $WHILE$ F . Allora devo mostrarti un programma $WHILE$ tale che sia in grado di calcolare tale funzione:

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

questo è un programma che prende in ingresso x e va a cercare il più piccolo y tale per cui la funzione si azzeri.

```

input(  $\underline{x}$  )
begin
  y:=0; // output MIN(f)
  while F( $\underline{x}$ , y) != 0 do
    y:=y+1;
end
  
```

Figura 4.17: Programma $WHILE$ che implementa la minimalizzazione di una funzione

Sia tale che questa dicitura è legale, poiché F è un programma $WHILE$, se però y dovesse mancare questo programma andrà in loop (semantica indeterminata, \perp). Si introduce la possibilità di programmi che vanno in loop.

Per induzione strutturale abbiamo mostrato che $\mathcal{P} \subseteq F(WHILE)$. Sorge una domanda, la nostra "idea teorica" di calcolabilità \mathcal{P} raggiungerà quella "pratica" di $F(WHILE)$? Può valere $F(WHILE) \subseteq \mathcal{P}$?

Nel caso in cui si possa dimostrare tale inclusione (e quindi poi dimostrare che $F(WHILE) = \mathcal{P}$).

4.2.6 Dimostrazione $F(WHILE) \subseteq \mathcal{P}$

Vogliamo dimostrare $F(WHILE) = \{\Psi_W : W \in W - PROG\} \subseteq \mathcal{P} = ELEM^{COMP,RP,MIN}$
le funzioni WHILE programmabili sono anche delle ricorsive parziali.

$$\Psi_W \in F(WHILE) \implies \Psi_W \in \mathcal{P} = ELEM^{COMP,RP,MIN}$$

Quindi prendiamo una funzione tale e dimostriamo che essa sia ricorsiva parziale, ma questo per definizione vuol dire dimostrare che Ψ_W può essere ottenuta come composizione, ricorsione primitiva e minimalizzazione a partire da funzioni in $ELEM$.

Se riesco a dimostrare questo allora ho dimostrato che la semantica di un qualsiasi programma WHILE è ricorsiva parziale.

Ricordiamo come è definita la semantica di un programma WHILE si definisce a partire dall'input x inserito in una funzione che inizializza la macchina (delle 21 variabili disponibili $x_1 = x$ ed il resto impostate a 0). Dopo di che si fa agire la semantica nel programma W , la quale è la funzione stato prossimo, ci restituisce lo stato successivo. L'output è lo stato finale che viene salvato sul registro/variabile 0 delle 21 disponibili, questo significa effettuare la proiezione.

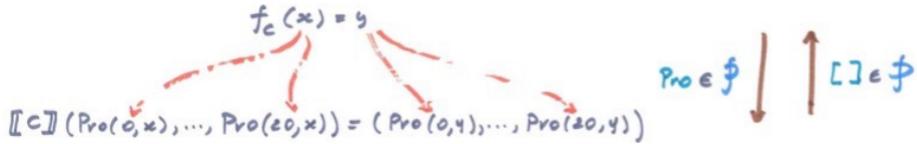
$$\Psi_W(x) = Pro_0^{21}([W](W - in(x)))$$

(la funzione stato prossimo $[C](\underline{x}) = \underline{y}$ calcola lo stato $\underline{y} \in \mathbb{N}^{21}$ a seguito dell'esecuzione del comando WHILE C partendo dallo stato $\underline{x} \in \mathbb{N}^{21}$).

Quindi vogliamo dimostrare che $\Psi_W(x) = Pro_0^{21}([W](W - in(x)))$ faccia parte delle ricorsive parziali, noto che la semantica di un programma WHILE si ottiene effettuando la composizione di una funzione elementare (la proiezione) con la funzione stato prossimo. Allora questo significa che posso rappresentare con le ricorsive parziali la funzione stato prossimo, vediamo perché:

1. $Pro_0^{21} \in ELEM \implies Pro_0^{21} \in \mathcal{P}$
2. \mathcal{P} è chiuso rispetto alla composizione
3. Questo dimostra che $[C](\underline{x}) = \underline{y} \in \mathcal{P}$ allora $\Psi_W \in \mathcal{P}$. Qui però c'è da aggiungere un dettaglio tecnico, la funzione stato prossimo è definita come $[\cdot] : \mathbb{N}^{21} \rightarrow \mathbb{N}^{21}$ ha come codominio \mathbb{N}^{21} (funzione che va da stati in stati). Mentre \mathcal{P} per come le abbiamo definite hanno come codominio \mathbb{N} . *Questo è un problema?* a dire il vero no, sappiamo che possiamo esprimere ciò con una codifica di 21 elementi utilizzando la lista di Cantor.

Quindi quello che faremo non sarà dimostrare che una funzione da vettore a vettore appartiene alle funzioni ricorsive ($[C](\underline{x}) = \underline{y} \in \mathcal{P}$ con $\underline{x}, \underline{y} \in \mathbb{N}^{21}$). Quanto $f_C(x) = y \in \mathcal{P}$ con $x = [\underline{x}]$ e $y = [\underline{y}]$, ovvero la funzione correlata con gli stati che non sono visti come vettori ma la codifica di array di 21 elementi (ovvero gli stati contenuti nelle 21 variabili) secondo Cantor.



Quindi diciamo che se dimostro che f_C è ricorsiva parziale allora anche $[C]$ è ricorsiva parziale. Utilizzando le parentesi quadre per impacchettare gli stati e le proiezioni per spacchettarli (importante sottolineare che queste operazioni sono sicuramente funzioni elementari $\in ELEM$).

Dimostrazione che $f_C \in \mathcal{P}$

Dimostrazione per induzione sul comando C che f_C è ricorsiva parziale, quindi prima di tutto vado ad individuare f_C sulla base del linguaggio *WHILE* ovvero gli assegnamenti.

- $C \equiv x_k := 0$ (azzeramento della componente k)

$$f_{x_k:=0}(x) = [Pro(0, x), \dots, 0, \dots, Pro(20, x)]$$

tutte le operazioni qui effettuate sono ricorsive parziali, le funzioni di proiezione sono elementari, fondere con le parentesi quadra consiste in una serie di somme. Quindi lo stato prossimo di un azzeramento è una funzione che globalmente è ricorsiva parziale.

- $C \equiv x_k := x_j + / \dot{-} 1$ (incremento/decremento di 1)

$$f_{x_k:=x_j+/-1} = [Pro(0, x), \dots, Pro(j, x) + / \dot{-} 1, \dots, Pro(20, x)]$$

anche qui molto similmente, prendiamo lo stato x e lo spacchettiamo, raggiungiamo la componente k -esima e gli sommiamo o decrementiamo di 1 il valore. Il risultato è un numero che è la "condensa" dello stato prossimo (identico a prima solo che incremento). Anche qui sono coinvolte solo funzioni ricorsive parziali, quindi lo stato prossimo per l'operatore di incremento decremento è una funzione ricorsiva parziale.

$$f_{x_k:=x_j+/-1}(x) \in \mathcal{P}$$

adesso vediamo la funzione stato prossimo per i comandi composti

- $C \equiv \underline{\text{begin}} \ C_1; \dots; C_m \ \underline{\text{end}}$ ipotizzando induttivamente che $f_C \in \mathcal{P}$ (la funzione composta) $f_C(x) = f_{C_m}(\dots f_{C_1}(x) \dots)$ effettuo la composizione, ovvero applico la funzione diverse volte sulla funzione degli altri comandi composti. Però noi sappiamo che la funzione composizione è una funzione ricorsiva parziale (questo per ipotesi induttiva).

- $C' \equiv \text{while } x_k \neq 0 \text{ do } C$, consideriamo sempre per ipotesi induttiva che la funzione composta faccia parte delle ricorsive parziali $f_c \in \mathcal{P}$

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

ovvero applichiamo il comando C nello stato x un numero e di volte, tale per cui questo numero sia il più piccolo numero di volte che serve per azzerare la variabile k .

Inizio a notare qualcosa che mi piace, ci sono operazioni di minimizzazione e proiezione, questi sono costrutti ricorsivi parziali.

Siamo attualmente tentati da dire che la semantica del *WHILE* sia ricorsiva parziale, ma c'è qualcosa che non va. Ma questa composizione non è un costrutto ricorsivo parziale, questo perché questa composizione è parametrizzata rispetto ad y . Il nostro operatore *COMP* è definito su un numero costante di argomenti, qui abbiamo un numero di argomenti **variabili**.

Quindi dato che $f_C^{e(x)}$ è la composizione di f_C per $e(x)$ volte, che non è un numero costante. Io so scrivere $\text{COMP}(h, g_1, \dots, g_k)$ con k costante. Quindi ci si chiede *come rappresentare in \mathcal{P} la composizione di una funzione con se stessa un numero non costante di volte?* ovvero l'operatore $T(x, y) = f_C^y(x)$

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

Da notare che questo costrutto è ricorsivo parziale, perchè è ottenuto attraverso una ricorsione primitiva sfruttando delle funzioni f_C che sappiamo per induzione essere ricorsive parziali. Ed è proprio quello che ci serviva, posso riscrivere utilizzando l'operatore ricorsivo parziale:

$$e(x) = \mu y(\text{Pro}(x, T(x, y)) = 0)$$

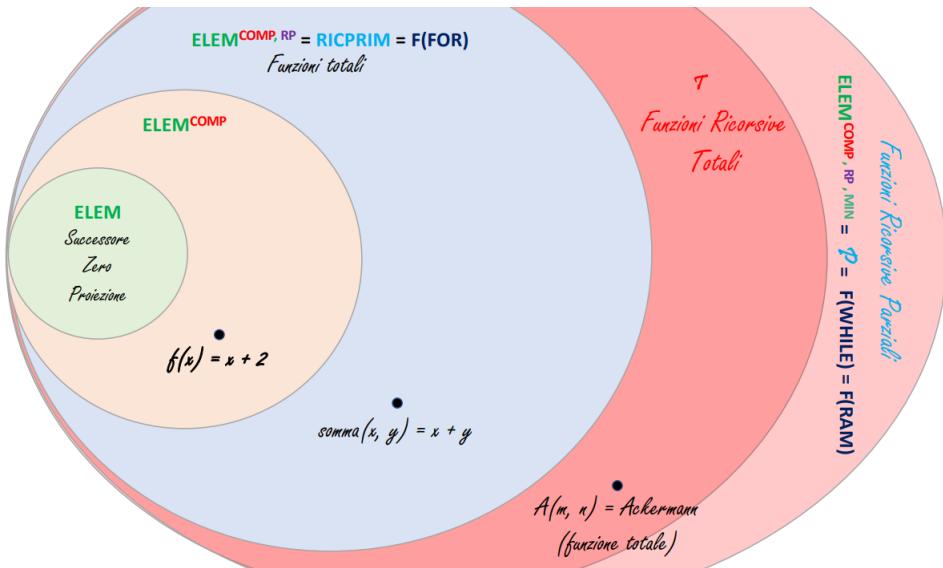
Questa è la minimizzazione dell'esponente della composizione di funzione, per esprimere la forma completa mi bastare dire:

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

anche lo stato prossimo del comando *while* è esprimibile con costrutti ricorsivi parziali, quindi abbiamo dimostrato che $F(\text{WHILE}) \subseteq \mathcal{P}$

Allora, abbiamo dimostrato che

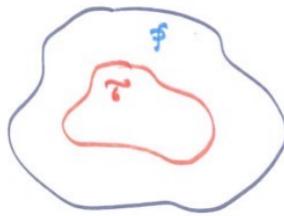
$$F(\text{WHILE}) = \mathcal{P}$$



Abbiamo caratterizzato matematicamente ciò che prima avevamo definito con un linguaggio ed una macchina. Da un punto di vista teorico i linguaggi $F(WHILE)$ e $F(RAM)$ si chiamano **classe delle funzioni ricorsive parziali**.

Esiste un interessante sottoclasse delle funzioni ricorsive parziali, la **classe delle funzioni ricorsive totali** indicata con \mathcal{T} .

4.2.7 Tesi di Church-Turing



Qualunque modello di calcolo sintetizzato negli anni '30 che individuasse le cose calcolabili va sempre ad attingere dalla classe delle funzioni ricorsive parziali: WHILE, RAM, Macchine di Turing, λ -calcolo, Grammatiche, ...

Tutti questi punti di vista diversi della calcolabilità hanno sempre detto che la classe delle funzioni calcolabili è sempre quella delle ricorsive parziali \mathcal{P} . Per questo è stata proposta la tesi di Church-Turing (entrambi hanno pensato la stessa cosa): la classe delle funzioni intuitivamente calcolabili coincide con la classe delle funzioni ricorsive parziali.

Tutto ciò che pensi sia calcolabile sarà sempre catturato dalla classe delle funzioni ricorsive parziali. *Perché non si parla di teorema?* Perché non è presente

una caratterizzazione di tutti i modelli di calcolo presenti, la vediamo come una presa di posizione prudente. Fin'ora è una tesi che è stata sempre confermata da un qualsiasi modello di calcolo ragionevole.

In altre parole, aderendo alla tesi di Church un sinonimo di "calcolabile" è "*ricorsivo parziale*", mentre se dico che un compito è "*ricorsivo totale*" intendo che è calcolabile da programmi che non possono andare mai in loop (ovvero che si arrestano su ogni input).