Informatica Teorica

Massimo Perego

Indice

Introduzione

Si "contrappone" all'informatica applicata, ovvero qualsiasi applicazione dell'informatica atta a raggiunger uno scopo, dove l'informatica è solamente lo strumento per raggiungere in maniera efficace un obiettivo.
Con "*informatica teorica*" l'oggetto è l'informatica stessa, si studiano i fondamenti della disciplina in modo rigoroso e scientifico. Può essere fatto ponendosi delle questioni fondamentali: il *cosa* e il *come* dell'informatica, ovvero cosa è in grado di fare l'informatica e come è in grado di farlo.

Cosa: L'informatica è "la disciplina che studia l'informazione e la sua elaborazione automatica", quindi l'oggetto sono l'informazione e i dispositivi di calcolo per gestirla; scienza dell'informazione. Diventa lo studio come risolvere automaticamente un problema. Ma tutti i problemi sono risolvibili in maniera automatica? Cosa è in grado di fare l'informatica?

La branca dell'informatica teorica che studia cosa è risolvibile si chiama **Teoria della Calcolabilità**, studia cosa è calcolabile per via automatica. Spoiler: non tutti i problemi sono risolvibili per via automatica, e non potranno mai esserlo per limiti dell'informatica stessa. Cerchiamo una caratterizzazione generale di cosa è calcolabile e cosa no, si vogliono fornire strumenti per capire ciò che è calcolabile. La caratterizzazione deve essere fatta matematicamente, in quanto il rigore e la tecnica matematica permettono di trarre conclusioni sull'informatica.

Come: Una volta individuati i problemi calcolabili, come possiamo calcolarli? Il dominio della **Teoria della Complessità** vuole descrivere le risoluzione dei problemi tramite mezzi automatici in termini di risorse computazionali necessarie. Una "risorsa computazionale" è qualsiasi cosa che viene consumata durante l'esecuzione per risolvere il problema, come pos-

sono essere elettricità o numero di processori, generalmente i parametri più importanti considerati sono tempo e spazio di memoria. Bisognerà definire in modo preciso cosa si intende con "tempo" e "spazio". Una volta fissati i parametri bisogna definire anche cosa si intende con "risolvere efficientemente" un problema, in termini di tempo e spazio.

La teoria della calcolabilità dice quali problemi sono calcolabili, la teoria della complessità dice, all'interno dei problemi calcolabili, quali sono risolvibili efficientemente.

Capitolo 1

Teoria della Calcolabilità

1.1 Notazione

1.1.1 Funzioni

Funzione: Una funzione f dall'insieme A all'insieme B è una legge che dice come associare a ogni elemento di A un elemento di B. Si scrive

$$f : A \to B$$

E chiamiamo A dominio e B codominio. Per dire come agisce su un elemento si usa $f(a) = b$, b è l'immagine di a secondo f (di conseguenza a è la controimmagine).
Per definizione di funzione, è possibile che elementi del codominio siano raggiungibili da più elementi del dominio, ma non il contrario. Possiamo classificare le funzioni in base a questa caratteristica:

- **Iniettiva:** $f : A \to B$ è iniettiva sse $\forall a, b \in A,\ a \neq b \implies f(a) \neq f(b)$
- **Suriettiva:** $f : A \to B$ è suriettiva sse $\forall b \in B,\ \exists a \in A : f(a) = b$: un altro modo per definirla è tramite l'insieme immagine di f, definito come
  $$\mathrm{Im}_f = \{b \in B : \exists a, f(a) = b\} = \{f(a) : a \in A\}$$
  Solitamente $\mathrm{Im}_f \subseteq B$, ma f è suriettiva sse $\mathrm{Im}_f = B$;
- **Biettiva:** $f : A \to B$ è biettiva sse è sia iniettiva che suriettiva, ovvero
  $$\begin{array}{ll} \forall a, b \in A, a \neq b : & f(a) \neq f(b) \\ \forall b \in B, \exists a \in A : & f(a) = b \end{array} \implies \forall b \in B, \exists! a \in A : f(a) = b$$

Inversa: Per le funzioni biettive si può naturalmente associare il concetto di "inversa": dato $f : A \to B$ biettiva, si definisce inversa la funzione $f^{-1} : B \to A$ tale che $f^{-1}(b) = a \Leftrightarrow f(a) = b$.

Composizione di funzioni: Date $f : A \to B$ e $g : B \to C$, f composto g è la funzione $g \circ f : A \to C$ definita come $g \circ f(a) = g(f(a))$. Generalmente non commutativo, $f \circ g \neq g \circ f$, ma è associativo.

Funzione identità: Dato l'insieme A, la funzione identità su A è la funzione $i_A : A \to A$ tale che $i_A(a) = a$, $\forall a \in A$.

Un'altra possibile definizione per l'inversa diventa:

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

Funzioni Parziali: Se una funzione $f : A \to B$ è definita per $a \in A$ si indica con $f(a) \downarrow$ e da questo proviene la categorizzazione: una funzione è **totale** se definita $\forall a \in A$, **parziale** altrimenti (definita solo per qualche elemento di A).

Insieme Dominio: Chiamiamo **dominio** (o campo di esistenza) di f l'insieme

$$\mathrm{Dom}_f = \{a \in A | f(a) \downarrow\} \subseteq A$$

Quindi se $\mathrm{Dom}_f = A$ la funzione è totale, se $\mathrm{Dom}_f \subsetneq A$ allora è una funzione parziale.

Totalizzazione: Si può **totalizzare una funzione parziale** f definendo una funzione a tratti $\overline{f} : A \to B \cup \{\perp\}$ tale che

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

Dove $\perp$ è il **simbolo di indefinito**, per tutti i valori per cui la funzione di partenza f non è definita. Da qui in poi $B_\perp$ significa $B \cup \{\perp\}$.

Insieme delle funzioni: L'insieme di tutte le funzioni che vanno da A a B si denota con

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

La notazione viene usata in quanto la cardinalità di B^A è esattamente $|B|^{|A|}$, con A e B insiemi finiti.
Volendo includere anche tutte le funzioni parziali:

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

Le due definizioni coincidono, $B^A = B_\perp^A$, ma quest'ultima permette di mettere in evidenza che tutte le funzioni presenti sono totali o totalizzate.

1.1.2 Prodotto Cartesiano

Chiamiamo **prodotto cartesiano** l'insieme

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

Rappresenta l'insieme di tutte le coppie ordinate di valori in A e B. In generale non è commutativo, a meno che $A = B$.

Può essere esteso a n-uple di valori:

$$A_1 \times \cdots \times A_n = \{(a_1, \ldots, a_n) | a_i \in A_i\}$$

Il prodotto di n volte lo stesso insieme verrà, per comodità, indicato come

$$A \times \cdots \times A = A^n$$

Proiettore: Operazione "opposta", il proiettore i-esimo è una funzione che estrae l'i-esimo elemento di una tupla, quindi è una funzione

$$\pi_i : A_1 \times \cdots \times A_n \to A_i \text{ t.c. } \pi_i(a_1, \ldots, a_n) = a_i$$

La proiezione sull'asse in cui sono presenti i valori dell'insieme a_i.

1.1.3 Funzione di Valutazione

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

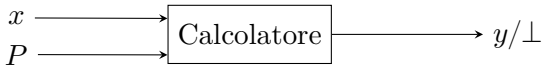
$$\omega : B_\perp^A \times A \to B \text{ t.c. } \omega(f, a) = f(a)$$

Prende una funzione f e la valuta su un elemento a del dominio. Si possono fare due tipi di analisi su questa funzione:

- Fisso a e provo tutte le f, ottenendo un *benchmark* di tutte le funzioni su a
- Fisso f e provo tutte le a del dominio, ottenendo il *grafico* di f

1.2 Sistemi di Calcolo

Vogliamo modellare teoricamente un **sistema di calcolo**; quest'ultimo può essere visto come una black box che prende in input un programma P, dei dati x e calcola il risultato y di P su input x. La macchina restituisce y se è riuscita a calcolare un risultato, $\perp$ (indefinito) se è entrata in un loop.



Quindi, formalmente, possiamo definire un sistema di calcolo come una funzione

$$\mathcal{C} : \text{PROG} \times \text{DATI} \to \text{DATI}_\perp$$

Possiamo vedere un sistema di calcolo come una funzione di valutazione:

- i dati x corrispondono all'input a
- il programma P corrisponde alla funzione f

Formalmente, un programma $P \in \text{PROG}$ è una sequenza di regole che trasformano un dato input in uno di output, ovvero l'espressione di una funzione secondo una sintassi

$$P : \text{DATI} \to \text{DATI}_\perp$$

e di conseguenza $P \in \text{DATI}_\perp^{\text{DATI}}$. In questo modo abbiamo mappato l'insieme PROG sull'insieme delle funzioni, il che ci permette di definire il sistema di calcolo come la funzione

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

Analoga alla funzione di valutazione. Con $\mathcal{C}(P, x)$ indichiamo la funzione calcolata da P su x dal sistema di calcolo $\mathcal{C}$, che viene detta **semantica**, ovvero il suo "significato" su input x.

Il modello solitamente considerato quando si parla di calcolatori è quello di **Von Neumann**.

1.3 Potenza Computazionale

Indicando con

$$\mathcal{C}(P, \_) : \text{DATI} \to \text{DATI}$$

la funzione che viene calcolata dal programma P (semantica di P).

La **potenza computazionale** di un calcolatore è definita come l'insieme di tutte le funzioni che quel sistema di calcolo è in grado di calcolare, ovvero

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

Ovvero, l'insieme di tutte le possibili semantiche di funzioni calcolabili con il sistema $\mathcal{C}$. Stabilire il carattere di quest'ultima inclusione equivale a stabilire *cosa può fare l'informatica*:

- se $F(\mathcal{C}) \subsetneq \text{DATI}_{\perp}^{\text{DATI}}$ allora esistono compiti **non automatizzabili**
- se $F(\mathcal{C}) = \text{DATI}_{\perp}^{\text{DATI}}$ allora l'informatica *può fare tutto*

Calcolare funzioni vuol dire risolvere problemi *in generale*, a ogni problema è possibile associare una funzione soluzione che permette di risolverlo automaticamente.

Un possibile approccio per risolvere l'inclusione è tramite la **cardinalità** (funzione che associa ogni insieme al numero di elementi che contiene) dei due insiemi. Potrebbe però presentare dei problemi: è efficace solo quando si parla di insiemi finiti. Ad esempio, l'insieme dei numeri naturali contiene l'insieme dei numeri pari $\mathbb{P} \subsetneq \mathbb{N}$, ma $|\mathbb{N}| = |\mathbb{P}| = \infty$.
Serve una diversa definizione di cardinalità che considera l'esistenza di infiniti *più densi di altri*.

1.4 Relazioni di Equivalenza

Dati due insiemi A, B, una relazione binaria R è un sottoinsieme $R \subseteq A \times B$ di coppie ordinate. Data $R \subseteq A^2$, due elementi sono in relazione sse $(a, b) \in R$. Indichiamo la relazione tra due elementi anche con la notazione infissa aRb.

Una classe importante di relazioni è quella delle **relazioni di equivalenza**: una relazione $R \subseteq A^2$ è una relazione di equivalenza sse rispetta le proprietà di

- riflessività: $\forall a \in A,\ (a, a) \in R$
- simmetria: $\forall a, b \in A,\ (a, b) \in R \Leftrightarrow (b, a) \in R$
- transitività: $\forall a, b, c \in A,\ (a, b) \in R \land (b, c) \in R \implies (a, c) \in R$

1.4.1 Partizione indotta dalla relazione di equivalenza

A ogni relazione di equivalenza $R \subseteq A^2$ si può associare una **partizione**, ovvero un insieme di sottoinsiemi $A_i \subseteq A$ tali che

- $\forall i \in \mathbb{N}^+,\ A_i \neq \varnothing$
- $\forall i, j \in \mathbb{N}^+$, se $i \neq j$ allora $A_i \cap A_j = \varnothing$
- $\bigcup_{i \in \mathbb{N}^+} A_i = A$

La relazione R definita su A^2 *induce* una partizione $\{A_1, A_2, \dots\}$ su A.

1.4.2 Classi di equivalenza e Insieme quoziente

Dato un elemento $a \in A$, chiamiamo **classe di equivalenza** di a l'insieme

$$[a]_R = \{b \in A | (a, b) \in R\}$$

Ovvero, tutti gli elementi in relazione con a, chiamato **rappresentante** della classe.

Si può dimostrare che

- non esistono classi di equivalenza vuote, per riflessività

- dati $a, b \in A$, allora $[a]_R \cap [b]_R = \varnothing$, oppure $[a]_R = [b]_R$, i due elementi o sono in relazione o non lo sono
- $\bigcup_{a \in A} [a]_R = A$

L'insieme delle classi di equivalenza, per definizione, è una partizione indotta da R su A, detta **insieme quoziente** di A rispetto ad R, denotato con A/R.

1.5 Cardinalità

1.5.1 Isomorfismi

Due insiemi A e B sono **isomorfi** (*equi-numerosi*) se esiste una biezione tra essi, denotato come $A \sim B$. Chiamando $\mathcal{U}$ l'insieme di tutti gli insiemi, la relazione $\sim$ è $\sim \subseteq \mathcal{U}^2$.

Dimostriamo che $\sim$ è una relazione di equivalenza:

- riflessività: $A \sim A$, la biezione è data dalla funzione identità i_A
- simmetria: $A \sim B \Leftrightarrow B \sim A$, la biezione è data dalla funzione inversa
- transitività: $A \sim B \wedge B \sim C \implies A \sim C$, la biezione è data dalla composizione delle funzioni usate per $A \sim B$ e $B \sim C$

Dato che $\sim$ è una relazione di equivalenza, permette di partizionare l'insieme $\mathcal{U}$, risultando in classi di equivalenza contenenti insiemi isomorfi, ovvero con la stessa cardinalità. Possiamo quindi definire la **cardinalità** come l'insieme quoziente di $\mathcal{U}$ rispetto alla relazione $\sim$.

Questo approccio permette il *confronto delle cardinalità di insiemi infiniti*, basta trovare una funzione biettiva tra i due insiemi per poter affermare che sono isomorfi.

1.5.2 Cardinalità finita

La prima classe di cardinalità è quella delle cardinalità finite. Definiamo la seguente famiglia di insiemi:

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

Un insieme A ha **cardinalità finita** sse $A \sim J_n$ per qualche $n \in \mathbb{N}$; in tal caso possiamo scrivere $|A| = n$. La classe di equivalenza $[J_n]_\sim$ identifica tutti gli insiemi di $\mathcal{U}$ contenenti n elementi.

1.5.3 Cardinalità infinita

L'altra classe di cardinalità è quella delle **cardinalità infinite**, ovvero gli insiemi non in relazione con J_n. Si possono dividere in **numerabili** e **non numerabili**.

Insiemi numerabili

Un insieme A è numerabile sse $A \sim \mathbb{N}$, ovvero $A \in [\mathbb{N}]_\sim$. Vengono anche detti **listabili**, in quanto è possibile elencare tutti gli elementi dell'insieme A tramite una funzione f biettiva tra $\mathbb{N}$ e A; grazie ad f possiamo elencare gli elementi di A, formando l'insieme

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

Ed è esaustivo, in quanto elenca tutti gli elementi di A.

Questi insiemi hanno cardinalità $\aleph_0$ (*aleph*).

Insiemi non numerabili

Gli insiemi non numerabili sono insiemi a cardinalità infinita ma non listabili, sono "più fitti" di $\mathbb{N}$; ogni lista generata non può essere esaustiva.

Il più noto tra gli insiemi non numerabili è l'insieme $\mathbb{R}$ dei numeri reali.

Teorema 1.5.1. *L'insieme $\mathbb{R}$ non è numerabile ($\mathbb{R} \nsim \mathbb{N}$)*

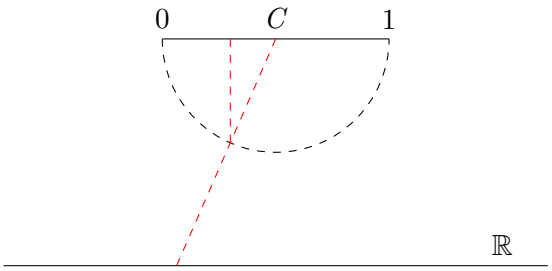
Dimostrazione. Suddividiamo la dimostrazione in 3 punti:

1. dimostriamo che $\mathbb{R} \sim (0,1)$
2. dimostriamo che $\mathbb{N} \not\sim (0,1)$
3. dimostriamo che $\mathbb{R} \not\sim \mathbb{N}$

Per dimostrare che $\mathbb{R} \sim (0,1)$ serve trovare una biezione tra $\mathbb{R}$ e $(0,1)$. Usiamo una rappresentazione grafica:

- disegnare una semicirconferenza di raggio $1/2$, centrata in $1/2$, quindi con diametro 1
- disegnare la perpendicolare al punto da mappare che interseca la circonferenza
- disegnare la semiretta passante per il centro C e l'intersezione precedente

L'intersezione tra asse reale (parallela al diametro) e semiretta finale è il punto mappato.



Questo approccio permette di dire che $\mathbb{R}$ è isomorfo a qualsiasi segmento di lunghezza maggiore di 0. La stessa biezione vale anche sull'intervallo chiuso $[0,1]$ (e di conseguenza qualsiasi intervallo chiuso), usando la "compattificazione" $\mathbb{R} = \mathbb{R} \cup \{\pm\infty\}$ e mappando 0 su $-\infty$ e 1 su $+\infty$.

Continuiamo dimostrando che $\mathbb{N} \prec (0,1)$: serve dimostrare che l'intervallo $(0,1)$ non è listabile, quindi che ogni lista manca di almeno un elemento. Proviamo a "costruire" un elemento che andrà a mancare. Per assurdo, sia $\mathbb{N} \sim (0,1)$, allora possiamo listare gli elementi di $(0,1)$ come

$$\begin{array}{llllll} 0. & a_{00} & a_{01} & a_{02} & \dots \\ 0. & a_{10} & a_{11} & a_{12} & \dots \\ 0. & a_{20} & a_{21} & a_{22} & \dots \\ 0. & & \dots & & \end{array}$$

dove con a_{ij} indichiamo la cifra di posto j dell'i-esimo elemento della lista.

Costruiamo il numero $c = 0.c_0c_1\dots$ tale che

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

Viene costruito "guardando" le cifre sulla diagonale principale, apparterrà sicuramente a $(0,1)$ ma differirà per almeno una posizione (quella sulla diagonale principale) da ogni numero presente all'interno della lista. Questo è assurdo sotto l'assunzione che $(0,1)$ è numerabile, quindi abbiamo provato che $\mathbb{N} \prec (0,1)$.

Il terzo punto $\mathbb{R} \prec \mathbb{N}$ si dimostra per transitività.

Più in generale, non si riesce a listare nessun segmento di lunghezza maggiore di 0.

□

Questa dimostrazione (punto 2 in particolare) è detta **dimostrazione per diagonalizzazione**.

L'insieme $\mathbb{R}$ viene detto **insieme continuo** e tutti gli insiemi isomorfi a $\mathbb{R}$ si dicono continui a loro volta.

Gli insiemi continui hanno cardinalità $\aleph_1$.

Insieme delle Parti

L'**insieme delle parti** di $\mathbb{N}$ (anche detto *power set*), è definito come

$$P(\mathbb{N}) = 2^{\mathbb{N}} = \{S | S \text{ è sottoinsieme di } \mathbb{N}\}$$

Teorema 1.5.2. $P(\mathbb{N}) \not\sim \mathbb{N}$.

Dimostrazione. Possiamo dimostrare questo teorema tramite diagonalizzazione. Il vettore caratteristico di un sottoinsieme è un vettore che nella posizione p_i ha 1 se $i \in A$, 0 altrimenti (tipo vettore di incidenza).

Rappresentiamo $A \subseteq \mathbb{N}$ sfruttando il suo vettore caratteristico

$$\begin{array}{lcccccccc} \mathbb{N}: & 0 & 1 & 2 & 3 & 4 & 5 & 6 & \dots \\ A: & 0 & 1 & 1 & 0 & 1 & 1 & 0 & \dots \end{array}$$

Supponiamo, per assurdo, che $P(\mathbb{N})$ sia numerabile. Vista questa proprietà, possiamo listare tutti i vettori caratteristiche che appartengono a $P(\mathbb{N})$ come

$$\begin{array}{lcccc} b_0 & = & b_{00} & b_{01} & b_{02} & \dots \\ b_1 & = & b_{10} & b_{11} & b_{12} & \dots \\ b_2 & = & b_{20} & b_{21} & b_{22} & \dots \end{array}$$

Vogliamo quindi costruire un vettore che appartiene a $P(\mathbb{N})$ ma non presente nella lista precedente. Definiamo

$$c = \overline{b_{00}}\,\overline{b_{11}}\,\overline{b_{22}}\dots$$

ovvero il vettore che contiene in posizione c_i il complemento di b_{ii}.

Questo vettore appartiene a $P(\mathbb{N})$, in quanto sicuramente sottoinsieme di $\mathbb{N}$, ma non è presente nella lista precedente perché diverso da ogni elemento almeno di una cifra (quella sulla diagonale principale).

Questo è assurdo per l'assunzione che $P(\mathbb{N})$ è numerabile, quindi $P(\mathbb{N}) \not\sim \mathbb{N}$. □

Insieme delle funzioni

L'**insieme delle funzioni** da $\mathbb{N}$ a $\mathbb{N}$ è definito come

$$\mathbb{N}_\perp^\mathbb{N} = \{f : \mathbb{N} \to \mathbb{N}\}$$

Teorema 1.5.3. $\mathbb{N}_\perp^\mathbb{N} \not\prec \mathbb{N}$.

Dimostrazione. Diagonalizzazione strikes again. Assumiamo, per assurdo, che $\mathbb{N}_\perp^\mathbb{N}$ sia numerabile. Possiamo quindi listare $\mathbb{N}_\perp^\mathbb{N}$ come $\{f_0, f_1, f_2, \dots\}$

	0	1	2	3	...	$\mathbb{N}$
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...	...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$	...	...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	...	...
...	...	...	...	...	...	...

Costruiamo una funzione $\varphi : \mathbb{N} \to \mathbb{N}_\perp$ per dimostrare l'assurdo. Un'idea potrebbe essere $\varphi(n) = f_n(n) + 1$, "spostando" la diagonale, ma non tiene in considerazione il caso $f_n(n) = \perp$ in quanto non sapremmo dare un valore a $\varphi(n) = \perp + 1$. Definiamo quindi

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

Questa funzione appartiene a $\mathbb{N}_\perp^\mathbb{N}$, ma non è presente nella lista precedente, infatti $\forall k \in \mathbb{N}$ si ottiene

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

Questo è assurdo sotto l'assunzione che $\mathbb{N}_\perp^\mathbb{N}$ è numerabile, quindi $\mathbb{N}_\perp^\mathbb{N} \not\prec \mathbb{N}$. □

1.6 Potenza Computazionale di un sistema di calcolo

1.6.1 Validità dell'inclusione $F(\mathcal{C}) \subseteq \mathbf{DATI}_{\perp}^{\mathbf{DATI}}$

Dopo aver dato una più robusta definizione di cardinalità, possiamo studiare la natura dell'inclusione

$$F(\mathcal{C}) \subseteq \mathrm{DATI}_{\perp}^{\mathrm{DATI}}$$

Due intuizioni, da dimostrare, sono:

- PROG $\sim \mathbb{N}$: ogni programma può essere identificato con un numero, come la sua codifica in binario
- DATI $\sim \mathbb{N}$: anche ogni dato può essere identificato tramite la sua codifica in binario

Da questo possiamo dire che

$$F(\mathcal{C}) \sim \mathrm{PROG} \sim \mathbb{N} \not\sim \mathbb{N}_{\perp}^{\mathbb{N}} \sim \mathrm{DATI}_{\perp}^{\mathrm{DATI}}$$

Questo dimostra che **esistono funzioni non calcolabili**, ci sono troppe funzioni e troppi pochi programmi.

Dobbiamo dimostrare le due assunzioni PROG $\sim \mathbb{N}$ e DATI $\sim \mathbb{N}$. Si può fare tramite tecniche di aritmetizzazione (o godelizzazione) di strutture, tecniche che rappresentano delle strutture tramite un numero.

1.7 DATI $\sim \mathbb{N}$

Serve trovare una legge che

1. Associ biunivocamente dati a numeri e viceversa
2. Consenta di operare direttamente sui numeri per operare sui corrispondenti dati, ovvero abbia delle primitive che permettano di lavorare sul numero che "riflettano" il risultato sul dato, senza passare dal dato stesso
3. Consenta di dire, senza perdita di generalità, che i programmi lavorano su numeri

1.7.1 Funzione Coppia di Cantor

La **funzione coppia di Cantor** è la funzione

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

E sfrutta le due "sotto-funzioni"

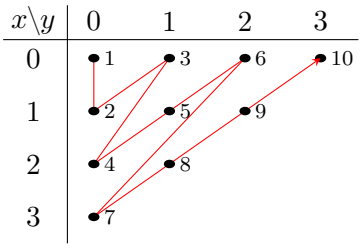
$$\begin{aligned} \sin : &\quad \mathbb{N}^+ \to \mathbb{N} \\ \mathrm{des} : &\quad \mathbb{N}^+ \to \mathbb{N} \end{aligned}$$

Tali che

$$\langle x, y \rangle = n \implies \begin{aligned} \sin(n) &= x \\ \mathrm{des}(n) &= y \end{aligned}$$

Si può rappresentare graficamente come

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



Il valore $\langle x, y \rangle$ rappresenta l'incrocio tra la x-esima riga e la y-esima colonna. Per costruirla:

1. $x = 0$
2. si parte dalla cella $(x, 0)$ e si enumerano le celle della diagonale identificata da $(x, 0)$ e $(0, x)$
3. si incrementa x di 1 e si ripete dal punto precedente

La funzione deve essere:

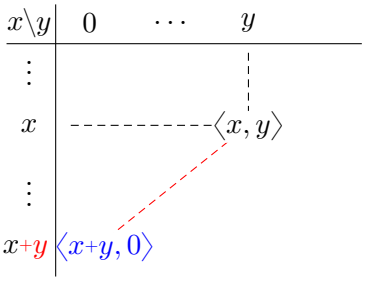
- iniettiva: non ci possono essere celle con lo stesso numero
- suriettiva: ogni numero in $\mathbb{N}^+$ deve comparire

Entrambe le proprietà sono soddisfatte, in quanto la numerazione avviene in maniera incrementale, quindi ogni numero prima o poi compare in una cella e di conseguenza ho una coppia che lo genera.

Forma analitica

Per la definizione di $\langle x, y\rangle$ si può notare che

$$\langle x, y\rangle = \langle x + y, 0\rangle + y$$



Intuitivamente, a partire da $\langle x + y, 0\rangle$ mi basta "salire" seguendo la diagonale fino a $\langle x, y$, ovvero y posti, e per definizione della funzione, y valori più in alto.

Il calcolo della funzione coppia si può quindi ridurre al calcolo di $\langle x + y, 0$. Chiamando $x + y = z$, si può notare come ogni cella

$$\langle z, 0\rangle = z + \langle z - 1, 0\rangle$$

E di conseguenza

$$\begin{aligned}\langle z, 0\rangle &= z + \langle z - 1, 0\rangle \\ &= z + (z - 1) + \langle z - 2, 0\rangle \\ &= z + (z - 1) + \cdots + 1 + \langle 0, 0\rangle = \\ &= \sum_{i=1}^{z} i + 1 = \frac{z(z + 1)}{2} + 1\end{aligned}$$

Mettendo insieme le due proprietà viste possiamo ottenere la formula analitica per la funzione coppia:

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

Forma analitica di sin **e** des

Vogliamo fare la stessa cosa per sin e des, in modo da poter computare l'inversa della funzione coppia, dato n. Grazie alle osservazioni precedenti

sappiamo che

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

Trovando il valore di γ possiamo trovare x e y.

Notiamo come γ sia il più grande valore che, quando calcolato sulla prima colonna ($\langle \gamma, 0 \rangle$) non supera n, ovvero

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

Intuitivamente, si tratta dell'inizio della diagonale che contiene n, è "l'inverso" dell'osservazione fatta in precedenza per la quale $\langle x, y \rangle = \langle x + y, 0 \rangle + y$.

Risolviamo quindi la disequazione

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

Come valore di γ scegliamo

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

E con γ noto possiamo definire le funzioni sin e des come

$$\begin{aligned} \mathrm{des}(n) &= y = n - \langle \gamma, 0 \rangle = n - \tfrac{\gamma(\gamma+1)}{2} - 1 \\ \mathrm{sin}(n) &= x = \gamma - y \end{aligned}$$

Teorema 1.7.1. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+$

Dimostrazione. La funzione di Cantor è una funzione biettiva tra l'insieme $\mathbb{N} \times \mathbb{N}$ e l'insieme $\mathbb{N}^+$, quindi i due insiemi sono isomorfi.

□

Possiamo estendere il risultato all'interno dell'insieme $\mathbb{N}$, ovvero:
Teorema 1.7.2. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$

Dimostrazione. Definiamo la funzione

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

tale che

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

Questa funzione è anch'essa biettiva, quindi i due insiemi sono isomorfi.
□

Grazie a questo è possibile dimostrare anche che $\mathbb{Q} \sim \mathbb{N}$, infatti i numeri razionali si possono rappresentare come coppie (num, den) e, in generale, tutte le tuple sono isomorfe e $\mathbb{N}$, basta iterare in qualche modo la funzione coppia di Cantor.

1.7.2 Applicazione alle strutture dati

I risultati ottenuti fin'ora rendono intuibile come ogni dato possa essere trasformato in un numero, soggetto a trasformazioni matematiche. La dimostrazione *formale* non verrà fatta, anche se verranno fatti esempi di alcune strutture dati che possono essere trasformate in un numero tramite la funzione coppia di Cantor. Ogni struttura dati può essere manipolata e trasformata in una coppia (x, y).

Le **liste** sono le strutture dati più utilizzate nei programmi. In generale non ne è nota la grandezza, di conseguenza è necessario trovare un modo, soprattutto durante l'applicazione di sin e des, per capire quando abbiamo esaurito gli elementi della lista.

Estendiamo la funzione coppia a una lista di interi $x_1, \ldots, x_n$:

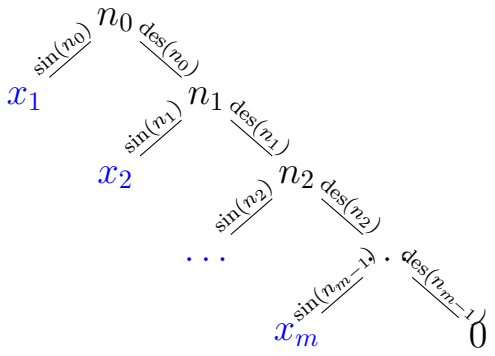
$$\langle x_1, \ldots, x_n \rangle \to \langle x_1, \langle x_2 \langle \ldots \langle x_n, 0 \rangle \ldots \rangle \rangle \rangle$$

Lo 0 rappresenta il fine lista e non è necessario nel caso in cui il numero di elementi è noto.

La decodifica è il processo inverso, partendo dal numero finale si applicano le funzioni sin e des ottenendo a ogni iterazione:

- da des la somma parziale, su cui riapplicare la funzione per ottenere il valore successivo
- da sin il valore presente all'interno della lista

Termina quando il risultato di des è zero, ovvero l'elemento di fine lista che abbiamo inserito (x_n nel caso di array).



Se è presente uno 0 all'interno della lista non è un problema in quanto solo des viene controllato e lo 0 come valore sarà risultato di sin.

Quindi è possibile codificare liste e, di conseguenza, **qualsiasi tipo di dato**, basta convertirlo in una lista di numeri. Per esempio:

- una matrice può essere vista come array di array
- un grafo può essere rappresentato tramite la sua matrice di adiacenza
- i testi sono liste di caratteri
- i suoni si possono campionare per ottenere una lista di valori
- le immagini sono una "lista" di pixel, ognuno dei quali ha un colore come valore

Abbiamo visto come i dati possano essere sostituiti da delle codifiche numeriche; di conseguenza possiamo sostituire tutte le funzioni

$$f : \mathrm{DATI} \to \mathrm{DATI} \quad \text{con funzioni} \quad f' : \mathbb{N} \to \mathbb{N}_\perp$$

In altre parole, l'universo dei problemi per i quali cerchiamo una soluzione automatica è rappresentabile da $\mathbb{N}_{\perp}^{\mathbb{N}}$ e di conseguenza DATI $\sim \mathbb{N}$.

1.8 PROG $\sim \mathbb{N}$

Adesso lavoriamo sulla parte della relazione che afferma

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

Ovvero, la potenza computazionale (l'insieme dei programmi che un sistema di calcolo $\mathcal{C}$ riesce a calcolare, $F(\mathcal{C})$) è isomorfa all'insieme di tutti i programmi, a loro volta isomorfi a $\mathbb{N}$.

Vogliamo arrivare a ricavare un numero dato un programmo e viceversa. Per farlo servirà vedere l'insieme PROG come l'insieme dei programmi scritti in un certo linguaggio di programmazione.

I sistemi analizzati saranno:

- sistema di calcolo RAM
- sistema di calcolo WHILE

Il sistema RAM può apparentemente sembrare "troppo semplice", quindi il sistema WHILE verrà usato per avere un confronto tra le potenze computazionali. Un sistema più sofisticato porta a poter risolvere più problemi?

Ci sono due possibili soluzioni:

- $F(\text{RAM} \neq F(\text{WHILE})$: la computabilità *dipende dal sistema usato*
- $F(\text{RAM}) = F(\text{WHILE})$: la computabilità è *intrinseca nei problemi* e, di conseguenza, tutti i sistemi sono equivalenti (Tesi di Church-Turing)

Il secondo caso è più promettente e, in quel caso, l'obiettivo diventerebbe trovare una *caratterizzazione teorica*, ovvero un "confine" per i problemi calcolabili.

1.8.1 Sistema di calcolo RAM

Il sistema di calcolo RAM è un sistema semplice che permette di definire rigorosamente:

- PROG $\sim \mathbb{N}$
- la **semantica** dei programmi eseguibili, ovvero $\mathcal{C}(P, \_)$, con $\mathcal{C} =$ RAM, ottenendo RAM$(P, _)$
- la **potenza computazionale**, ovvero calcolare $F(\mathcal{C})$ con $\mathcal{C} =$ RAM, ottenendo F(RAM)

Struttura

Una macchina RAM è formata da un processore e da una memoria teoricamente infinita, divisa in **celle/registri** contenenti numeri naturali (dati aritmetizzati).

Indichiamo i **registri** con R_k, con $k \geqslant 0$. Tra questi

- R_0 contiene l'output
- R_1 contiene l'input

Inoltre è presente un registro L, anche detto **program counter** PC che indica l'indirizzo dell'istruzione successiva.

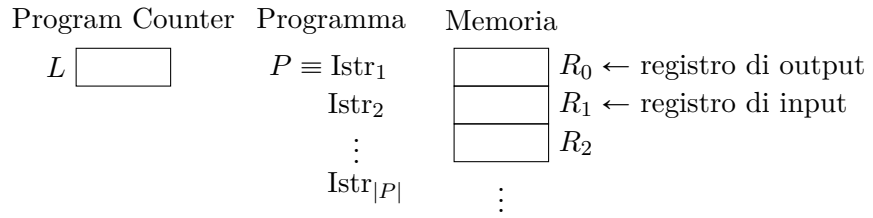
Dato un **programma** P, indichiamo con $|P|$ il numero di istruzioni che il programma contiene.

Le **istruzioni** nel linguaggio RAM sono:

- **incremento**: $R_k \leftarrow R_k + 1$
- **decremento**: $R_k \leftarrow R_k - 1$
- **salto condizionato**: `if` $R_k = 0$ `then goto` m, con $m \in \{1, \ldots, |P|\}$

L'istruzione di decremento è tale che

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



Esecuzione di un programma RAM

L'esecuzione di un programma su una macchina RAM segue i passi:

1. **Inizializzazione**:
   - viene caricato il programma $P \equiv \text{Istr}_1, \ldots \text{Istr}_n$ in memoria
   - il PC viene posto a 1 per indicare di eseguire la prima istruzione del programma
   - viene caricato l'input in R_1
   - ogni altro registro è azzerato
2. **Esecuzione**: le istruzioni vengono eseguite una dopo l'altra, a ogni iterazione passa da L a $L+1$ (escluse operazioni di salto). Essendo il linguaggio RAM *non strutturato* richiede un PC per sapere l'operazione da eseguire al passo successivo.
3. **Terminazione**: per convenzione, si usa $L = 0$ per indicare che l'esecuzione del programma è terminata o andata in loop. Nel caso in cui il programma termini, è detto **segnale di halt** e arresta la macchina
4. **Output**: il contenuto di R_0, in caso di halt, contiene il risultato dell'esecuzione del programma P. Si indica con $\varphi_P(n)$ il contenuto del registro R_0 in caso di halt, oppure $\perp$ in caso di loop

$$\varphi_P(n) = \begin{cases} cont(R_0) & \text{se halt} \\ \perp & \text{se loop} \end{cases}$$

Con $\varphi_P : \mathbb{N} \to \mathbb{N}_\perp$ indichiamo la semantica del programma P.

Con $\mathcal{C}(P, \_)$ indicavamo la semantica di P nel sistema di calcolo $\mathcal{C}$, quindi con $\text{RAM}(P, \_) = \varphi_P$ indichiamo la semantica di P nel sistema di calcolo RAM.

Semantica Operazionale

Per dare una definizione formale della semantica di un programma RAM va specificato il significato di ogni istruzione (**semantica operazionale**), esplicitando l'effetto che quell'istruzione ha sui registri della macchina.

Ogni istruzione fa passare la macchina da uno stato all'altro e la **semantica operazionale** di un'istruzione è la **coppia** formata dagli **stati** della macchina **prima e dopo l'istruzione**.

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

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

Uno stato deve descrivere completamente la situazione della macchina in un certo istante. Il programma rimane uguale, quindi l'informazione da salvare è la situazione globale dei registri R_k e il registro L.

La **computazione** del programma P è una sequenza di stati $\mathcal{S}_i$, ognuno generato dall'esecuzione di un'istruzione del programma; P induce una sequenza di stati $\mathcal{S}_i$, se questa è formata da un numero infinito di stati, allora il programma è andato in loop; in caso contrario, nel registro R_0 si trova il risultato y della computazione di P.

$$\varphi_P : \mathbb{N} \rightarrow \mathbb{N}_\perp \quad \text{t.c.} \quad \varphi_P(n) = \begin{cases} y & \text{se } \exists \mathcal{S}_{fin} \\ \perp & \text{altrimenti} \end{cases}$$

Per definire come passare da uno stato all'altro, definiamo formalmente:

- **Stato**: istantanea di tutte le componenti della macchina, è una funzione
$$\mathcal{S} : \{L, R_i\} \rightarrow \mathbb{N}$$
tale che $\mathcal{S}(R_k)$ restituisce il contenuto del registro R_k quando la macchina si trova nello stato $\mathcal{S}$. Gli stati possibili di una macchina appartengono all'insieme
$$\text{STATI} = \{f : \{L, R_i\} \rightarrow \mathbb{N}\} = \mathbb{N}^{\{L, R_i\}}$$

- **Stato Finale**: uno stato finale $\mathcal{S}_{fin}$ è un qualsiasi stato $\mathcal{S}$ tale che $\mathcal{S}(L) = 0$

- **Dati**: già dimostrato come DATI $\sim \mathbb{N}$
- **Inizializzazione**: serve una funzione che, preso l'input, restituisca lo stato iniziale della macchina:

  $$\text{in} : \mathbb{N} \to \text{STATI} \;\text{ t.c. }\; \text{in}(n) = \mathcal{S}_{init}$$

  Lo stato iniziale $\mathcal{S}_{init}$ è tale che

  $$\mathcal{S}_{init}(R) = \begin{cases} 1 & \text{se } R = L \\ n & \text{se } R = R_1 \\ 0 & \text{altrimenti} \end{cases}$$

- **Programmi**: PROG è definito come l'insieme dei programmi RAM

Manca da definire la *parte dinamica* del programma, ovvero l'esecuzione. Per farlo, definiamo la **funzione di stato prossimo**:

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

tale che

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

dove $\mathcal{S}$ rappresenta lo stato attuale e $\mathcal{S}'$ rappresenta lo stato prossimo dopo l'esecuzione di un'istruzione di P.

La funzione $\delta(\mathcal{S}, P) = \mathcal{S}'$ è tale che

- se $\mathcal{S}(L) = 0$ ho halt, ovvero deve terminare la computazione. Poniamo lo stato come indefinito, ovvero $\mathcal{S}' = \perp$
- Se $\mathcal{S}(L) > |P|$ vuol dire che P non contiene istruzioni che bloccano esplicitamente l'esecuzione del programma. Lo stato $\mathcal{S}'$ è tale che

  $$\mathcal{S}'(R) = \begin{cases} 0 & \text{se } R = L \\ \mathcal{S}(R_i) \text{ se } R = R_i \forall i \end{cases}$$

- Se $1 \leqslant \mathcal{S}(L) \leqslant |P|$ considero l'istruzione $\mathcal{S}(L)$-esima:
  - se ho un incremento/decremento sul registro R_k definisco $\mathcal{S}'$ tale che

    $$\begin{cases} \mathcal{S}'(L) & = \mathcal{S}(L) + 1 \\ \mathcal{S}'(R_k) & = \mathcal{S}(R_k) \pm 1 \\ \mathcal{S}'(R_i) & = \mathcal{S}(R_i) \;\text{ per }\; i \neq k \end{cases}$$

– Se ho un `goto` sul registro R_k che salta all'indirizzo m, definisco $\mathcal{S}'$ tale che

$$\mathcal{S}'(L) = \begin{cases} m & \text{se } \mathcal{S}(R_k) = 0 \\ \mathcal{S}(L) + 1 & \text{altrimenti} \end{cases}$$

$$\mathcal{S}'(R_i) = \mathcal{S}(R_i)\ \forall i$$

L'esecuzione di un programma $P \in \text{PROG}$ su input $n \in \mathbb{N}$ genera una sequenza di stati

$$\mathcal{S}_0, \mathcal{S}_1, \ldots, \mathcal{S}_i, \mathcal{S}_{i+1}, \ldots$$

tali che

$$\begin{array}{cccc} & \mathcal{S}_0 & = & \text{in}(n) \\ \forall i & \mathcal{S}_{i+1} & = & \delta(\mathcal{S}_i, P) \end{array}$$

La sequenza è infinita quando P va in loop, mentre se termina raggiunge uno stato $\mathcal{S}_m$ tale che $\mathcal{S}_m(L) = 0$, ovvero ha ricevuto il segnale di halt.

La semantica di P è

$$\varphi_P(n) = \begin{cases} y & \text{se } P \text{ termina in } \mathcal{S}_m, \text{ con } \mathcal{S}_m(L) = 0 \text{ e } \mathcal{S}_m(R_0) = y \\ \perp & \text{se P va in loop} \end{cases}$$

La potenza computazionale del sistema RAM è

$$F(\text{RAM}) = \left\{ f \in \mathbb{N}_\perp^{\mathbb{N}} | \exists P \in \text{PROG} | \varphi_P = f \right\} = \{\varphi_P | P \in \text{PROG}\} \subsetneq \mathbb{N}_\perp^{\mathbb{N}}$$

L'insieme è formato da tutte le funzioni $f : \mathbb{N} \to \mathbb{N}_\perp$ che hanno un programma che le calcola in un sistema RAM.

1.8.2 Aritmetizzazione di un programma

Per verificare che PROG $\sim \mathbb{N}$ basterebbe trovare una funzione che permetta di codificare i programmi in numeri in modo biunivoco. Data una lista di istruzioni semplici $P \equiv \text{Istr}_1, \ldots, \text{Istr}_m$ se questa fosse codificata come una lista di interi potremmo sfruttare la funzione coppia di Cantor per ottenere un numero associato al programma P.

Quindi vogliamo trovare una funzione Ar che associ a ogni istruzione I_k la sua codifica numerica c_k. Se la funzione trovata è anche biunivoca siamo sicuri di poter trovare anche la sua inversa, ovvero la funzione che ci permette di ricavare I_k da c_k.

Riassumendo, vogliamo trasformare la lista di istruzioni un una lista di numeri su cui successivamente applicare la funzione coppia di Cantor. Vorremmo anche ottenere la lista di istruzioni originale data la codifica.

$$
\begin{array}{ccccccc}
P \equiv & \mathrm{Istr}_1, & \mathrm{Istr}_2, & \ldots, & \mathrm{Istr}_m \\
& \updownarrow & \updownarrow & & \updownarrow \\
& \boxed{Ar} & \boxed{Ar} & \cdots & \boxed{Ar} \\
& \updownarrow & \updownarrow & & \updownarrow \\
& c_1, & c_2, & \ldots, & c_m \\
& \updownarrow & \updownarrow & & \updownarrow \\
& \multicolumn{3}{c}{\boxed{\langle c_1, c_2, \ldots, c_m \rangle}} \\
& & \updownarrow & & \\
cod(P) & = & n & &
\end{array}
$$

L'associazione biunivoca di un numero a una struttura si dice aritmetizzazione o Gödelizzazione.

Applicazione ai programmi RAM

Dovendo codificare tre istruzioni nel linguaggio RAM, definiamo la funzione Ar tale che

$$
Ar(I) = \begin{cases} 3k & \text{se } I \equiv R_k \leftarrow R_k + 1 \\ 3k+1 & \text{se } I \equiv R_k \leftarrow R_k \dot{-} 1 \\ 3k\langle k, m \rangle - 1 & \text{se } I \equiv \texttt{ if } R_k = 0 \texttt{ then goto } m \end{cases}
$$

Per l'inversa, in base al modulo tra n e 3 ottengo una certa istruzione:

$$
Ar^{-1}(n) = \begin{cases} R_{\frac{n}{3}} \leftarrow R_{\frac{n}{3}} + 1 & \text{se } n \mod 3 = 0 \\ R_{\frac{n-1}{3}} \leftarrow R_{\frac{n-1}{3}} \dot{-} 1 & \text{se } n \mod 3 = 1 \\ \texttt{if } R_{\sin\left(\frac{n+1}{3}\right)} = 0 \texttt{ then goto } \mathrm{des}\left(\frac{n+1}{3}\right) & \text{se } n \mod 3 = 2 \end{cases}
$$

Per tornare indietro devo prima invertire la funzione coppia di Cantor e poi invertire la funzione Ar.

La lunghezza del programma P, indicata con $|P|$, si calcola come $len(cod(P))$.

Abbiamo quindi dimostrato che PROG $\sim \mathbb{N}$.

Osservazioni

Avendo $n = cod(P)$ si può scrivere

$$\varphi_P(t) = \varphi_n(t)$$

Ovvero, la semantica di P è uguale alla semantica della sua codifica.

I numeri diventano un *linguaggio di programmazione*.

Si può scrivere l'insieme

$$F(\text{RAM}) = \{\varphi_P : P \in \text{PROG}\}$$

come

$$F(\text{RAM}) = \{\varphi_i\}_{i \in \mathbb{N}}$$

L'insieme, grazie alla dimostrazione di PROG $\sim \mathbb{N}$, è numerabile.

Abbiamo dimostrato rigorosamente che

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

Di conseguenza, anche nel sistema di calcolo RAM esistono funzioni no calcolabili.

La RAM è troppo elementare affinché $F(\text{RAM})$ rappresenti formalmente la "classe dei problemi risolubili automaticamente", quindi considerando un sistema di calcolo $\mathcal{C}$ più sofisticato, ma comunque trattabile rigorosamente come il sistema RAM, potremmo dare un'idea formale di "ciò che è calcolabile automaticamente".

Se riesco a dimostrare che $F(\text{RAM}) = F(\mathcal{C})$ allora cambiare la tecnologia non cambia ciò che è calcolabile, ovvero la calcolabilità è intrinseca ai problemi, quindi la si può caratterizzare matematicamente.

1.8.3 Sistema di calcolo WHILE

Introduciamo quindi il sistema di calcolo WHILE per vedere se riusciamo a "catturare" più o meno funzioni calcolabili dalla macchina RAM.

Struttura

La macchina WHILE ha anch'essa, come la macchina RAM, una serie di registri, ma al posto di essere *potenzialmente infiniti* sono esattamente 21. Il registro R_0 è il **registro di output**, mentre R_1 è il **registro di input**. Non esiste il Program Counter in quanto il linguaggio è **strutturato** e ogni istruzione in questo linguaggio va eseguita in ordine.

Il linguaggio WHILE prevede una **definizione induttiva**: vengono definiti alcuni comandi base e i comandi più complessi sono una concatenazione dei comandi base.

Assegnamento: Comando di base, ne esistono di tre tipi:

$$x_k := 0,$$
$$x_k := x_j + 1$$
$$x_k := x_j \dot{-} 1$$

Queste istruzioni sono più complete rispetto alle istruzioni RAM, in una sola istruzione possiamo azzerare il valore di una variabile o assegnare a una variabile il valore di un'altra aumentato/diminuito di 1.

While: Primo comando "induttivo". Si tratta di un comando della forma:

$$\texttt{while } x_k \neq 0 \texttt{ do } C$$

Dove C è detto **corpo** e può essere un assegnamento, un comando while o un comando composto.

Composto: Altro comando induttivo. Si tratta di un comando nella forma

$$\texttt{begin } C_1; \ldots; C_n \texttt{ end}$$

Dove i vari C_i sono, come prima, assegnamenti, comandi while o comandi composti.

Programma WHILE: Un programma WHILE è un comando composto, e l'insieme di tutti i programmi WHILE è l'insieme

$$W - \text{PROG} = \{\text{PROG scritti in linguaggio WHILE}\}$$

Chiamiamo

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

la **semantica** del programma $W \in W - \text{PROG}$.

Per dimostrare una proprietà P di un programma $W \in W - \text{PROG}$, data la definizione induttiva del linguaggio WHILE, è naturale procedere induttivamente:

1. dimostro P vera sugli assegnamenti
2. suppongo P vera sul comando C e la dimostro vera per `while` $x_k \neq 0$ `do` C
3. suppongo P vera sui comandi $C_1, \ldots, C_n$ e la dimostro vera per `begin` $C_1; \ldots; C_n$ `end`

Esecuzione di un programma WHILE

L'esecuzione di un programma WHILE W è composta dalle seguenti fasi:

1. **Inizializzazione**: ogni registro x_i viene posto a 0, tranne x_1, che contiene l'input n
2. **Esecuzione**: essendo WHILE un linguaggio con strutture di controllo, non serve un Program Counter, poiché le istruzioni di W vengono eseguite l'una dopo l'altra
3. **Terminazione**: l'esecuzione di W può
   - *arrestarsi*: se arriva al termine delle istruzioni
   - *non arrestarsi*: se entra in un loop
4. **Output**: Se il programma va in halt, l'output è contenuto nel registro x_0. Possiamo scrivere

$$\Psi_W(n) = \begin{cases} cont(x_0) & \text{se halt} \\ \perp & \text{se loop} \end{cases}$$

Definizione formale per l'esecuzione

Come per i programmi RAM, serve una definizione formale per la semantica di un programma WHILE, per la quale servono una serie di elementi

- **Stato**: una tupla grande quanto il numero di variabili, dove quindi $\underline{x} = (c_0, \ldots, c_{20})$ rappresenta uno stato, con c_i rappresentante il contenuto della variabile i
- W**-STATI**: Insieme di tutti gli stati possibili, contenuto in $\mathbb{N}^{21}$ vista la definizione degli stati
- **Dati**: già visto che DATI $\sim \mathbb{N}$
- **Inizializzazione**: Lo stato iniziale è descritto dalla funzione

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

- **Semantica operazionale**: Vogliamo trovare una funzione che, presi comando da eseguire e stato corrente, restituisce lo stato successivo

Funzione stato prossimo: Soffermandoci sull'ultimo punto, vogliamo trovare la funzione

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

Che, dati un comando C del linguaggio WHILE e lo stato corrente $\underline{x}$, calcoli

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

con $\underline{y}$ stato prossimo. Quest'ultimo dipende dal comando C, ma essendo C induttivo, possiamo provare a dare una definizione induttiva della funzione.

Partendo dal passo base, gli **assegnamenti**:

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

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

Proseguiamo con il **passo induttivo**:

- **Comando composto**: vogliamo calcolare

$$[\![\texttt{begin } C_1; \dots ; C_n \texttt{ end}]\!](\underline{x})$$

Conoscendo ogni $[\![C_i]\!]$ per ipotesi induttiva, calcoliamo la funzione

$$[\![C_n]\!]\left(\dots\left([\![C_2]\!]\left([\![C_1]\!](\underline{x})\right)\right)\dots\right) = \left([\![C_n]\!] \circ \dots \circ [\![C_1]\!]\right)(\underline{x})$$

Ovvero, applichiamo in ordine i comandi C_i presenti nel comando composto C

- **Comando while**: vogliamo calcolare

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

Conoscendo ogni $[\![C_i]\!]$ per ipotesi induttiva, calcoliamo la funzione

$$[\![C]\!]\left(\dots\left([\![C]\!](\underline{x})\right)\dots\right)$$

Bisogna capire quante volte eseguire il ciclo: dato $[\![C]\!]^e$ (comando C eseguito e volte) vorremmo trovare il valore di e. Questo è il numero minimo di iterazioni che portano in uno stato in cui $x_k = 0$, ovvero il comando `while` diventa

$$\texttt{while } x_k \neq 0 \texttt{ do } C = \begin{cases} [\![C]\!]^e(\underline{x}) & \text{se } e = \mu_t \\ \bot & \text{altrimenti} \end{cases}$$

Il valore $e = \mu_t$ è quel numero tale che $[\![C]\!]^e(\underline{x})$ ha la k-esima componente dello stato uguale a 0

Definita la semantica operazionale, manca solo da definire cos'è la **semantica del programma** W su input n. Questa è la funzione

$$\Psi_W : \mathbb{N} \to \mathbb{N}_\bot \mid \Psi_W(n) = \text{Proj}\,(0, [\![W]\!](w\text{-}in(n)))$$

Questo è valido in quanto W programma WHILE è un programma composto, e abbiamo definito come deve comportarsi la funzione $[\![]\!]()$ sui comandi composti.

Potenza Computazionale: La potenza computazionale del sistema di calcolo WHILE è l'insieme

$$F(\text{WHILE}) = \left\{ f \in \mathbb{N}_\bot^{\mathbb{N}} | \exists W \in W\text{-PROG} | f = \Psi_W \right\} = \{\Psi_W : W \in W\text{-PROG}\}$$

Ovvero, l'insieme formato da tutte le funzioni che possono essere calcolate con un programma in W-PROG.

1.8.4 Confronto tra macchina RAM e WHILE

Viene naturale confrontare i due sistemi presentati per capire il "più potente", sempre che ce ne sia uno. Le possibilità sono:

- $F(\text{RAM}) \subsetneq F(\text{WHILE})$, data l'estrema semplicità del sistema RAM
- $F(\text{RAM}) \cap F(\text{WHILE}) = \varnothing$, avere insiemi disgiunti significherebbe che il concetto di calcolabile dipende dalla macchina considerata
- $F(\text{WHILE}) \subseteq F(\text{RAM})$, che sarebbe sorprendente vista l'apparente maggiore sofisticatezza del sistema WHILE
- $F(\text{WHILE}) = F(\text{RAM})$, ovvero il concetto di calcolabile non dipende dalla tecnologia utilizzata ma è intrinseco nei problemi

Confronto tra sistemi di calcolo: Ponendo di avere $\mathcal{C}_1$ e $\mathcal{C}_2$ sistemi di calcolo con programmi in $\mathcal{C}_1$-PROG e $\mathcal{C}_2$-PROG e le relative potenze computazionali

$$F(\mathcal{C}_1) = \{f : \mathbb{N} \to \mathbb{N}_\perp | \exists P_1 \in \mathcal{C}_1\text{-PROG} | f = \Psi_{P_1}\} = \{\Psi_{P_1} : P_1 \in \mathcal{C}_1\text{-PROG}\}$$

$$F(\mathcal{C}_2) = \{f : \mathbb{N} \to \mathbb{N}_\perp | \exists P_2 \in \mathcal{C}_2\text{-PROG} | f = \Psi_{P_2}\} = \{\Psi_{P_2} : P_2 \in \mathcal{C}_2\text{-PROG}\}$$

Mostrare che il primo sistema non è più potente del secondo $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$ vuol dire dimostrare che ogni elemento nel primo insieme deve stare anche nel secondo

$$\forall f \in F(\mathcal{C}_1) \implies f \in F(C_2)$$

Espandendo la definizione di $f \in F(\mathcal{C})$ la relazione diventa

$$\forall P_1 \in \mathcal{C}_1\text{-PROG} | f = \Psi_{P_1} \implies \exists P_2 \in \mathcal{C}_2\text{-PROG} | f = \Psi_{P_2}$$

Per ogni programma calcolabile nel primo sistema di calcolo ne esiste uno con la stessa semantica nel secondo sistema. Vogliamo trovare un **compilatore** (o **traduttore**), ovvero una funzione che trasformi un programma del primo sistema in uno del secondo.

Traduzioni

Dati $\mathcal{C}_1$ e $\mathcal{C}_2$ due sistemi di calcolo, definiamo **traduzione** da $\mathcal{C}_1$ a $\mathcal{C}_2$ una funzione

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

Con le seguenti proprietà:

- **programmabile**: esiste un modo per programmarla
- **completa**: deve saper tradurre *ogni* programma in $\mathcal{C}_1$-PROG in un programma in $\mathcal{C}_2$-PROG
- **corretta**: mantiene la semantica dei programmi di partenza, ovvero

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

  dove Ψ rappresenta la semantica dei programmi in $\mathcal{C}_1$-PROG e φ rappresenta la semantica dei programmi in $\mathcal{C}_2$-PROG

Teorema 1.8.1. *Se esiste T : $\mathcal{C}_1$-PROG $\to$ $\mathcal{C}_2$-PROG allora $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$*

Dimostrazione. Se $f \in F(\mathcal{C}_1)$ allora esiste un programma $P_1 \in \mathcal{C}_1$-PROG tale che $\Psi_{P_1} = f$.

A questo programma P_1 applico T, ottenendo $T(P_1) = P_2 \in \mathcal{C}_2$-PROG (per *completezza*) tale che $\varphi_{P_2} = \Psi_{P_1} = f$ (per *correttezza*).

Ho trovato un programma $P_2 \in \mathcal{C}_2$-PROG la cui semantica è f, allora $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$.

□

Mostreremo che $F(\text{WHILE}) \subseteq F(\text{RAM})$, ovvero che il sistema WHILE non è più potente del sistema RAM. Costruiremo un compilatore

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

che rispetti le caratteristiche di programmabilità, completezza e correttezza.

1.8.5 $F(\textbf{WHILE}) \subseteq F(\textbf{RAM})$

Per provare che $F(\text{WHILE}) \subseteq F(\text{RAM})$ vogliamo costruire un compilatore da WHILE a RAM.

Per comodità, introduciamo un linguaggio RAM *etichettato*: aggiunge la possibilità di etichettare un'istruzione che indica un punto di salto o di arrivo (stile label assembly). Non altera la potenza espressiva del linguaggio in quanto si tratta di un'aggiunta puramente sintattica: il RAM etichettato si traduce facilmente in RAM puro.

Essendo W-PROG un insieme definito induttivamente, possiamo definire induttivamente anche il compilatore:

- **Passo base**: come compilare gli assegnamenti
- **Passo induttivo**:
  1. Per I.H., assumo di sapere $\text{Comp}(C_1), \ldots, \text{Comp}(C_m)$ e mostro come compilare il comando composto `begin` $C_1; \ldots; C : n$ `end`
  2. Per I.H., assumo di sapere $\text{Comp}(C)$ e mostro come compilare il comando `while` $x_k \neq 0$ `do` C

Nelle traduzioni andremo a mappare la variabile WHILE x_k nel registro RAM R_k. Questo non crea problemi in quanto stiamo mappando un numero finito di registri in un insieme infinito.

Il primo assegnamento che mappiamo è $x_k := 0$

$$\text{Comp}(x_k := 0) = \boxed{\begin{array}{ll} \texttt{LOOP:} & \texttt{IF } R_k = 0 \texttt{ THEN GOTO EXIT} \\ & R_k \leftarrow R_k \dot{-} 1 \\ & \texttt{IF } R_{21} = 0 \texttt{ THEN GOTO LOOP} \\ \texttt{EXIT:} & R_k \leftarrow R_k \dot{-} 1 \end{array}}$$

Questo programma RAM azzera il valore di R_k usando il registro R_{21} per saltare al check della condizione iniziale. Viene utilizzato il registro R_{21} perché, non essendo mappato a nessuna variabile WHILE, sarà sempre nullo dopo la fase di inizializzazione.

Gli altri due assegnamenti da mappare sono $x_k := x_j + 1$ e $x_k := x_j \dot{-} 1$:

- Se $k = j$, la traduzione è banale e l'istruzione RAM è

$$\text{Comp}(x_k := x_k \pm 1) = R_k \leftarrow R_k \pm 1$$

- Invece se $k \neq j$ la prima idea è quella di "spostare" x_j in x_k e poi fare ± 1 ma non funziona per due ragioni

  1. se $R_k \neq 0$ la migrazione (quindi sommare R_j a R_k) non genera R_j dentro R_k. Si può risolvere azzerando il registro R_k prima della migrazione
  2. R_j dopo il trasferimento è ridotto a 0, ma questo non è il senso dell'istruzione, vorrei solo "*fotocopiarlo*" dentro R_k. Questo può essere risolto salvando R_j in un altro registro, azzerare R_k, spostare R_j e ripristinare il valore originale di R_j

  Quindi possiamo mappare come:

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

```
LOOP:  IF R_j = 0 THEN GOTO E1
       R_j ← R_j ∸ 1
       R_22 ← R_22 + 1
       IF R_21 = 0 THEN GOTO LOOP
  E1:  IF R_k = 0 THEN GOTO E2
       R_k ← R_k ∸ 1
       IF R_21 = 0 THEN GOTO EXIT 1
  E2   IF R_22 = 0 THEN GOTO E3
       R_k ← R_k + 1
       R_j ← R_j + 1
       R_22 ← R_22 ∸ 1
       IF R_21 = 0 THEN GOTO E2
  E3:  R_k ← R_k ± 1
```

Salva R_j in R_{22}

Azzera R_k

Rigenera R_j e R_k da R_{22}

Per I.H. sappiamo come compilare $C_1, \dots C_m$. Possiamo calcolare la compilazione del comando composto come

$$\text{Comp}(\texttt{begin } C_1;\ldots;C_n \texttt{ end}) = \boxed{\begin{array}{l}\text{Comp}(C_1)\\ \ldots\\ \text{Comp}(C_m)\end{array}}$$

Per I.H. sappiamo come compilare C. Possiamo calcolare la compilazione del comando `while` come

$$\text{Comp}(\texttt{while } x_k \neq 0 \texttt{ do } C) = \boxed{\begin{array}{ll}\texttt{LOOP:} & \texttt{IF } R_k = 0 \texttt{ THEN GOTO EXIT}\\ & \text{Comp}(C)\\ & \texttt{IF } R_{21} = 0 \texttt{ THEN GOTO LOOP}\\ \texttt{EXIT:} & R_k \leftarrow R_k \dot{-} 1\end{array}}$$

La funzione

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

appena costruita soddisfa le tre proprietà desiderate in quanto:

- Programmabile
- Compila sempre
- Mantiene la semantica

Di conseguenza

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

1.8.6 $F(\mathbf{RAM}) \subseteq F(\mathbf{WHILE})$

Abbiamo mostrato che una macchina WHILE non è più potente di una macchina RAM, ora vogliamo fare l'inverso. Per farlo si userà il concetto di interprete.

Interprete in WHILE per RAM

Introduciamo il concetto di **interprete**. Chiamiamo I_W l'interprete scritto in linguaggio WHILE per programmi scritti in linguaggio RAM.

I_W prende in input un programma $P \in \text{PROG}$ e un dato $x \in \mathbb{N}$ e restituisce "*l'esecuzione*" di P sull'input x. Più formalmente, restituisce la semantica di P su x, quindi $\varphi_P(x)$.

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

Notiamo come l'interprete non crei dei prodotti intermedi, ma si limita a eseguire P sull'input x.

Due problemi principali:

1. Il primo riguarda il tipo di input della macchina WHILE in quanto questa non sa leggere il programma P (listato di istruzioni RAM), sa leggere solo numeri. Dobbiamo modificare I_W in modo che non passi più P ma la sua codifica $cod(P) = n \in \mathbb{N}$. Questo restituisce la semantica del programma codificato con n, che è P, quindi $\varphi_n(x))\varphi_P(x)$
2. Il secondo problema riguarda la quantità di dati in input alla macchina WHILE: questa legge l'input da un singolo registro, mentre qui ne stiamo passando due. Bisogna modificare I_W, condensando l'input tramite la funzione coppia di Cantor, che diventa $\langle x, n \rangle$

Quindi

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

La semantica di I_W diventa

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

Similmente a prima, useremo per comodità il linguaggio **macro-WHILE**, che include alcune macro comode nella scrittura di I_W. Viene modificata solo la sintassi e di conseguenza non la potenza del linguaggio.

Le **macro** utilizzate sono:

- $x_k := x_j + x_s$
- $x_k := \langle x_j, x_s \rangle$
- $x_k := \langle x_1, \ldots, x_n \rangle$
- $x_k :=$ Proj(x_j, x_s), proiezione, estrae l'elemento x_j-esimo dalla lista codificata in x_s
- $x_k :=$ incr(x_j, x_s), incremento, codifica la lista x_s con l'elemento in posizione x_j-esima aumentato di 1

- $x_k := \text{decr}(x_j, x_s)$, decremento, codifica la lista x_s con l'elemento in posizione x_j-esima diminuito di 1
- $x_k := \sin(x_j)$
- $x_k := \text{des}(x_j)$
- Costrutto `If …then …else`

Stato della macchina RAM nell'interprete

Risolto il problema dell'input di un interprete scritto in linguaggio WHILE per i programmi RAM, ora vogliamo scrivere questo interprete. In sintesi, l'interprete esegue una dopo l'altra le istruzioni RAM del programma P e restituisce il risultato $\varphi_P(x)$ (da notare che restituisce il risultato, non un eseguibile).

L'interprete ricostruisce virtualmente tutto ciò che gli serve per gestire il programma. Nel caso di I_W deve ricostruire l'ambiente di una macchina RAM. Quello che faremo sarà ricreare il programma P, il Program Counter L e i registri $R_0, R_1, \ldots,$ dentro le variabili messe a disposizione dalla macchina WHILE.

Primo problema: i programmi RAM possono utilizzare infiniti registri, mentre i programmi WHILE ne hanno solo 21. *Ma P usa veramente un numero infinito di registri?*

In realtà no; se $cod(P) = n$ allora P non userà mai registri R_j con $j > n$. Il programma P userà sempre un numero finito di registri il cui contenuto può essere racchiuso in una lista $a_0, \ldots, a_n$. La soluzione consiste nel raggruppare tutti i valori dei registri tramite Cantor $\langle a_0, \ldots, a_n \rangle$ e salvarne la codifica in un unica variabile. Useremo due registri in più per comodità (possono essere utili, you never know).

L'interprete I_W salva lo stato della macchina RAM nel seguente modo (con $I_W(\langle x, n \rangle) = \varphi_n(x)$)

- $x_0 \leftarrow \langle R_0, \ldots, R_{n+2} \rangle$: stato della memoria della macchina RAM
- $x_1 \leftarrow L$: Program Counter
- $x_2 \leftarrow x$: dato su cui lavora P

- $x_3 \leftarrow n$: "listato" del programma P
- x_4: codice dell'istruzione da eseguire, prelevata da x_3 grazie a x_1

Implementazione

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

Avendo l'interprete I_W si può costruire un compilatore

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

tale che

$$\text{Comp}(P \in \text{PROG}) \equiv \boxed{\begin{array}{l} x_2 := cod(P) \\ x_1 := \langle x_1, x_2 \rangle \\ I_W \end{array}}$$

Ovvero, il compilatore non fa altro che cablare all'input x il programma RAM da interpretare e procede con l'esecuzione dell'interprete. Possiamo verificare le proprietà di un compilatore:

- **programmabile**: lo abbiamo appena fatto
- **completo**: l'interprete riconosce ogni istruzione RAM e riesce a codificarla
- **corretto**: vale la relazione $P \in \text{PROG} \implies \text{Comp}(P) \in W\text{-PROG}$, quindi
  $$\Psi_{\text{Comp}(P)}(x) = \Psi_{I_W}(\langle x, n \rangle) = \varphi_n(x) = \varphi_P(x)$$
  rappresenta la sua semantica

Abbiamo dimostrato che

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

Ovvero l'inclusione opposta al risultato precedente.

Conseguenze

Visti i risultati ottenuti possiamo definire un teorema importante.

Teorema 1.8.2 (Teorema di Böhm-Jacopini)**.** *Per ogni programma con `GOTO` RAM ne esiste uno equivalente in un linguaggio strutturato (WHILE).*

Permette di legare la programmazione a basso livello con quella ad alto livello. In breve, il `GOTO` può essere eliminato e la programmazione a basso livello può essere sostituita con quella ad alto livello.

Inoltre abbiamo di conseguenza dimostrato che

$$\begin{gathered} F(\text{WHILE}) \subseteq F(\text{RAM}) \\ F(\text{RAM}) \subseteq F(\text{WHILE}) \\ \Downarrow \\ F(\text{RAM}) = F(\text{WHILE}) \\ \Downarrow \\ F(\text{RAM}) = F(\text{WHILE}) \sim \text{PROG} \sim \mathbb{N} \nsim \mathbb{N}_{\perp}^{\mathbb{N}} \end{gathered}$$

Quindi, abbiamo dimostrato che i sistemi RAM e WHILE sono in grado di **calcolare le stesse cose**, oltre al fatto che **esistono funzioni non calcolabili**, ovvero la parte destra della catena.

1.8.7 Interprete Universale

Usando il compilatore da WHILE a RAM

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

sul programma I_W, otteniamo

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

E la sua semantica è

$$\varphi_{\mathcal{U}}(\langle x, n \rangle) = \Psi_{I_W}(\langle x, n \rangle) = \varphi_n$$

dove n è la codifica del programma RAM e x il dato di input.

Abbiamo mostrato che esiste un programma RAM in grado di **simulare** tutti gli altri programmi RAM. Questo programma viene detto **interprete universale**.

Un linguaggio verrà considerato "*buono*" se ammette un interprete universale.

1.8.8 Concetto di Calcolabilità

Abbiamo visto come due sistemi di calcolo diversi abbiano la stessa potenza computazionale numerabile. Ma abbiamo visto anche che esistono funzioni non calcolabili. Il nostro obiettivo è quindi definire la regione delle funzioni calcolabili.

Questa regione può essere definita a prescindere dalle macchine usate per calcolare? Bisognerà definire il concetto di "*calcolabile*" in termini astratti, "lontani" dall'informatica, usando la matematica.

1.9 Chiusura

1.9.1 Operazioni

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

$$\text{op} : \underbrace{U \times \cdot \times U}_{k} \to U$$

Il numero k indica l'arietà (o arità) dell'operazione, ovvero la dimensione del dominio dell'operazione.

1.9.2 Proprietà di Chiusura

L'insieme $A \subseteq U$ si dice **chiuso** rispetto all'operazione $\text{op} : U^k \to U$ se e solo se

$$\forall a_1, \ldots, a_k \in A \quad \text{op}(a_1, \ldots, a_k) \in A$$

Ovvero, l'operazione applicata in A restituisce valori $\in A$. In generale, se Ω è un insieme di operazioni su U, allora $A \subseteq U$ è chiuso rispetto a Ω se e solo se A è chiuso per ogni operazione in Ω.

1.9.3 Chiusura di un insieme

Siano $A \subseteq U$ un insieme e $\text{op} : U^k \to U$ un'operazione su di esso. Vogliamo espandere l'insieme A per trovare il più piccolo sottoinsieme di U tale che

- contiene A

- chiuso rispetto a op

Vogliamo espandere A il meno possibile per garantire la chiusura rispetto a op.

Ci sono due risposte banali:

- Se A è già chiuso rispetto a op, allora A stesso è l'insieme cercato
- Sicuramente U soddisfa le due richieste, ma non è detto sia il più piccolo

Teorema 1.9.1. *Siano $A \subseteq U$ insiemi e* op *un'operazione su di essi. Il più piccolo sottoinsieme di U contenente A e chiuso rispetto a* op *si ottiene calcolando la chiusura di A rispetto a* op*.*

Tale chiusura A^{op} si definisce induttivamente come:

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

Una definizione più *operativa* di A^{op} può essere:

1. inserisco in A^{op} tutti gli elementi di A
2. applico op a una k-upla di elementi in A^{op}
3. se il risultato non è in A^{op} lo aggiungo
4. Ripetere i punti 2 e 3 finché A^{op} cresce

Siano $\Omega = \{\text{op}_1, \ldots, \text{op}_t\}$ un insieme di operazioni su U di arietà rispettivamente $k_1, \ldots, k_t$ e $A \subseteq U$ insieme. Definiamo **chiusura di A rispetto a Ω** il più piccolo sottoinsieme di U contenente A chiuso rispetto a Ω, cioè l'insieme A^{Ω} definito come

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

1.10 Calcolabilità

Per la definizione teorica di calcolabilità introdurremo gli insiemi:

- **ELEM**: insieme di tre funzioni che *qualunque* idea di calcolabile proponibile deve considerare calcolabili. Non può esaurire il concetto di calcolabilità, quindi verrà espanso con altre funzioni
- Ω: insieme di operazioni su funzioni che *costruiscono nuove funzioni*. Le operazioni in Ω sono banalmente implementabili e, applicate a funzioni calcolabili, generano nuove funzioni calcolabili
- $\mathbf{ELEM}^{\Omega} = \mathscr{P}$: la classe delle funzioni ricorsive parziali. L'idea astratta della classe delle funzioni calcolabili secondo Kleene

In seguito alla definizione di $\mathscr{P}$ ci sarà da domandarci se questa idea di calcolabile coincida con le funzioni presenti in $F(\text{RAM}) = F(\text{WHILE})$.

1.10.1 ELEM

Definiamo ELEM con le seguenti funzioni

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

Queste sono funzioni di partenza che qualsiasi idea teorica di calcolabilità non può non considerare come calcolabile. Ovviamente non rappresenta tutto ciò che è calcolabile, va quindi ampliato.

1.10.2 Ω

Definiamo ora un insieme Ω di operazioni che permettano di ampliare le funzioni di ELEM fino a coprire tutte le funzioni calcolabili.

Composizione

Il primo operatore è quello di **composizione**. Siano

- $h : \mathbb{N}^k \to \mathbb{N}$ funzione di composizione
- $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$ "funzioni intermedie"

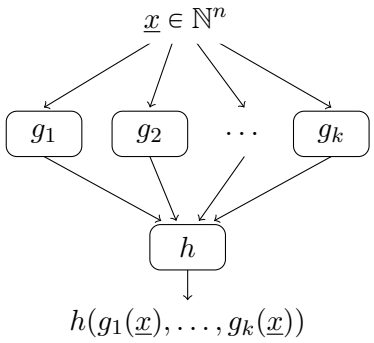
- $\underline{x} \in \mathbb{N}^n$ input

Allora definiamo

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

La funzione tale che

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



La funzione COMP è *intuitivamente* calcolabile partendo da funzioni calcolabili: prima eseguo le singole funzioni $g_1, \ldots, g_k$, poi applico la funzione h sui risultati.

Calcoliamo ora la **chiusura** di ELEM rispetto a COMP, ovvero l'insieme $\text{ELEM}^{\text{COMP}}$.

La funzione $f(x) = x + 2$ appartiene a questo insieme perché:

$$\text{COMP}(s, s)(x) = s(s(x)) = (x + 1) + 1 = x + 2$$

Di conseguenza tutte le funzioni lineari del tipo $f(x) = x + k$ con k prefissato apparterranno all'insieme.

Ma la funzione somma, senza valori prefissati, non appartiene all'insieme, quindi doppiamo ampliare $\text{ELEM}^{\text{COMP}}$ con altre operazioni.

Ricorsione Primitiva

Definiamo un'operazione che permetta di **iterare** sull'operatore di composizione, la **ricorsione primitiva**, usata per definire funzioni ricorsive.

Siano

- $g : \mathbb{N}^n \to \mathbb{N}$ funzione **caso base**
- $h : \mathbb{N}^{n+2} \to \mathbb{N}$ funzione **passo ricorsivo**
- $\underline{x} \in \mathbb{N}^n$ **input**

Definiamo

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

Funzione che generalizza la definizione ricorsiva di funzioni.

Chiusura di $\mathrm{ELEM}^{\mathrm{COMP}}$ rispetto a RP vuol dire calcolare l'insieme $\mathrm{ELEM}^{\{\mathrm{COMP,RP}\}}$. Chiamiamo

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

l'insieme ottenuto dalla chiusura, cioè l'insieme delle funzioni ricorsive primitive.

In questo insieme abbiamo la somma, infatti:

$$\mathrm{somma}(x,y) = \begin{cases} x = \mathrm{Pro}_1^2(x,y) & \text{se } y = 0 \\ s(\mathrm{somma}(x,y-1)) & \text{se } y > 0 \end{cases}$$

Altre funzioni in RICPRIM:

$$\mathrm{prodotto}(x,y) = \begin{cases} 0 = 0^2(x,y) & \text{se } y = 0 \\ \mathrm{somma}(x,\mathrm{prodotto}(x,y-1)) & \text{se } y > 0 \end{cases}$$

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

RICPRIM vs WHILE

L'insieme RICPRIM contiene *molte funzioni*, ma dobbiamo capire se ha raggiunto l'insieme F(WHILE).

RICPRIM è definito come:

- $\forall f \in \text{ELEM} \implies f \in \text{RICPRIM}$
- se $h, g_1, \ldots, g_k \in \text{RICPRIM} \implies \text{COMP}(h, g_1, \ldots, g_k) \in \text{RICPRIM}$
- se $g, h, \in \text{RICPRIM} \implies \text{RP}(g, h) \in \text{RICPRIM}$
- nient'altro sta in RICPRIM

Teorema 1.10.1. *RICPRIM* $\subseteq F($*WHILE*$)$

Dimostrazione. **Passo Base:** Le funzioni di ELEM sono ovviamente while-programmabili, mostrato in precedenza.

Passo Induttivo: Per COMP, assumiamo per ipotesi induttiva che $h, g_1, \ldots, g_k \in$ RICPRIM siano while-programmabili, allora esistono $H, G_1, \ldots, G_K \in W$-PROG tali che $\Psi_H = h$, $\Psi_{G_1} = g_1, \ldots, \Psi_{G_k} = g_k$.
Un programma WHILE che calcola COMP è

```
// Inizialmente in x_1 ho x
input(x)
begin
    x_0 := G_1(x_1);
    x_0 := [x_0, G_2(x_1)];
    ...
    x_0 := [x_0, G_k(x_1)];
    x_1 := H(x_0);
end
```

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

Per RP, assumiamo che $h, g \in$ RICPRIM siano while-programmabili, allora esistono $H, G \in W$-PROG tali che $\Psi_H = h$ e $\Psi_G = g$. Le funzioni ricorsive primitive le possiamo vedere come delle iterazioni che, partendo dal caso base G, mano a mano compongono con H fino a quando non si raggiunge y (escluso). Mostriamo un programma WHILE che calcola

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

```
// Inizialmente in x_1 ho ⟨x, y⟩
input(x, y)
begin
    t := G(x);                    // t contiene f(x, y)
    k := 1;
    while k ⩽ y do
        begin
            t := H(t, k − 1, x);
            x := k + 1;
        end
end
```

Quindi $\Psi_W(\langle x, y\rangle) = \mathrm{RP}(h,g)(\underline{x}, y)$.

□

Abbiamo dimostrato che RICPRIM $\subseteq F$(WHILE) e possiamo anche dire che l'*inclusione è propria*: nel linguaggio WHILE si possono avere cicli infiniti mentre in RICPRIM no in quanto contiene solo funzioni totali (si dimostra per induzione strutturale) mentre WHILE contiene anche delle funzioni parziali. Di conseguenza

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

Sistema di calcolo FOR

Per poter *raggiungere* F(WHILE) bisognerà ampliare RICPRIM. Visto che le funzioni in RICPRIM sono tutte totali, ogni ciclo in RICPRIM ha un inizio e una fine ben definiti: il costrutto utilizzato per dimostrare RP $\in$ F(WHILE) nella dimostrazione precedente permette di definire un nuovo tipo di ciclo, il **ciclo FOR**

```
input(x, y)
begin
    t := G(x);
    for k := 1 to y do
        t := H(t, k − 1, x);
end
```

Il FOR è un costrutto che si serve di una variabile di controllo che parte da un preciso valore e arriva a un valore limite, senza che la variabile di controllo venga toccata.

Il FOR language è quindi un linguaggio WHILE dove l'istruzione di loop è un `for`. Possiamo quindi dire che FOR = RICPRIM, quindi che $F(\text{FOR}) \subset F(\text{WHILE})$.

Dato che WHILE sembra "vincere" su RICPRIM solo per i loop infiniti, restringiamo WHILE imponendo loop finiti. Creiamo

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

Ma dove si posiziona questo insieme rispetto a RICPRIM? L'inclusione è propria? La risposta è sì, ci sono funzioni in $\tilde{F}(\text{WHILE})$ non riscrivibili come funzioni in RICPRIM. Un esempio è la funzione di Ackermann

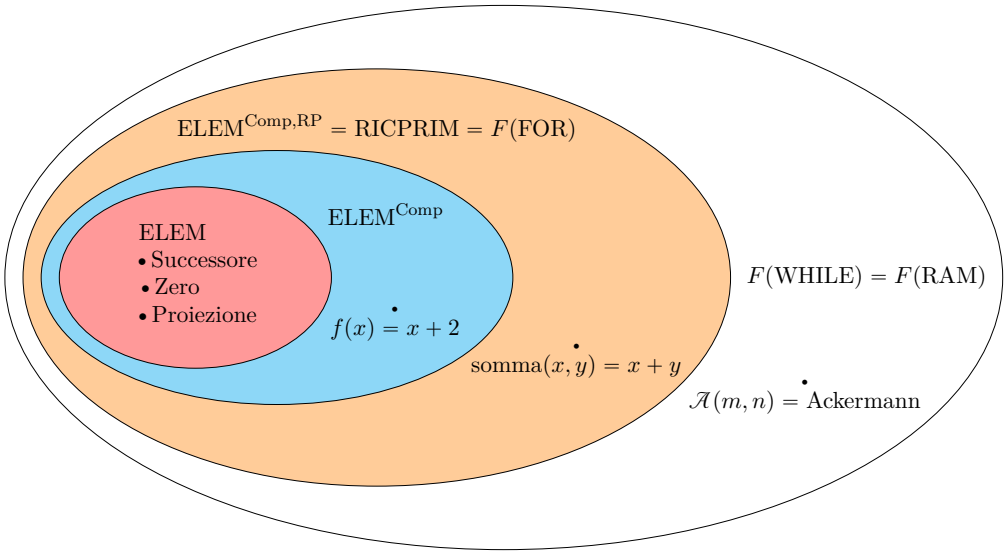
$$\mathcal{A}(m,n) = \begin{cases} n+1 & \text{se } m = 0 \\ \mathcal{A}(m-1,1) & \text{se } m > 0 \wedge n = 0 \\ \mathcal{A}(m-1, A(m,n-1)) & \text{se } m > 0 \wedge n > 0 \end{cases}$$

La quale non appartiene a RICPRIM, per causa della doppia ricorsione cresce troppo in fretta.

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

1.10.3 Ampliamento di RICPRIM

Siamo nella "*direzione giusta*" non avendo cose che non avremmo catturato in $F(\text{RAM})$, ma questo non basta: mancano da catturare le funzioni parziali.



Minimalizzazione

Introduciamo un operatore per permettere la presenza di funzioni parziali: **minimalizzazione**. Sia $f : \mathbb{N}^{n+1} \to \mathbb{N}$ con $f(\underline{x}, y)$ e $\underline{x} \in \mathbb{N}^n$, allora:

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

Un'altra definizione di MIN è

$$\mu_y(f(\underline{x}, y) = 0)$$

Informalmente, restituisce il più piccolo valore di y che azzera $f(\underline{x}, y)$, ovunque precedentemente definita su y'.

Alcuni esempi con $f : \mathbb{N}^2 \to \mathbb{N}$

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

1.10.4 Classe $\mathscr{P}$

Ampliamo RICPRIM chiudendolo con l'operazione MIN

$$\text{ELEM}^{\{\text{COMP,RP,MIN}\}} = \mathscr{P} = \{\text{Funzioni Ricorsive Parziali}\}$$

Abbiamo ottenuto la classe delle **funzioni ricorsive parziali**.

$\mathscr{P}$ vs WHILE

Vogliamo ora analizzare come $\mathscr{P}$ si pone rispetto a $F(\text{WHILE})$, sapendo che *sicuramente* amplia RICPRIM.

Teorema 1.10.2. $\mathscr{P} \subseteq F(\mathit{WHILE})$.

Dimostrazione. $\mathscr{P}$ è definito per chiusura, ma si può anche definire induttivamente:

- le funzioni ELEM sono in $\mathscr{P}$
- se $h, g_1, \ldots, g_k \in \mathscr{P}$ allora $\text{COMP}(h, g_1, \ldots, g_k) \in \mathscr{P}$
- se $h, g \in \mathscr{P}$ allora $\text{RP}(h, g) \in \mathscr{P}$
- se $f \in \mathscr{P}$ allora $\text{MIN}(f) \in \mathscr{P}$
- nient'altro in $\mathscr{P}$

Di conseguenza, per induzione strutturale su $\mathscr{P}$ dimostriamo:

- **Passo base:** le funzioni elementari sono while-programmabili, come già dimostrato
- **Passi induttivi:**
  - siano $h, g_1, \ldots, g_k \in \mathscr{P}$ while-programmabili per I.H., allora mostro che $\text{COMP}(h, g_1, \ldots, g_k)$ è while-programmabile; già fatto per RICPRIM
  - siano $h, g \in \mathscr{P}$ while-programmabili per I.H., mostro che $\text{RP}(h, g)$ è while-programmabile; già fatto per RICPRIM
  - sia $f \in \mathscr{P}$ while-programmabile per I.H., mostro che $\text{MIN}(f)$ è while-programmabile. Devo trovare un programma WHILE che calcoli la minimizzazione:

```
P ≡ input(x)
begin
    y := 0;                                  // parte da 0
    while f(x, y) ≠ 0 do
        y := y + 1;                          // aumenta di 1
end
```

Questo è un programma che calcola la minimizzazione: se non esiste y che azzeri $f(\underline{x}, y)$ il programma va in loop (i.e., incrementa di 1 ad ogni iterazione e termina quando azzera la funzione), quindi la semantica di P è $\perp$ secondo MIN

Concludiamo quindi che $\mathscr{P} \subseteq F(\text{WHILE})$.

□

Viene naturale chiedersi se vale la relazione inversa.

Teorema 1.10.3. *$F(WHILE) \subseteq \mathscr{P}$.*

Dimostrazione. Sappiamo che

$$F(\text{WHILE}) = \{\Psi_W : W \in W\text{-PROG}\}$$

Consideriamo $\Psi_W \in F(\text{WHILE})$ e mostriamo che $\Psi_W \in \mathscr{P}$, facendo vedere che può essere espressa come composizione, ricorsione primitiva e minimalizzazione a partire dalle funzioni in ELEM.

Le funzioni in W-PROG sono nella forma

$$\Psi_W = \text{Pro}_0^{21}\left([\![W]\!](W\text{-in}(\underline{x}))\right)$$

Dove:

- $[\![C]\!](\underline{x}) = \underline{y}$ la funzione che restituisce lo stato prossimo $\underline{y} \in \mathbb{N}^{21}$ a seguito dell'esecuzione del comando C a partire dallo stato corrente $\underline{x} \in \mathbb{N}^{21}$

- La funzione W-in(n) restituisce lo stato iniziale della macchina WHILE su input n
- Pro_0^{21} preleva l'output dal registro x_0

Abbiamo definito Ψ_W come composizione delle funzioni Pro_0^{21} e $[\![W]\!]W\text{-in}(\underline{x})$, ma allora

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

Di conseguenza, se dimostro che la funzione di stato prossimo $[\![C]\!]$ è ricorsiva parziale $\in \mathscr{P}$ allora anche $\Psi_W \in \mathscr{P}$, per la definizione induttiva di $\mathscr{P}$.

La funzione di stato prossimo $[\![]\!]() : \mathbb{N}^{21} \to \mathbb{N}^{21}$ restituisce elementi in $\mathbb{N}^{21}$, mentre gli elementi in $\mathscr{P}$ hanno codominio $\mathbb{N}$.

Per risolvere il problema si definisce la **funzione f_C numero prossimo** in cui viene applicato Cantor sull'array degli stati.

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

$$f_C(x) = y \quad \text{con} \quad x = [\underline{x}] \ , \ y = [\underline{y}]$$

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

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

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

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

Dimostriamo, tramite induzione strutturale, sul comando while C:

- **Passo base**: azzeramento e incremento/decremento

– $C \equiv \boxed{x_k := 0}$

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

Viene usata una composizione di funzioni $\in \mathscr{P} \Rightarrow f_{x_k:=0} \in \mathscr{P}$

– $C \equiv \boxed{x_k := x_j +/ \dot{-} 1}$

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

Viene usata una composizione di funzioni $\in \mathscr{P} \Rightarrow f_{x_k:=x_j \pm 1} \in \mathscr{P}$

- **Passi induttivi**: i comandi "complessi" WHILE devono rientrare in $\mathscr{P}$

– $C \equiv \boxed{\texttt{begin } C_1;\ C_2;\ \ldots;\ C_m;\ \texttt{end}}$

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

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

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

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

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

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

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

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

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

Siccome:

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

$$\implies \quad T(x,y) \in \mathscr{P}$$

L'ultima cosa da sistemare è $e(x)$:

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

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

In conclusione:

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

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

□

Visti i risultati ottenuti dai due teoremi precedenti, possiamo concludere che:

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

La classe delle funzioni ricorsive parziali, ovvero l'idea di *calcolabile* in termini matematici, coincide con l'idea di *calcolabile* proveniente dall'insieme di problemi per i quali vediamo una macchina che li risolva.

1.10.5 Tesi di Church-Turing

Il risultato principale di questo studio è aver trovato due classi di funzioni importanti:

- $\mathscr{P}$ insieme delle **funzioni ricorsive parziali**
- $\mathscr{T}$ insieme delle **funzioni ricorsive totali**

Il primo insieme contiene anche tutte le funzioni del secondo, quindi

$$\mathscr{T} \subset \mathscr{P}$$

Inoltre abbiamo visto (ad esempio tramite la funzione di Ackermann) che

$$\text{RICPRIM} \subset \mathscr{T}$$

L'insieme $\mathscr{P}$ cattura tutti i sistemi di calcolo esistenti: WHILE, RAM, Macchine di Turing, Lambda-calcolo di Church, …
Tutti i modelli di calcolo proposti alla fine individuano sempre la classe delle funzioni ricorsive parziali. Visti questi risultati, Church e Turing decidono di enunciare un risultato molto importante.

Tesi di Church-Turing: La classe delle funzioni intuitivamente calcolabili coincide con la classe $\mathscr{P}$ delle funzioni ricorsive parziali.

Questa è una tesi e non un teorema in quanto non è possibile caratterizzare i modelli di calcolo ragionevoli che sono stati e saranno proposti in maniera completa (potrebbe non essere vera, ma per ora nessuno è riuscito a provarlo). Possiamo solo decidere se aderire o meno a questa tesi.

Per noi un problema è *calcolabile* quando esiste un modello di calcolo che riesce a risolverlo ragionevolmente. Se volessimo aderire alla tesi di Church-Turing, potremmo dire, in maniera più formale, che:

- *problema ricorsivo parziale* è sinonimo di calcolabile
- *problema ricorsivo totale* è sinonimo di calcolabile da un programma che si arresta su ogni input

1.11 Sistemi di Programmazione