Informatica teorica

Indice

1. Lezione 01

1.1. Introduzione

1.1.1. Definizione

L'**informatica teorica** è quella branca dell'informatica che si "contrappone" all'informatica applicata: in quest'ultima, l'informatica è usata solo come uno *strumento* per gestire l'oggetto del discorso, mentre nella prima l'informatica diventa l'*oggetto* del discorso, di cui ne vengono studiati i fondamenti.

1.1.2. Cosa e come

Analizziamo i due aspetti fondamentali che caratterizzano ogni materia:

1. il **cosa**: l'informatica è la scienza che studia l'informazione e la sua elaborazione automatica mediante un sistema di calcolo. Ogni volta che ho un *problema* cerco di risolverlo automaticamente scrivendo un programma. *Posso farlo per ogni problema? Esistono problemi che non sono risolubili?* Possiamo chiamare questo primo aspetto con il nome di **teoria della calcolabilità**, quella branca che studia cosa è calcolabile e cosa non lo è, a prescindere dal costo in termini di risorse che ne deriva. In questa parte utilizzeremo una caratterizzazione molto rigorosa e una definizione di problema il più generale possibile, così che l'analisi non dipenda da fattori tecnologici, storici...
2. il **come**: è relazionato alla **teoria della complessità**, quella branca che studia la quantità di risorse computazionali richieste nella soluzione automatica di un problema. Con *risorsa computazionale* si intende qualsiasi cosa venga consumato durante l'esecuzione di un programma, ad esempio:
   - elettricità;
   - numero di processori;
   - tempo;
   - spazio di memoria.

   In questa parte daremo una definizione rigorosa di tempo, spazio e di problema efficientemente risolubile in tempo e spazio, oltre che uno sguardo al famoso problema `P = NP`.

Possiamo dire che il *cosa* è uno studio **qualitativo**, mentre il *come* è uno studio **quantitativo**.

Grazie alla teoria della calcolabilità individueremo le funzioni calcolabili, di cui studieremo la complessità.

1.2. Richiami matematici

1.2.1. Funzioni

Una **funzione** da un insieme A ad un insieme B è una *legge*, spesso indicata con f, che spiega come associare agli elementi di A un elemento di B.

Abbiamo due tipi di funzioni:

- **generale**: la funzione è definita in modo generale come $f : A \longrightarrow B$, in cui A è detto **dominio** di f e B è detto **codominio** di f;
- **locale/puntuale**: la funzione riguarda i singoli valori a e b:

$$f(a) = b \quad | \quad a \overset{f}{\longmapsto} b$$

  in cui b è detta **immagine** di a rispetto ad f e a è detta **controimmagine** di b rispetto ad f.

Possiamo categorizzare le funzioni in base ad alcune proprietà:

- **iniettività**: una funzione $f : A \longrightarrow B$ si dice *iniettiva* se e solo se:

$$\forall a_1, a_2 \in A \quad a_1 \neq a_2 \Longrightarrow f(a_1) \neq f(a_2)$$

In poche parole, non ho *confluenze*, ovvero *elementi diversi finiscono in elementi diversi.*

- **suriettività**: una funzione $f : A \longrightarrow B$ si dice *suriettiva* se e solo se:

$$\forall b \in B \quad \exists a \in A \mid f(a) = b.$$

In poche parole, *ogni elemento del codominio ha almeno una controimmagine.*

Se definiamo l'**insieme immagine**:

$$\mathrm{Im}_f = \{b \in B \mid \exists a \in A \text{ tale che } f(a) = b\} = \{f(a) \mid a \in A\} \subseteq B$$

possiamo dare una definizione alternativa di funzione suriettiva, in particolare una funzione è *suriettiva* se e solo se $\mathrm{Im}_f = B$.

Infine, una funzione $f : A \longrightarrow B$ si dice **biiettiva** se e solo se è iniettiva e suriettiva, quindi vale: $\forall b \in B \quad \exists! a \in A \mid f(a) = b$.

Se $f : A \longrightarrow B$ è una funzione biiettiva, si definisce **inversa** di f la funzione $f^{-1} : B \longrightarrow A$ tale che:

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

Per definire la funzione inversa f^{-1}, la funzione f deve essere biiettiva: se così non fosse, la sua inversa avrebbe problemi di definizione.

Un'operazione definita su funzioni è la **composizione**: date $f : A \longrightarrow B$ e $g : B \longrightarrow C$, la funzione f *composto* g è la funzione $g \circ f : A \longrightarrow C$ definita come $(g \circ f)(a) = g(f(a))$.

La composizione *non è commutativa*, quindi $g \circ f \neq f \circ g$ in generale, ma è *associativa*, quindi $(f \circ g) \circ h = f \circ (g \circ h)$.

La composizione *f composto g* la possiamo leggere come *prima agisce f poi agisce g*.

Dato l'insieme A, la **funzione identità** su A è la funzione $i_A : A \longrightarrow A$ tale che:

$$i_A(a) = a \quad \forall a \in A,$$

ovvero è una funzione che mappa ogni elemento su se stesso.

Grazie alla funzione identità, possiamo dare una definizione alternativa di funzione inversa: data la funzione $f : A \longrightarrow B$ biiettiva, la sua inversa è l'unica funzione $f^{-1} : B \longrightarrow A$ che soddisfa:

$$f \circ f^{-1} = f^{-1} \circ f = \mathrm{id}_A .$$

Possiamo vedere f^{-1} come l'unica funzione che ci permette di *tornare indietro.*

2. Lezione 02

2.1. Richiami matematici

2.1.1. Funzioni totali e parziali

Prima di fare un'ulteriore classificazione per le funzioni, introduciamo la notazione $f(a)\downarrow$ per indicare che la funzione f è definita per l'input a, mentre la notazione $f(a)\uparrow$ per indicare la situazione opposta.

Ora, data $f : A \longrightarrow B$ diciamo che f è:
- **totale** se è definita *per ogni elemento* $a \in A$, ovvero $f(a)\downarrow \ \forall a \in A$;
- **parziale** se è definita *per qualche elemento* $a \in A$, ovvero $\exists a \in A \mid f(a)\uparrow$.

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

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

Notiamo che:
- $\mathrm{Dom}_f \subsetneq A \Longrightarrow f$ parziale;
- $\mathrm{Dom}_f = A \Longrightarrow f$ totale.

2.1.2. Totalizzare una funzione

È possibile *totalizzazione* una funzione parziale definendo una funzione a tratti $f : A \longrightarrow B \cup \{\perp\}$ tale che

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

Il nuovo simbolo introdotto è il *simbolo di indefinito*, e viene utilizzato per tutti i valori per cui la funzione di partenza f non è appunto definita.

Da qui in avanti utilizzeremo $B_\perp$ come abbreviazione di $B \cup \{\perp\}$.

2.1.3. Prodotto cartesiano

Chiamiamo **prodotto cartesiano** l'insieme

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

che rappresenta l'insieme di tutte le *coppie ordinate* di valori in A e in B.

In generale, il prodotto cartesiano **non è commutativo**, a meno che $A = B$.

Possiamo estendere il concetto di prodotto cartesiano a n-uple di valori, ovvero

$$A_1 \times ... \times A_n = \{(a_1, ..., a_n) \mid a_i \in A_i\}.$$

L'operazione "*opposta*" è effettuata dal **proiettore** i-esimo: esso è una funzione che estrae l'i-esimo elemento di un tupla, ovvero è una funzione $\pi_i : A_1 \times ... \times A_n \longrightarrow A_i$ tale che

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

Per comodità chiameremo $\underbrace{A \times ... \times A}_{n} = A^n$

2.1.4. Insieme di funzioni

L'insieme di tutte le funzioni da A in B si indica con

$$B^A = \{f : A \longrightarrow B\}.$$

Viene utilizzata questa notazione in quanto la cardinalità di B^A è esattamente $|B|^{|A|}$, se A e B sono insiemi finiti.

In questo insieme sono presenti anche tutte le funzioni parziali da A in B: mettiamo in evidenza questa proprietà, scrivendo

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

Sembrano due insiemi diversi ma sono la stessa cosa: nel secondo viene messo in evidenza il fatto che tutte le funzioni che sono presenti sono totali oppure parziali che sono state totalizzate.

2.1.5. Funzione di valutazione

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

$$\omega : B_\perp^A \times A \longrightarrow B$$

tale che

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

In poche parole, è una funzione che prende una funzione f e la valuta su un elemento a del dominio.

Abbiamo due possibili approcci:
- tengo fisso a e provo tutte le funzioni f: sto eseguendo un *benchmark*, quest'ultimo rappresentato da a;
- tengo fissa f e provo tutte le a del dominio: sto ottenendo il grafico di f.

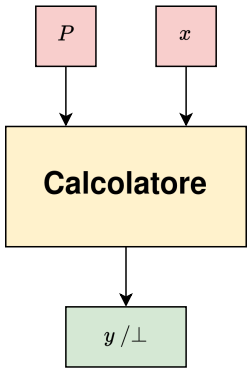
2.2. Teoria della calcolabilità

2.2.1. Sistema di calcolo

Quello che faremo ora è *modellare teoricamente un sistema di calcolo.*

Un **sistema di calcolo** lo possiamo vedere come una *black-box* che prende un programma P, una serie di dati x e calcola il risultato di P sull'input x.

La macchina ci restituisce:
- y se è riuscita a calcolare un risultato;
- $\perp$ se è entrata in un loop.



Un sistema di calcolo quindi è una funzione del tipo

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

Come vediamo, è molto simile ad una funzione di valutazione: infatti, la parte dei dati x la riusciamo a "mappare" sull'input a del dominio, ma facciamo più fatica con l'insieme dei programmi, questo perché non abbiamo ancora definito cos'è un programma.

Un **programma** P è una **sequenza di regole** che trasformano un dato di input in uno di output (o in un loop): in poche parole, un programma è una funzione del tipo

$$P : \mathrm{DATI} \longrightarrow \mathrm{DATI}_\perp,$$

ovvero una funzione che appartiene all'insieme $\mathrm{DATI}_\perp^{\mathrm{DATI}}$.

Con questa definizione riusciamo a "mappare" l'insieme PROG sull'insieme delle funzioni che ci serviva per definire la funzione di valutazione.

Formalmente, un sistema di calcolo è quindi la funzione

$$\mathcal{C} : \mathrm{DATI}_\perp^{\mathrm{DATI}} \times \mathrm{DATI} \longrightarrow \mathrm{DATI}.$$

Con $\mathcal{C}(P, x)$ indichiamo la funzione calcolata da P su x, ovvero la sua **semantica**, il suo *significato*.

Il modello classico che viene considerato quando si parla di calcolatori è quello di **Von Neumann**.

2.2.2. Potenza computazionale

Prima di definire la potenza computazionale facciamo una breve premessa: indichiamo con

$$\mathcal{C}(P, \_) : \mathrm{DATI} \longrightarrow \mathrm{DATI}_\perp$$

la funzione che viene calcolata dal programma P, ovvero la semantica di P.

Fatta questa premessa, definiamo la **potenza computazionale** di un calcolatore come l'insieme di tutte le funzioni che quel sistema di calcolo può calcolare grazie ai programmi, ovvero

$$F(\mathcal{C}) = \{\mathcal{C}(P, \_) \mid P \in \mathrm{PROG}\} \subseteq \mathrm{DATI}_\perp^{\mathrm{DATI}}.$$

In poche parole, $F(\mathcal{C})$ rappresenta l'insieme di tutte le possibili semantiche di funzioni che sono calcolabili con il sistema $\mathcal{C}$.

Stabilire *cosa può fare l'informatica* equivale a studiare quest'ultima inclusione: in particolare, se
- $F(\mathcal{C}) \subsetneq \mathrm{DATI}_\perp^{\mathrm{DATI}}$ allora esistono compiti **non automatizzabili**;
- $F(\mathcal{C}) = \mathrm{DATI}_\perp^{\mathrm{DATI}}$ allora l'informatica può fare tutto.

Se ci trovassimo nella prima situazione dovremmo individuare una sorta di "recinto" per dividere le funzioni calcolabili da quelle che non lo sono.

Calcolare una funzione vuole dire *risolvere un problema*: ad ognuno di questi associo una **funzione soluzione**, che mi permette di risolvere in modo automatico il problema.

Grazie a questa definizione, calcolare le funzioni equivale a risolvere problemi.

In che modo possiamo risolvere l'inclusione?

Un primo approccio è quello della **cardinalità**: viene definita come una funzione che associa ad ogni insieme il numero di elementi che contiene.

Sembra un ottimo approccio, ma presenta alcuni problemi: infatti, funziona solo quando l'insieme è finito, mentre è molto fragile quando si parla di insiemi infiniti.

Ad esempio, gli insiemi
- $\mathbb{N}$ dei numeri naturali e
- $\mathbb{P}$ dei numeri naturali pari

sono tali che $\mathbb{P} \subsetneq \mathbb{N}$ ma hanno cardinalità $|\mathbb{N}| = |\mathbb{P}| = \infty$.

Dobbiamo dare una definizione diversa di cardinalità, visto che possono esistere "*infiniti più fitti/densi di altri*", come abbiamo visto nell'esempio precedente.

3. Lezione 03

3.1. Relazioni di equivalenza

3.1.1. Definizione

Una **relazione binaria** R su due insiemi A, B è un sottoinsieme $R \subseteq A \times B$ di coppie ordinate. Una relazione particolare che ci interessa è $R \subseteq A^2$. Due elementi $a, b \in A$ sono in relazione R se e solo se $(a, b) \in R$. Indichiamo la relazione tra due elementi tramite notazione infissa aRb.

Una classe molto importante di relazioni è quella delle **relazioni di equivalenza**: una relazione $R \subseteq A^2$ è una relazione di equivalenza se e solo se R è RST, ovvero:

- **riflessiva**: $\forall a \in A \quad aRa$;
- **simmetrica**: $\forall a, b \in A \quad aRb \implies bRa$;
- **transitiva**: $\forall a, b, c \in A \quad aRb \land bRc \implies aRc$.

3.1.2. Partizione

Ad ogni relazione di equivalenza si può associare una **partizione**, ovvero un insieme di sottoinsiemi tali che:

- $\forall i \in \mathbb{N}^+ \quad A_i \neq \varnothing$;
- $\forall i, j \in \mathbb{N}^+ \quad i \neq j \implies A_i \cap A_j = \varnothing$;
- $\bigcup_{i \in \mathbb{N}^+} A_i = A$.

Diremo che R definita su A^2 *induce* una partizione $A_1, A_2, \ldots$ su A.

3.1.3. Classi di equivalenza e insieme quoziente

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

$$[a]_R = \{b \in A \mid aRb\},$$

ovvero tutti gli elementi che sono in relazione con a, chiamato anche *rappresentante della classe*.

Si può dimostrare che:

- non esistono classi di equivalenza vuote $\rightarrow$ garantito dalla riflessività;
- dati $a, b \in A$ allora $[a]_R \cap [b]_R = \varnothing$ oppure $[a]_R = [b]_R \rightarrow$ in altre parole, due elementi o sono in relazione o non lo sono;
- $\bigcup_{a \in A} [a]_R = A$.

Notiamo che, per definizione, l'insieme delle classi di equivalenza è una partizione indotta dalla relazione R sull'insieme A. Questa partizione è detta **insieme quoziente** di A rispetto a R ed è denotato con A/R.

3.2. Cardinalità

3.2.1. Isomorfismi

Due insiemi A e B sono **isomorfi** (*equinumerosi* o *insiemi che hanno la stessa cardinalità*) se esiste una biiezione tra essi. Formalmente scriviamo:

$$A \sim B.$$

Detto $\mathcal{U}$ l'insieme di tutti gli insiemi, la relazione $\sim$ è sottoinsieme di $\mathcal{U}^2$.

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

- *riflessività*: $A \sim A$ se la biiezione è i_A;
- *simmetria*: $A \sim B \implies B \sim A$ se la biiezione è la funzione inversa;

- *transitività*: $A \sim B \land B \sim C \implies A \sim C$ se la biiezione è la composizione della funzione usata per $A \sim B$ con la funzione usata per $B \sim C$.

Dato che $\sim$ è una relazione di equivalenza, è possibile partizionare l'insieme $\mathcal{U}$. La partizione creata è formata da classi di equivalenza che contengono insiemi isomorfi, ossia con la stessa cardinalità.

Possiamo quindi definire la **cardinalità** come l'insieme quoziente di $\mathcal{U}$ rispetto alla relazione $\sim$.

Questo approccio permette di utilizzare la nozione di *cardinalità* anche con gli insiemi infiniti, dato che l'unica incognita da trovare è una funzione biettiva tra i due insiemi.

3.2.2. Cardinalità finita

La prima classe di cardinalità che vediamo è quella delle **cardinalità finite**. Prima di tutto definiamo la famiglia di insiemi:

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

Un insieme A ha cardinalità finita se $A \sim J_n$ per qualche $n \in \mathbb{N}$. In tal caso possiamo scrivere $|A| = n$.

La classe di equivalenza $[J_n]_\sim$ riunisce tutti gli insiemi di $\mathcal{U}$ contenenti n elementi.

3.2.3. Cardinalità infinita

L'altra classe di cardinalità da studiare è quella delle **cardinalità infinite**, ovvero gli insiemi non in relazione con J_n.

3.2.3.1. Insiemi numerabili

I primi insiemi a cardinalità infinita sono gli **insiemi numerabili**. Un insieme A è numerabile se e solo se $A \sim \mathbb{N}$, ovvero $A \in [\mathbb{N}]_\sim$.

Gli insiemi numerabili sono "**listabili**", ovvero è possibile elencare *tutti* gli elementi dell'insieme A tramite una regola, in questo caso la funzione f biiezione tra $\mathbb{N}$ e A. Infatti, grazie alla funzione f, è possibile elencare gli elementi di A formando l'insieme:

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

Questo insieme è esaustivo, quindi elenca ogni elemento dell'insieme A senza perderne nessuno.

Gli insiemi numerabili più famosi sono:
- numeri pari $\mathbb{P}$ e numeri dispari $\mathbb{D}$;
- numeri interi $\mathbb{Z}$ generati con la biiezione $f(n) = (-1)^n \left(\frac{n + (n \bmod 2)}{2} \right)$;
- numeri razionali $\mathbb{Q}$.

Gli insiemi numerabili hanno cardinalità $\aleph_0$ (si legge *"aleph"*).

3.2.3.2. Insiemi non numerabili

Gli **insiemi non numerabili** sono insiemi a cardinalità infinita che non sono listabili come gli insiemi numerabili, ovvero sono "più fitti" di $\mathbb{N}$.

Il *non poter listare gli elementi* si traduce in *qualunque lista generata mancherebbe di qualche elemento*, di conseguenza non sarebbe una lista esaustiva di tutti gli elementi.

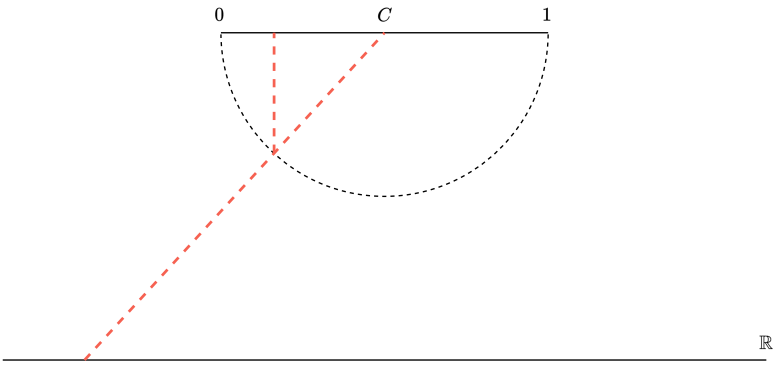
Il più famoso insieme non numerabile è l'insieme dei numeri reali $\mathbb{R}$.

Teorema L'insieme $\mathbb{R}$ non è numerabile

Dimostrazione
Suddividiamo la dimostrazione in tre punti:
1. dimostriamo che $\mathbb{R} \sim (0,1)$;
2. dimostriamo che $\mathbb{N} \nsim (0,1)$;
3. dimostriamo che $\mathbb{R} \nsim \mathbb{N}$.

[1] Partiamo con il dimostrare che $\mathbb{R} \sim (0,1)$: mostro che esiste una biiezione tra $\mathbb{R}$ e $(0,1)$. Usiamo una biiezione "grafica" costruita in questo modo:
- disegna la circonferenza di raggio $\frac{1}{2}$ centrata in $\frac{1}{2}$;
- disegna la perpendicolare al punto da mappare che interseca la circonferenza;
- disegna la retta passante per il centro C e l'intersezione precedente.

L'intersezione tra l'asse reale e la retta finale è il punto mappato.



In realtà, $\mathbb{R}$ è isomorfo a qualsiasi segmento di lunghezza maggiore di 0.

La stessa biiezione vale anche sull'intervallo chiuso $[0,1]$ utilizzando la "compattificazione" $\mathring{\mathbb{R}} = \mathbb{R} \cup \{\pm\infty\}$ di $\mathbb{R}$, mappando 0 su $-\infty$ e 1 su $+\infty$.

[2] Continuiamo dimostrando che $\mathbb{N} \nsim (0,1)$: devo dimostrare che l'intervallo $(0,1)$ non è listabile, ovvero ogni lista che scrivo è un "colabrodo", termine tecnico che indica la possibilità di costruire un elemento che dovrebbe appartenere alla lista ma che invece non è presente.
Per assurdo sia $\mathbb{N} \sim (0,1)$, allora posso listare gli elementi di $(0,1)$ esaustivamente come:

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

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

Costruisco il *"numero colpevole"* $c = 0.c_0c_1c_2\dots$ tale che

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

In poche parole, questo numero è costruito "guardando" tutte le cifre sulla diagonale.

Questo numero sicuramente appartiene a $(0,1)$ ma non appare nella lista: infatti ogni cifra c_i del colpevole differisce da qualunque numero nella lista in almeno una posizione, che è quella della diagonale. Ma questo è assurdo: avevamo assunto $(0,1)$ numerabile.

Quindi $N \nsim (0,1)$.

Questo tipo di dimostrazione è detta **dimostrazione per diagonalizzazione**.

[3] Terminiamo dimostrando che $\mathbb{R} \nsim \mathbb{N}$: per transitività. Vale il generico, ovvero non si riesce a listare nessun segmento di lunghezza maggiore di 0. □

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

- $\mathbb{R}$ insieme dei numeri reali;
- $\mathbb{C}$ insieme dei numeri complessi;
- $\mathbb{T} \subset \mathbb{I}$ insieme dei numeri trascendenti.

4. Lezione 04

4.1. Cardinalità

Vediamo due insiemi continui che saranno importanti successivamente.

4.1.1. Insieme delle parti

Il primo insieme che vediamo è l'**insieme delle parti**, o *power set*, di $\mathbb{N}$.

Quest'ultimo è l'insieme

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

Teorema $P(\mathbb{N}) \nsim \mathbb{N}$.

Dimostrazione
Dimostriamo questo teorema con la diagonalizzazione.

Rappresentiamo il sottoinsieme $A \subseteq \mathbb{N}$ tramite il suo **vettore caratteristico**:

$$\begin{aligned} &\mathbb{N} : 0\ 1\ 2\ 3\ 4\ 5\ 6\ \ldots \\ &A : 0\ 1\ 1\ 0\ 1\ 1\ 0\ \ldots \quad . \end{aligned}$$

Il vettore caratteristico di un sottoinsieme è un vettore che nella posizione p_i ha 1 se $i \in A$, altrimenti ha 0.

Per assurdo sia $P(\mathbb{N})$ numerabile. Vista questa proprietà posso listare tutti i vettori caratteristici che appartengono a $P(\mathbb{N})$ come

$$\begin{aligned} b_0 &= b_{00}\ b_{01}\ b_{02}\ \ldots \\ b_1 &= b_{10}\ b_{11}\ b_{12}\ \ldots \\ b_2 &= b_{20}\ b_{21}\ b_{22}\ \ldots \quad . \end{aligned}$$

Costruiamo un *colpevole among us* che appartiene a $P(\mathbb{N})$ ma non è presente nella lista precedente. Definiamo il vettore

$$c = \overline{b_{00}}\ \overline{b_{11}}\ \overline{b_{22}}\ldots$$

che contiene nella posizione c_i il complemento di b_{ii}.

Questo vettore appartiene a $P(\mathbb{N})$ ma non è presente nella lista precedente perché è diverso da ogni elemento della lista in almeno una cifra.

Ma questo è assurdo perché $P(\mathbb{N})$ era numerabile, quindi $P(\mathbb{N}) \nsim \mathbb{N}$. □

Visto questo teorema possiamo affermare che:

$$P(\mathbb{N}) \sim [0,1] \sim \mathring{\mathbb{R}}.$$

4.1.2. Insieme delle funzioni

Il secondo insieme che vediamo è l'insieme delle funzioni da $\mathbb{N}$ in $\mathbb{N}$.

Quest'ultimo è l'insieme

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

Teorema $\mathbb{N}_{\perp}^{\mathbb{N}} \nsim \mathbb{N}$.

Dimostrazione
Anche in questo caso useremo la dimostrazione per diagonalizzazione.

Per assurdo sia $\mathbb{N}_\perp^{\mathbb{N}}$ numerabile, quindi possiamo listare $\mathbb{N}_\perp^{\mathbb{N}}$ come $\{f_0, f_1, f_2, ...\}$.

	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	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$	...	...
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$	...	...

Scriviamo un colpevole $\varphi : \mathbb{N} \longrightarrow \mathbb{N}_\perp$ per dimostrare l'assurdo. Una prima versione potrebbe essere la funzione $\varphi(n) = f_n(n) + 1$ per *disallineare* la diagonale, ma questo non va bene: infatti, se $f_n(n) = \perp$ non sappiamo dare un valore a $\varphi(n) = \perp + 1$.

Definiamo quindi la funzione

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

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

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

Ma questo è assurdo perché $P(\mathbb{N})$ era numerabile, quindi $P(\mathbb{N}) \nsim \mathbb{N}$. □

4.2. Potenza computazionale

4.2.1. Validità dell'inclusione $F(\mathcal{C}) \subseteq \text{DATI}_\perp^{\text{DATI}}$

Ora che abbiamo una definizione "potente" di cardinalità, essendo basata su strutture matematiche, possiamo verificare la validità dell'inclusione

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

Diamo prima qualche considerazione:
- $\text{PROG} \sim \mathbb{N}$: identifico ogni programma con un numero, ad esempio la sua codifica in binario;
- $\text{DATI} \sim \mathbb{N}$: come prima, identifico ogni dato con la sua codifica in binario.

In poche parole, stiamo dicendo che programmi e dati non sono più dei numeri naturali $\mathbb{N}$.

Ma questo ci permette di dire che:

$$F(\mathcal{C}) \sim \text{PROG} \sim \mathbb{N} \nsim \mathbb{N}_\perp^{\mathbb{N}} \sim \text{DATI}_\perp^{\text{DATI}}.$$

Questo è un risultato importantissimo: abbiamo appena dimostrato con la relazione precedente che **esistono funzioni non calcolabili**. Le funzioni non calcolabili sono problemi pratici e molto sentiti

al giorno d'oggi: un esempio di funzione non calcolabile è la funzione che, dato un software, dice se è corretto o no. Il problema è che *ho pochi programmi e troppe/i funzioni/problemi*.

Questo risultato però è arrivato considerando vere le due considerazioni precedenti: andiamo quindi a dimostrarle utilizzando le **tecniche di aritmetizzazione** (o *godelizzazione*) **di strutture**, ovvero delle tecniche che rappresentano delle strutture con un numero, così da avere la matematica e l'insiemi degli strumenti che ha a disposizione.

5. Lezione 05

5.1. DATI $\sim \mathbb{N}$

Vogliamo formare 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 per lavorare il numero che "riflettano" il risultato sul dato senza ripassare per il dato stesso;
3. ci consenta di dire, senza perdita di generalità, che i nostri programmi lavorano sui numeri.

5.1.1. Funzione coppia di Cantor

5.1.1.1. Definizione

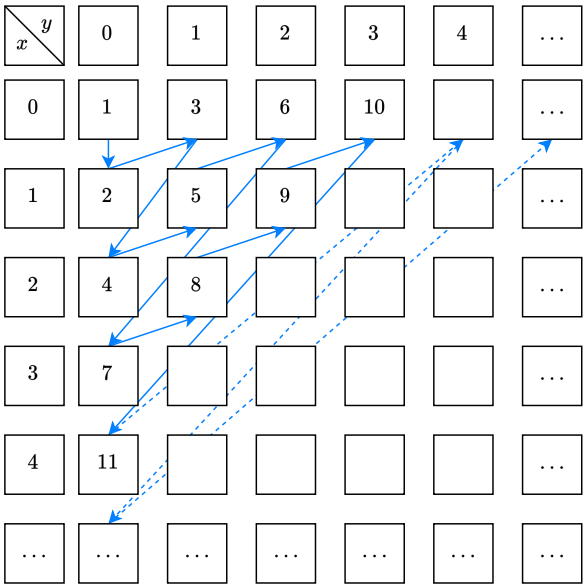
La **funzione coppia di Cantor** è la funzione

$$<,>: \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}^+.$$

Questa funzione sfrutta due *"sotto-funzioni"*, *sin* e *des*, tali che

$$< x, y >= n,$$
$$\text{sin} : \mathbb{N}^+ \longrightarrow \mathbb{N}, \quad \text{sin } (n) = x$$
$$\text{des} : \mathbb{N}^+ \longrightarrow \mathbb{N}, \quad \text{des } (n) = y.$$

Vediamo una rappresentazione grafica della funzione di Cantor.



$< x, y >$ rappresenta il valore all'incrocio tra la x-esima riga e la y-esima colonna.

La tabella viene riempita *diagonale per diagonale*, ovvero:

1. sia $x = 0$;
2. partendo dalla cella $(x, 0)$ si enumerano le celle della diagonale identificata da $(x, 0)$ e da $(0, x)$;
3. si ripete il punto 2 aumentando x di 1.

Vorremmo che questa funzione sia iniettiva e suriettiva, quindi:

- non posso avere celle con lo stesso numero (*iniettiva*);
- ogni numero in $\mathbb{N}^+$ deve comparire.

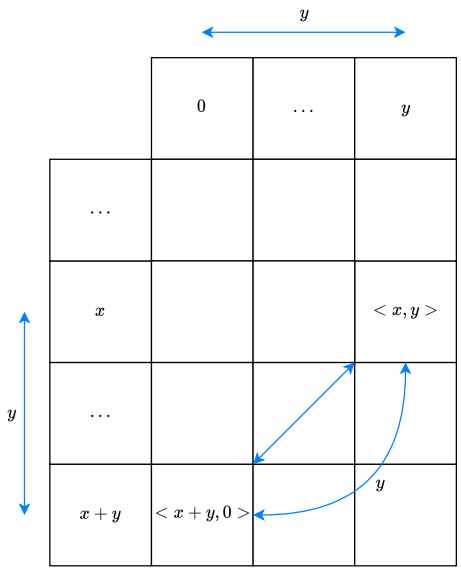
Questa richiesta è soddisfatta in quanto:

- numeriamo in maniera incrementale (*iniettiva*);
- ogni numero prima o poi compare in una cella, quindi ho una coppia che lo genera (*suriettiva*).

5.1.1.2. Forma analitica della funzione coppia

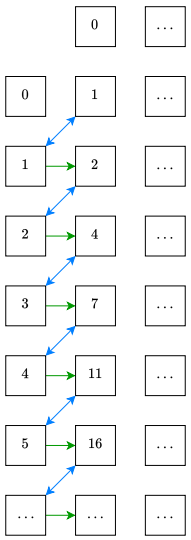
Quello che vogliamo fare ora è cercare una forma analitica della funzione coppia, questo perché non è molto comodo costruire ogni volta la tabella sopra. Nella successiva immagine notiamo come valga la relazione

$$< x, y >=< x + y, 0 > +y.$$



Questo è molto comodo perché il calcolo della funzione coppia si riduce al calcolo di $< x + y, 0 >$.

Chiamiamo $x + y = z$, osserviamo con la successiva immagine un'altra proprietà.



Ogni cella $< z, 0 >$ la si può calcolare come la somma di z e $< z - 1, 0 >$, ma allora

$$
\begin{aligned}
< z, 0 >= z + < z-1, 0 >= \\
&= z + (z-1) + < z-2, 0 >= \\
&= z + (z-1) + ... + 1 + < 0, 0 >= \\
&= z + (z-1) + ... + 1 + 1 = \\
&= \sum_{i=1}^{z} i + 1 = \frac{z(z+1)}{2} + 1.
\end{aligned}
$$

Questa forma è molto più compatta ed evita il calcolo di tutti i singoli $< z, 0 >$.

Mettiamo insieme le due proprietà per ottenere la formula analitica della funzione coppia:

$$
< x, y >=< x + y, 0 > + y = \frac{(x+y)(x+y+1)}{2} + y + 1.
$$

5.1.1.3. Forma analitica di sin e des

Vogliamo adesso dare la forma analitica di *sin* e *des* per poter computare l'inversa della funzione di Cantor, dato n.

Grazie alle osservazioni precedenti sappiamo che

$$
\begin{aligned}
n = y + < \gamma, 0 > \implies y = n - < \gamma, 0 >, \\
\gamma = x + y \implies x = \gamma - y.
\end{aligned}
$$

Se troviamo il valore di gamma abbiamo trovato anche i valori di x e y.

Notiamo come γ sia il *"punto di attacco"* della diagonale che contiene n, ma allora

$$
\gamma = \max\{z \in \mathbb{N} \mid < z, 0 >\leq n\}
$$

perché tra tutti i punti di attacco $< z, 0 >$ voglio quello che potrebbe contenere n e che sia massimo, ovvero sia esattamente la diagonale che contiene n.

Risolviamo quindi la disequazione

$$
\begin{aligned}
< z, 0 >\leq n &\implies \frac{z(z+1)}{2} + 1 \leq n \\
&\implies z^2 + z - 2n + 2 \leq 0 \\
&\implies z_{1,2} = \frac{-1 \pm \sqrt{1 + 8n - 8}}{2} \\
&\implies \frac{-1 - \sqrt{8n-7}}{2} \leq z \leq \frac{-1 + \sqrt{8n-7}}{2}.
\end{aligned}
$$

Come valore di γ scelgo

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

Ora che abbiamo γ possiamo definire le funzioni *sin* e *des* come

$$
\begin{aligned}
\text{des}\ (n) = y = n - < \gamma, 0 >= n - \frac{\gamma(\gamma+1)}{2} - 1, \\
\text{sin}\ (n) = x = \gamma - y.
\end{aligned}
$$

5.1.1.4. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$

Con la funzione coppia di Cantor possiamo dimostrare un importante risultato.

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

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

Estendiamo adesso il risultato all'interno insieme $\mathbb{N}$, ovvero

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

Teorema $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$.

Dimostrazione
Definiamo la funzione

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

tale che

$$[x, y] =< x, y > -1.$$

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

Grazie a questi risultati si può dimostrare che $\mathbb{Q} \sim \mathbb{N}$: infatti, i numeri razionali li possiamo rappresentare come coppie (num, den). In generale, tutte le tuple sono isomorfe a $\mathbb{N}$, iterando in qualche modo la funzione coppia di Cantor.

5.1.2. Dimostrazione

I risultati ottenuti fino a questo punto ci permettono di dire che ogni dato è trasformabile in un numero, che può essere soggetto a trasformazioni e manipolazioni matematiche.

La dimostrazione *formale* non verrà fatta, ma verranno fatti esempi di alcune strutture dati che possono essere trasformate in un numero tramite la funzione coppia di Cantor. Vedremo come ogni struttura dati verrà manipolata e trasformata in una coppia (x, y) per poterla applicare alla funzione coppia.

5.1.2.1. Strutture dati

5.1.2.1.1. Liste

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

Codifichiamo la lista $[x_1, ..., x_n]$ con

$$< x_1, ..., x_n >=< x_1, < x_2, < ... < x_n, 0 > ... \ggg .$$

Abbiamo quindi applicato la funzione coppia di Cantor alla coppia formata da un elemento della lista e il risultato della funzione coppia stessa applicata ai successivi elementi.

Ad esempio, la codifica della lista $M = [1, 2, 5]$ risulta essere:

$$
\begin{aligned}
< 1, 2, 5 > &= < 1, < 2, < 5, 0 \gg \gg \\
&= < 1, < 2, 16 \gg \\
&= < 1, 188 > \\
&= 18144.
\end{aligned}
$$

Per decodificare la lista M applichiamo le funzioni *sin* e *des* al risultato precedente. Alla prima iterazione otterremo il primo elemento della lista e la restante parte ancora da decodificare.

Quando ci fermiamo? Durante la creazione della codifica di M abbiamo inserito un *"tappo"*, ovvero la prima iterazione della funzione coppia $< x_n, 0 >$. Questo ci indica che quando des (M) sarà uguale a 0 ci dovremo fermare.

Cosa succede se abbiamo uno 0 nella lista? Non ci sono problemi: il controllo avviene sulla funzione *des*, che restituisce la *"somma parziale"* e non sulla funzione *sin*, che restituisce i valori della lista.

Possiamo anche anche delle implementazioni di queste funzioni. Assumiamo che:
- 0 codifichi la lista nulla;
- esistano delle routine per $<, >$, sin e des.

Codifica

```
def encode(numbers: list[int]) -> int:
  k = 0
  for i in range(n,0,-1):
    xi = numbers[i]
    k = <xi,k>
  return k
```

Decodifica

```
def decode(n: int) -> list[int]:
  numbers = []
  while True:
    left, n = sin(n), des(n)
    numbers.append(left)
    if n == 0:
      break
  return numbers
```

Un metodo molto utile delle liste è quello che ritorna la sua **lunghezza**.

Lunghezza

```
def length(n: int) -> int:
  return 0 if n == 0 else 1 + length(des(n))
```

Infine, definiamo la funzione **proiezione** come:

$$
\text{proj}(t, n) = \begin{cases} -1 & \text{se } t > \text{length}(n) \vee t \leq 0 \\ x_t & \text{altrimenti} \end{cases}
$$

e la sua implementazione:

Proiezione

```
def proj(t: int, n: int) -> int:
  if t <= 0 or t > length(n):
    return -1
  else:
    if t == 1:
      return sin(n)
    else:
      return proj(t - 1, des(n))
```

5.1.2.1.2. Array

Per gli **array** il discorso è più semplice, in quanto la dimensione è nota a priori. Di conseguenza, non necessitiamo di un carattere di fine sequenza. Dunque avremo che l'array $\{x_1, ..., x_n\}$ viene codificato con:

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

5.1.2.1.3. Matrici

Per quanto riguarda **matrici** l'approccio utilizzato codifica singolarmente le righe e successivamente codifica i risultati ottenuti come se fossero un array di dimensione uguale al numero di righe.

Ad esempio, la matrice

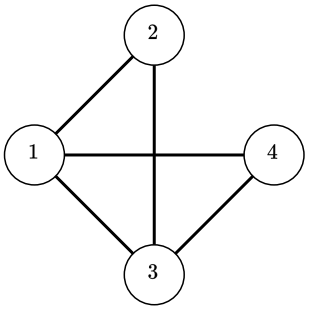
$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

viene codificata in:

$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} = [[x_{11}, x_{12}, x_{13}], [x_{21}, x_{22}, x_{23}], [x_{31}, x_{32}, x_{33}]].$$

5.1.2.1.4. Grafi

Consideriamo il seguente grafo.



I **grafi** sono rappresentati principalmente in due modi:

- **liste di adiacenza dei vertici**: per ogni vertice si ha una lista che contiene gli identificatori dei vertici che collegati direttamente con esso. Il grafo precedente ha

  $$\{1 : [2, 3, 4], 2 : [1, 3], 3 : [1, 2, 4], 4 : [1, 3]\}$$

  come lista di adiacenza, e la sua codifica si calcola come:

  $$[< 2, 3, 4 >, < 1, 2 >, < 1, 2, 4 >, < 1, 3 >].$$

  Questa soluzione esegue prima la codifica di ogni lista di adiacenza e poi la codifica dei risultati del passo precedente.
- **matrice di adiacenza**: per ogni cella m_{ij} della matrice $|V| \times |V|$, dove V è l'insieme dei vertici, si ha:
  - 1 se esiste un arco dal vertice i al vertice j;
  - 0 altrimenti;

  Essendo questa una matrice la andiamo a codificare come abbiamo già descritto.

5.1.2.2. Applicazioni

Una volta visto come rappresentare le principali strutture dati, è facile trovare delle vie per codificare qualsiasi tipo di dato in un numero. Vediamo alcuni esempi.

5.1.2.2.1. Testi

Dato un **testo**, possiamo ottenere la sua codifica nel seguente modo:

1. trasformiamo il testo in una lista di numeri tramite la codifica ASCII dei singoli caratteri;
2. sfruttiamo l'idea dietro la codifica delle liste per codificare quanto ottenuto.

Per esempio,

$$\text{ciao} = [99, 105, 97, 111] =< 99, < 105, < 97, < 111, 0 \ggg .$$

Possiamo chiederci:

- *Il codificatore proposto è un buon compressore?*
  No, si vede facilmente che il numero ottenuto tramite la funzione coppia (o la sua concatenazione) sia generalmente molto grande, e che i bit necessari a rappresentarlo crescano esponenzialmente sulla lunghezza dell'input. Ne segue che questo è un *pessimo* modo per comprimere messaggi.
- *Il codificatore proposto è un buon sistema crittografico?*
  La natura stessa del processo garantisce la possibilità di trovare un modo per decifrare in modo analitico, di conseguenza chiunque potrebbe, in poco tempo, decifrare il mio messaggio. Inoltre, questo metodo è molto sensibile agli errori.

5.1.2.2.2. Suoni

Dato un **suono**, possiamo *campionare* il suo segnale elettrico a intervalli di tempo regolari e codificare la sequenza dei valori campionati tramite liste o array.

5.1.2.2.3. Immagini

Per codificare le **immagini** esistono diverse tecniche, ma la più usata è la **bitmap**: ogni pixel contiene la codifica numerica di un colore, quindi posso codificare separatamente ogni pixel e poi codificare i singoli risultati insieme tramite liste o array.

5.1.3. Conclusioni

Abbiamo mostrato come i dati possano essere *"buttati via"* in favore delle codifiche numeriche associate ad essi.

Di conseguenza, possiamo sostituire tutte le funzioni $f : \text{DATI} \longrightarrow \text{DATI}_\perp$ con delle funzioni $f' : \mathbb{N} \to \mathbb{N}_\perp$. In altre parole, l'universo dei problemi per i quali cerchiamo una soluzione automatica è rappresentabile dall'insieme $\mathbb{N}_\perp^\mathbb{N}$.

6. Lezione 06

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

La relazione interviene nella parte che afferma che

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

In poche parole, la potenza computazionale, cioè l'insieme dei programmi che $\mathcal{C}$ riesce a calcolare, è isomorfa all'insieme di tutti i programmi, a loro volta isomorfi a $\mathbb{N}$.

Per dimostrare l'ultima parte di questa catena di relazione dobbiamo esibire una legge che mi permetta di ricavare un numero dato un programma e viceversa.

Per fare questo vediamo l'insieme PROG come l'insieme dei programmi scritti in un certo linguaggio di programmazione. Analizzeremo due sistemi diversi:
- Sistemi di calcolo RAM;
- Sistemi di calcolo while.

In generale, ogni sistema di calcolo ha la propria macchina e il proprio linguaggio.

6.1.1. Sistema di calcolo RAM

6.1.1.1. Introduzione

Questo sistema è molto semplice e ci permette di definire rigorosamente:
- $\text{PROG} \sim \mathbb{N}$;
- la **semantica dei programmi eseguibili**, ovvero calcolo $\mathcal{C}(P, \_)$ con $\mathcal{C} = \text{RAM}$ ottenendo $\text{RAM}(P, \_)$;
- la **potenza computazionale**, ovvero calcolo $F(\mathcal{C})$ con $\mathcal{C} = \text{RAM}$ ottenendo $F(\text{RAM})$.

Il linguaggio utilizzato è un assembly molto semplificato, immediato e semplice.

Dopo aver definito $F(\text{RAM})$ potremmo chiederci se questa definizione sia troppo stringente e riduttiva per definire tutti i sistemi di calcolo. In futuro introdurremo delle macchine più sofisticate, dette **macchine while**, che, a differenza delle macchine RAM, sono *strutturate*. Infine, confronteremo $F(\text{RAM})$ e $F(\text{WHILE})$. I due risultati possibili sono:
- le potenze computazionali sono *diverse*: ciò che è computazionale dipende dallo strumento, cioè dal linguaggio utilizzato;
- le potenze computazionali sono *uguali*: la computabilità è intrinseca dei problemi, non dello strumento.

Il secondo è il caso più promettente e, in quel caso, cercheremo di trovare una caratterizzazione teorica, ovvero di *"recintare"* tutti i problemi calcolabili.

6.1.1.2. Macchina RAM

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

Indichiamo i registri con R_k, con $k \geq 0$. Tra questi ci sono due registri particolari:
- R_0 contiene l'*output*;
- R_1 contiene l'*input*.

Un altro registro molto importante, che non rientra nei registri R_k, è il registro L, detto anche **program counter** (*PC*). Questo registro è essenziale in questa architettura, in quanto indica l'indirizzo della prossima istruzione da eseguire.

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

Le istruzioni nel linguaggio RAM sono:
- **incremento**: $R_k \longleftarrow R_k + 1$;
- **decremento**: $R_k \longleftarrow R_k \dot{-} 1$;
- **salto condizionato**: IF $R_k = 0$ THEN GOTO m, con $m \in \{1, ..., |P|\}$.

L'istruzione di decremento é tale che

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

6.1.1.3. Fasi dell'esecuzione

L'esecuzione di un programma su una macchina RAM segue i seguenti passi:
1. **Inizializzazione**:
   1. viene caricato il programma $P \equiv \text{Istr}_1, ..., \text{Istr}_n$ in memoria;
   2. il PC viene posto a 1 per indicare di eseguire la prima istruzione del programma;
   3. nel registro R_1 viene caricato l'input;
   4. ogni altro registro è azzerato.
2. **Esecuzione**: si eseguono tutte le istruzioni *una dopo l'altra*, ovvero ad ogni iterazione passo da L a $L + 1$, a meno di istruzioni di salto. Essendo il linguaggio RAM *non strutturato* il PC è necessario per indicare ogni volta l'istruzione da eseguire al passo successivo. Un linguaggio strutturato, invece, sa sempre quale istruzione eseguire dopo quella corrente, infatti non è dotato di PC;
3. **Terminazione**: per convenzione si mette $L = 0$ per indicare che l'esecuzione del programma è finita oppure è andata in loop. Questo segnale, nel caso il programma termini, è detto **segnale di halt** e arresta la macchina;
4. **Output**: il contenuto di R_0, se vado in halt, contiene il risultato dell'esecuzione del programma P. Indichiamo con $\varphi_P(n)$ il contenuto del registro R_0 (in caso di halt) oppure $\perp$ (in caso di loop).

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

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

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

6.1.1.4. Definizione formale semantica

Vogliamo dare una definizione formale della semantica di un programma RAM. Quello che faremo sarà dare una **semantica operazionale** alle istruzioni RAM, ovvero specificare il significato di ogni istruzione esplicitando l'**effetto** che quell'istruzione ha sui registri della macchina.

Per descrivere l'effetto di un'istruzione ci serviamo di una *foto*. L'idea che ci sta dietro è:
1. faccio una foto della macchina *prima* dell'esecuzione dell'istruzione;
2. eseguo l'istruzione;
3. faccio una foto della macchina *dopo* l'esecuzione dell'istruzione.

La foto della macchina si chiama **stato** e deve descrivere completamente la situazione della macchina in un certo istante. La coppia $(\text{StatoPrima}, \text{StatoDopo})$ rappresenta la semantica operazionale di una data istruzione del linguaggio RAM.

L'unica informazione da salvare dentro una foto è la situazione globale dei registri R_k e il registro L. Il programma non serve, visto che rimane sempre uguale.

La **computazione** del programma P è una sequenza di stati S_i, ognuno generato dall'esecuzione di un'istruzione del programma. Diciamo che P induce una sequenza di stati S_i. Se quest'ultima 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. In poche parole:

$$\varphi_P : \mathbb{N} \longrightarrow \mathbb{N}_\perp \text{ tale che } \varphi_P(n) = \begin{cases} y & \text{se } \exists S_{\text{finale}} \\ \perp & \text{altrimenti} \end{cases} .$$

Definiamo ora come passiamo da uno stato all'altro. Per far ciò, definiamo:

- **stato**: istantanea di tutte le componenti della macchina, è una funzione

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

  tale che $S(R_k)$ restituisce il contenuto del registro R_k quando la macchina si trova nello stato S. Gli stati possibili di una macchina appartengono all'insieme

  $$\text{STATI} = \{f : \{L, R_i\} \longrightarrow \mathbb{N}\} = \mathbb{N}^{\{L, R_i\}}.$$

  Questa rappresentazione è molto comoda perché ho potenzialmente un numero di registri infinito. Se così non fosse avrei delle tuple per indicare tutti i possibili registri al posto dell'insieme $\{L, R_i\}$;
- **stato finale**: uno stato finale è un qualsiasi stato S tale che $S(L) = 0$;
- **dati**: abbiamo già dimostrato come $\text{DATI} \sim \mathbb{N}$;
- **inizializzazione**: serve una funzione che, preso l'input, ci dia lo stato iniziale della macchina. La funzione è

  $$\text{in} : \mathbb{N} \longrightarrow \text{STATI} \text{ tale che } \text{in}(n) = S_{\text{iniziale}}.$$

  Lo stato S_{iniziale} è tale che

  $$S_{\text{iniziale}}(R) = \begin{cases} 1 & \text{se } R = L \\ n & \text{se } R = R_1 \\ 0 & \text{altrimenti} \end{cases} ;$$
- **programmi**: definisco PROG come l'insieme dei programmi RAM.

Ci manca da definire la *parte dinamica* del programma, ovvero l'**esecuzione**. Definiamo la **funzione di stato prossimo**

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

tale che

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

dove S rappresenta lo stato attuale e S' rappresenta lo stato prossimo dopo l'esecuzione di un'istruzione di P.

La funzione $\delta(S, P) = S'$ è tale che:

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

  $$S'(R) = \begin{cases} 0 & \text{se } R = L \\ S(R_i) & \text{se } R = R_i \ \forall i \end{cases} ;$$
- se $1 \leq S(L) \leq |P|$ considero l'istruzione $S(L)$-esima:

- se ho incremento/decremento sul registro R_k definisco S' tale che

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

- se ho il GOTO sul registro R_k che salta all'indirizzo m definisco S' tale che

$$S'(L) = \begin{cases} m & \text{se } S(R_k) = 0 \\ S(L) + 1 & \text{altrimenti} \end{cases},$$
$$S'(R_i) = S(R_i) \quad \forall i.$$

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

$$S_0, S_1, ..., S_i, S_{i+1}, ...$$

tali che

$$S_0 = \text{in}(n)$$
$$\forall i \quad S_{i+1} = \delta(S_i, P).$$

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

La semantica di P è

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

La potenza computazionale del sistema RAM è:

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

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

7. Lezione 07

7.1. Aritmetizzazione di un programma

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

Quello che dobbiamo fare è trovare una funzione che trasforma le istruzioni in numeri, così da avere poi accesso alla funzione coppia per fare la codifica effettiva. In generale, il processo che associa biunivocamente un numero ad una struttura viene chiamato **aritmetizzazione** o **godelizzazione**.

Troviamo una funzione Ar che associ ad ogni istruzione I_k la sua codifica numerica c_k. Se la funzione trovata è anche biunivoca siamo sicuri di trovare la sua inversa, ovvero quella funzione che ci permette di ricavare l'istruzione I_k data la sua codifica c_k.

Riassumendo quanto detto finora, abbiamo deciso di trasformare ogni lista di istruzioni in una lista di numeri e, successivamente, applicare la funzione coppia di Cantor, ovvero

$$[\text{Istr}_1, ..., \text{Istr}_n] \overset{\text{Ar}}{\rightsquigarrow} [c_1, ..., c_n] \overset{<>}{\rightsquigarrow} n.$$

Vorremmo anche ottenere la lista di istruzioni originale data la sua codifica, ovvero

$$n \overset{<>}{\rightsquigarrow} [c_1, ..., c_n] \overset{\text{Ar}^{-1}}{\rightsquigarrow} [\text{Istr}_1, ..., \text{Istr}_n].$$

La nostra funzione *"complessiva"* è biunivoca se dimostriamo la biunivocità della funzione Ar, avendo già dimostrato questa proprietà per $<>$.

Dobbiamo quindi trovare una funzione biunivoca $\text{Ar} : \text{ISTR} \longrightarrow \mathbb{N}$ con la sua funzione inversa $\text{Ar}^{-1} : \mathbb{N} \longrightarrow \text{ISTR}$ tali che

$$\text{Ar}(I) = n \iff \text{Ar}^{-1}(n) = I.$$

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

$$\text{Ar}(I) = \begin{cases} 3k & \text{se } I \equiv R_k \longleftarrow R_k + 1 \\ 3k+1 & \text{se } I \equiv R_k \longleftarrow R_k \dot{-} 1 \\ 3 < k, m > -1 & \text{se } I \equiv \text{IF } R_k = 0 \text{ THEN GOTO } m \end{cases}.$$

Come è fatta l'inversa Ar^{-1}? In base al modulo tra n e 3 ottengo una certa istruzione:

$$\text{Ar}^{-1}(n) = \begin{cases} R_{\frac{n}{3}} \longleftarrow R_{\frac{n}{3}} + 1 & \text{se } n \bmod 3 = 0 \\ R_{\frac{n-1}{3}} \longleftarrow R_{\frac{n-1}{3}} \dot{-} 1 & \text{se } n \bmod 3 = 1 \\ \text{IF } R_{\text{sin }(\frac{n+1}{3})} = 0 \text{ THEN GOTO des } \left(\frac{n+1}{3}\right) & \text{se } n \bmod 3 = 2 \end{cases}.$$

La codifica del programma P è quindi

$$\text{cod}(P) =< \text{Ar}(\text{Istr}_1), ..., \text{Ar}(\text{Istr}_n) > .$$

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 $\text{length}(\text{cod}(P))$.

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

7.2. Osservazioni

Vediamo una serie di osservazioni importanti:

- avendo $n = \text{cod}(P)$ 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*;
- posso 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 $\text{PROG} \sim \mathbb{N}$, è numerabile;
- ho dimostrato rigorosamente che

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

  quindi anche nel sistema di calcolo RAM esistono funzioni non calcolabili;
- la RAM è troppo elementare affinche $F(\text{RAM})$ 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 possiamo *"catturarla"* matematicamente.

7.3. Sistema di calcolo WHILE

7.3.1. Macchina WHILE

La macchina WHILE, come quella RAM, è molto semplice, essendo formata da una serie di **registri**, detti **variabili**. Al contrario delle macchine RAM, questi ultimi non sono *potenzialmente infiniti*, ma sono esattamente 21. Il registro R_0 è il **registro di output**, mentre R_1 è il **registro di input**. Inoltre, non esiste il registro del program counter in quanto il linguaggio è *strutturato* e ogni istruzione di questo linguaggio va eseguita in ordine.

7.3.2. Linguaggio WHILE

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

Il comando di base è l'**assegnamento**. In questo linguaggio ne esistono di tre tipi:

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

Vediamo come queste istruzioni siamo molto più complete rispetto alle istruzioni RAM

$$R_k \longleftarrow R_k + 1$$
$$R_k \longleftarrow R_k \dot{-} 1$$

in quanto con una sola istruzione possiamo azzerare il valore di una variabile o assegnare ad una variabile il valore di un'altra aumentata/diminuita di 1.

I comandi "induttivi" sono invece il comando while e il comando composto.

Il **comando while** è un comando nella forma

$$\text{while } x_k \neq 0 \text{ do } C,$$

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

Il **comando composto** è un comando nella forma

$$\text{begin } C_1; ...; C_n \text{ end},$$

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

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 $W \in W$-PROG.

Per dimostrare una proprietà P di un programma $W \in W$-PROG procederemo 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 $C_1, ..., C_n$ e la dimostro vera per begin $C_1; ...; C_n$ end.

8. Lezione 08

8.1. Semantica 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, perché le istruzioni di W vengono eseguite una dopo l'altra;
3. **terminazione**: l'esecuzione di W può:
   - *arrestarsi*, se sono arrivato al termine delle istruzioni;
   - *non arrestarsi*, se si è entrati in un loop;
4. **output**: se il programma va in `halt`, l'output è contenuto nel registro x_0. Possiamo scrivere

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

La funzione $\Psi_W : \mathbb{N} \longrightarrow \mathbb{N}_\perp$ indica la semantica del programma W.

Diamo ora la definizione formale della semantica di un programma WHILE. Come per i programmi RAM, abbiamo bisogno di una serie di elementi:

1. **stato**: usiamo una tupla grande quanto il numero di variabili, quindi $\underline{x} = (c_0, ..., c_{20})$ rappresenta il nostro stato, con c_i contenuto della variabile i;
2. $\boldsymbol{W}$**-STATI**: insieme di tutti gli stati possibili, che è in $\mathbb{N}^{21}$, vista la definizione degli stati;
3. **dati**: sappiamo già che DATI $\sim \mathbb{N}$;
4. **inizializzazione**: lo stato iniziale è descritto dalla seguente funzione

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

5. **semantica operazionale**: vogliamo trovare una funzione che, presi il comando da eseguire e lo stato corrente, restituisce lo stato prossimo.

Soffermiamoci sull'ultimo punto. Vogliamo trovare la funzione

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

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.

Partiamo dal passo base, quindi dagli **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, quindi:

- **comando composto**: vogliamo calcolare

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

  conoscendo ogni $[\![C_i]\!]$ per ipotesi induttiva. Calcoliamo allora la funzione:

$$[\![C_n]\!](...([\![C_2]\!]([\![C_1]\!](\underline{x})))...) = ([\![C_n]\!] \circ ... \circ [\![C_1]\!])(\underline{x}),$$

ovvero applichiamo in ordine i comandi C_i presenti nel comando composto C.

- **comando while**: vogliamo calcolare

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

conoscendo ogni $[\![C_i]\!]$ per ipotesi induttiva. Calcoliamo allora la funzione:

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

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

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

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. Quest'ultima è la funzione

$$\Psi_W : \mathbb{N} \longrightarrow \mathbb{N}_\perp \quad | \quad \Psi_W(n) = \text{Proj}(0, [\![W]\!](\text{w-in}(n))).$$

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

La **potenza computazionale** del sistema di calcolo WHILE è l'insieme

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

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

8.2. Confronto macchina RAM e macchina WHILE

Viene naturale andare a confrontare i due sistemi di calcolo descritti, cercando di capire quale è "più potente" dell'altro, sempre che ce ne sia uno.

Ci sono quattro possibili situazioni:

- $F(\text{RAM}) \subsetneq F(\text{WHILE})$, che sarebbe anche comprensibile vista l'estrema semplicità del sistema RAM;
- $F(\text{RAM}) \cap F(\text{WHILE})$ vuota (*insiemi disgiunti*) o abbia elementi (*insiemi sghembi*). Questo scenario sarebbe preoccupante, perché il concetto di calcolabile dipenderebbe dalla macchina che si sta utilizzando;
- $F(\text{WHILE}) \subseteq F(\text{RAM})$, che sarebbe sorprendente dato che il sistema WHILE sembra più sofisticato del sistema RAM, ma la relazione decreterebbe che il sistema WHILE non è più potente del sistema RAM;
- $F(\text{WHILE}) = F(\text{RAM})$, sarebbe il risultato migliore, perché il concetto di *calcolabile* non dipenderebbe dalla tecnologia utilizzata, ma sarebbe intrinseco nei problemi.

Poniamo di avere C_1 e C_2 sistemi di calcolo con programmi in C_1-PROG e C_2-PROG e potenze computazionali

$$F(C_1) = \left\{ f : \mathbb{N} \longrightarrow \mathbb{N}_\perp \mid \exists P_1 \in C_1\text{-PROG} \mid f = \Psi_{P_1} \right\} = \left\{ \Psi_{P_1} : P_1 \in C_1\text{-PROG} \right\},$$

$$F(C_2) = \left\{ f : \mathbb{N} \longrightarrow \mathbb{N}_\perp \mid \exists P_2 \in C_2\text{-PROG} \mid f = \Psi_{P_2} \right\} = \left\{ \Psi_{P_2} : P_2 \in C_2\text{-PROG} \right\}.$$

Come mostro che $F(C_1) \subseteq F(C_2)$, ovvero che il primo sistema di calcolo non è più potente del secondo? Devo dimostrare che ogni elemento nel primo insieme deve stare anche nel secondo

$$\forall f \in F(C_1) \Longrightarrow f \in F(C_2).$$

Se *espandiamo* la definizione di $f \in F(C)$ allora la relazione diventa:

$$\exists P_1 \in C_1\text{-PROG} \mid f = \Psi_{P_1} \Longrightarrow \exists P_2 \in C_2\text{-PROG} \mid f = \Psi_{P_2}.$$

In poche parole, per ogni programma calcolabile nel primo sistema di calcolo ne esiste uno con la stessa semantica nel secondo sistema. Quello che vogliamo trovare è un **compilatore**, ovvero una funzione che trasformi un programma del primo sistema in un programma del secondo sistema. Useremo il termine **traduttore** al posto di *compilatore*.

8.3. Traduzioni

Dati C_1 e C_2 due sistemi di calcolo, definiamo **traduzione** da C_1 a C_2 una funzione

$$T : C_1\text{-PROG} \longrightarrow C_2\text{-PROG}$$

con le seguenti proprietà:

- **programmabile** $\rightarrow$ esiste un modo per programmarla;
- **completa** $\rightarrow$ sappia tradurre **ogni** programma in C_1-PROG in un programma in C_2-PROG;
- **corretta** $\rightarrow$ mantiene la semantica del programma di partenza

$$\forall P \in C_1\text{-PROG} \quad : \quad \Psi_P = \varphi_{T(P)},$$

dove Ψ rappresenta la semantica dei programmi in C_1-PROG e φ rappresenta la semantica dei programmi in C_2-PROG.

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

Dimostrazione
Se $f \in F(C_1)$ allora esiste un programma $P_1 \in C_1$-PROG tale che $\Psi_{P_1} = f$.

A questo programma P_1 applico T, ottenendo $T(P_1) = P_2 \in C_2$-PROG (per *completezza*) tale che $\varphi_{P_2} = \Psi_{P_1} = f$ (per *correttezza*).

Ho trovato un programma $P_2 \in C_2$-PROG la cui semantica è f, allora $F(C_1) \subseteq F(C_2)$. □

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

$$\text{Comp} : W\text{-PROG} \longrightarrow \text{PROG}.$$

9. Lezione 09

9.1. Introduzione

Vogliamo dimostrare che

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

ovvero che ogni funzione programmabile in WHILE lo è anche in RAM.

Dobbiamo trovare un compilatore

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

che rispetti le caratteristiche di programmabilità, completezza e correttezza viste per i traduttori.

9.2. Compilatore per il linguaggio WHILE

Per comodità andiamo ad usare un linguaggio RAM **etichettato**: esso aggiunge la possibilità di etichettare un'istruzione che indica un punto di salto o di arrivo. In altre parole, le etichette rimpiazzano gli indirizzi di salto, che erano indicati con un numero di istruzione.

Questa aggiunta non aumenta la potenza espressiva del linguaggio, essendo pura sintassi: il RAM etichettato si traduce facilmente nel RAM *puro*.

9.2.1. Forma del compilatore

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

- **passo base**: mostro come compilare gli assegnamenti;
- **passo induttivo**:
  1. per ipotesi induttiva, assumo di sapere $\text{Comp}(C_1), ..., \text{Comp}(C_m)$ e mostro come compilare il comando composto begin $C_1; ...; C_n$ end;
  2. per ipotesi induttiva, assumo di sapere $\text{Comp}(C)$ e mostro come compilare il comando while while $x_k \neq 0$ do C.

Nelle traduzioni andremo a mappare la variabile WHILE x_k nel registro RAM R_k. Questo non mi crea problemi o conflitti, perché sto mappando un numero finito di registri (21) in un insieme infinito.

9.2.1.1. Assegnamento

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

$$\begin{aligned}\text{Comp}(x_k := 0) = \text{LOOP} : &\text{IF } R_k = 0 \text{ THEN GOTO EXIT} \\ &R_k \longleftarrow R_k \dot{-} 1 \\ &\text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \\ \text{EXIT} : &R_K \longleftarrow R_K \dot{-} 1 \quad .\end{aligned}$$

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 su 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 immediata e banale e l'istruzione RAM è

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

Se invece $k \neq j$ la prima idea che viene in mente è quella di "migrare" 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 mi genera R_j dentro R_k. Possiamo 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 di quella istruzione: infatti, io 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.

Ricapitolando:

1. salviamo x_j in R_{22}, registro *sicuro* perché mai coinvolto in altre istruzioni;
2. azzeriamo R_k;
3. rigeneriamo R_j e settiamo R_k da R_{22};
4. ± 1 in R_k.

$$
\begin{aligned}
\text{Comp}(x_k := x_j \pm 1) = \text{LOOP} &: \text{IF } R_j = 0 \text{ THEN GOTO EXIT1} \\
&\quad R_j \longleftarrow R_j \dot{-} 1 \\
&\quad R_{22} \longleftarrow R_{22} + 1 \\
&\quad \text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \\
\text{EXIT1} &: \text{IF } R_k = 0 \text{ THEN GOTO EXIT2} \\
&\quad R_k \longleftarrow R_k \dot{-} 1 \\
&\quad \text{IF } R_{21} = 0 \text{ THEN GOTO EXIT1} \\
\text{EXIT2} &: \text{IF } R_{22} = 0 \text{ THEN GOTO EXIT3} \\
&\quad R_k \longleftarrow R_k + 1 \\
&\quad R_j \longleftarrow R_j + 1 \\
&\quad R_{22} \longleftarrow R_{22} \dot{-} 1 \\
&\quad \text{IF } R_{21} = 0 \text{ THEN GOTO EXIT2} \\
\text{EXIT3} &: R_k \longleftarrow R_k \pm 1 \quad .
\end{aligned}
$$

9.2.1.2. Comando composto

Per ipotesi induttiva, sappiamo come compilare $C_1, ..., C_m$. Possiamo calcolare la compilazione del comando composto come

$$
\begin{aligned}
\text{Comp}(\text{begin } C_1; ...; C_n \text{ end}) = &\text{Comp}(C_1) \\
&... \\
&\text{Comp}(C_m) \quad .
\end{aligned}
$$

9.2.1.3. Comando while

Per ipotesi induttiva, sappiamo come compilare C. Possiamo calcolare la compilazione del comando while come

$$
\begin{aligned}
\text{Comp}(\text{while } x_k \neq 0 \text{ do } C) = \text{LOOP} &: \text{IF } R_k = 0 \text{ THEN GOTO EXIT} \\
&\quad \text{Comp}(C) \\
&\quad \text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \\
\text{EXIT} &: R_k \longleftarrow R_k \dot{-} 1 \quad .
\end{aligned}
$$

9.2.2. Risultati ottenuti

La funzione

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

che abbiamo costruito soddisfa le tre proprietà che desideravamo, quindi

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

Questa inclusione appena dimostrata è propria?

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

In questa sezione dimostriamo come

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

Allo stesso modo di prima, mostreremo l'esistenza di un compilatore *"inverso"* che trasformi sorgenti RAM in sorgenti WHILE.

9.3.1. Interprete

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 $\varphi_P(x)$.

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

Abbiamo due problemi principali:

1. Il primo riguarda il tipo di input della macchina WHILE: 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, piuttosto la sua codifica $\text{cod}(P) = n \in \mathbb{N}$. Questo mi 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 di input della macchina WHILE: quest'ultima legge l'input da un singolo registro, mentre qui ne stiamo passando due. Dobbiamo modificare I_W condensando l'input con la funzione coppia di Cantor, che diventa $< x, n >$.

La semantica di I_W diventa

$$\forall x, n \in \mathbb{N} \quad \Psi_{I_W}(< x, n >) = \varphi_n(x) = \varphi_P(x).$$

9.3.1.1. **Macro**-WHILE

Come prima, per comodità di scrittura useremo un altro linguaggio, il **macro**-WHILE. Questo include alcune macro che saranno molto comode nella scrittura di I_W. Visto che viene modificata solo la sintassi, la potenza del linguaggio non WHILE non aumenta.

Le macro utilizzate sono:

- $x_k := x_j + x_s$;
- $x_k :=< x_j, x_s >$;
- $x_k :=< x_1, ..., x_n >$;
- proiezione $x_k := \text{Proj}(x_j, x_s) \rightarrow$ estrae l'elemento x_j-esimo dalla lista codificata in x_s;
- incremento $x_k := \text{incr}(x_j, x_s) \rightarrow$ codifica la lista x_s con l'elemento in posizione x_j-esima aumentato di uno;
- decremento $x_k := \text{decr}(x_j, x_s) \rightarrow$ codifica la lista x_s con l'elemento in posizione x_j-esima diminuito di uno;
- $x_k := \text{sin}\ (x_j)$;
- $x_k := \text{des}\ (x_j)$;

- costrutto if ... then ... else.

10. Lezione 10

10.1. Interprete WHILE di programmi RAM

Risolto il problema dell'input di un interprete scritto in linguaggio WHILE per i programmi RAM, ora vogliamo scrivere questo interprete, ricordando che per comodità non useremo il WHILE puro ma il macro-WHILE.

Cosa fa l'interprete? In poche parole, esegue una dopo l'altra le istruzioni RAM del programma P e restituisce il risultato $\varphi_P(x)$. Notiamo come restituiamo un risultato, non un eseguibile.

Infatti, quello che fa l'interprete è ricostruire virtualmente tutto ciò che gli serve per gestire il programma. Nel nostro caso, 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 $R_0, R_1, R_2, ...$ dentro le variabili messe a disposizione dalla macchina WHILE.

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

La risposta è no. Infatti, se $\text{cod}(P) = n$ allora P non utilizza mai dei registri R_j, con $j > n$. Se uso un registro di indice $n + 1$ non posso avere codifica n perché l'istruzione che usa $n + 1$ ha come codifica i numeri $3(n + 1)$ se incremento, $3(n + 1) + 1$ se decremento oppure il numero generato da Cantor se GOTO. Inoltre, le singole istruzioni codificate vanno codificate tramite lista di Cantor, e abbiamo mostrato come questa funzione cresca molto rapidamente.

Di conseguenza, possiamo restringerci a modellare i registri $R_0, ..., R_{n+2}$. Usiamo i registri fino a a $n + 2$ solo per avere un paio di registri in più che potrebbero tornare utili. Ciò ci permette di codificare la memoria utilizzata dal programma P tramite la funzione di Cantor.

Vediamo, nel dettaglio, l'interno di $I_w(< x, n >) = \varphi_n(x)$:
- x_0 contiene $< R_0, ..., R_{n+2} >$, lo stato della memoria della macchina RAM;
- x_1 contiene L, il program counter;
- x_2 contiene x, il dato su cui lavora P;
- x_3 contiene n, il "listato" del programma P;
- x_4 contiene il codice dell'istruzione da eseguire, prelevata da x_3 grazie a x_1.

Ricordiamo che all'avvio l'interprete I_w trova il suo input nella variabile di input x_1.

10.1.1. Codice dell'interprete I_w

```
# Inizializzazione
input(<x,n>);                          # In x1
x2 := sin(x1);                         # Dato x
x3 := des(x1);                         # Programma n
x0 := <0,x2,0,...,0>;                  # Memoria iniziale
x1 := 1;                               # Program counter

# Esecuzione
while (x1 != 0) do:                    # Se x1 = 0 HALT
  if (x1 > length(x2)) then            # Sono fuori dal programma?
    x1 := 0;                           # Vado in HALT
  else
    x4 := Pro(x1, x3);                 # Fetch istruzione
    if (x4 mod 3 == 0) then            # Incremento?
      x5 := x4 / 3;                    # Trovo k
      x0 := incr(x5, x0);              # Incremento
      x1 := x1 + 1;                    # Istruzione successiva
    if (x4 mod 3 == 1) then            # Decremento?
      x5 := (x4 - 1) / 3;              # Trovo k
      x0 := decr(x5, x0);              # Decremento
      x1 := x1 + 1;                    # Istruzione successiva
    if (x4 mod 3 == 2) then            # GOTO?
      x5 := sin((x4 + 1) / 3);         # Trovo k
      x6 := des((x4 + 1) / 3);         # Trovo m
      if (Proj(x5, x0) == 0) then      # Devo saltare?
        x1 := x4;                      # Salto a m
      else
        x1 := x1 + 1;                  # Istruzione successiva

# Finalizzazione
x0 := sin(x0);                         # Oppure Pro(0,x0)
```

10.1.2. Conseguenza dell'esistenza di I_w

Avendo in mano l'interprete I_W, possiamo costruire un compilatore

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

tale che

$$\begin{aligned} \text{Comp}(P \in \text{PROG}) \equiv n &\longleftarrow \text{cod}(P) \\ x_1 &:=< x_1, n > \\ I_W &\quad . \end{aligned}$$

Questo significa che il compilatore non fa altro che cablare all'input x il programma RAM da interpretare e procede con l'esecuzione dell'interprete.

Vediamo se le tre proprietà di un compilatore sono soddisfatte:

- **programmabile**: sì, lo abbiamo appena fatto;
- **completo**: l'interprete riesce a riconoscere ogni istruzione RAM e la riesce a codificare;
- **corretto**: vale $P \in \text{PROG} \longmapsto \text{Comp}(P) \in W\text{-PROG}$, quindi:

$$\Psi_{\text{Comp}(P)}(x) = \Psi_{I_W}(< x, n >) = \varphi_n(x) = \varphi_P(x)$$

rappresenta la sua semantica.

Abbiamo dimostrato quindi che

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

che è l'inclusione opposta del precedente risultato.

Il risultato appena ottenuto ci permette di definire un teorema molto importante.

Teorema *(Teorema di Böhm-Jacopini del 1970)* Per ogni programma con GOTO (RAM) ne esiste uno equivalente in un linguaggio strutturato (WHILE).

Questo teorema è fondamentale perché lega la programmazione a basso livello con quella ad alto livello. In poche parole, il GOTO può essere eliminato e la programmazione a basso livello può essere sostituita da quella ad alto livello.

10.1.3. Altre conseguenze e osservazioni

Grazie alle due inclusioni dimostrate in precedenza abbiamo dimostrato anche 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, seppur profondamente diversi, i sistemi RAM e WHILE calcolano le stesse cose. Viene naturale chiedersi quindi quale sia la vera natura della calcolabilità.

Un altro risultato che abbiamo dimostrato *formalmente* è che nei sistemi di programmazione RAM e WHILE esistono funzioni **non calcolabili**, che sono nella *parte destra* della catena scritta sopra.

10.1.4. Compilatore universale

Facciamo una mossa esotica: usiamo il compilatore da WHILE a RAM $\text{Comp} : W\text{-PROG} \to \text{PROG}$ sul programma I_w. Lo possiamo fare? Certo, posso compilare I_W perché è un programma WHILE.

Chiamiamo questo risultato

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

La sua semantica è

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

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

Cosa abbiamo fatto vedere? Abbiamo appena mostrato che esiste un programma RAM in grado di simulare tutti gli altri programmi RAM.

Questo programma viene detto **interprete universale**.

Considereremo *"buono"* un linguaggio se esiste un interprete universale per esso.

10.2. Riflessioni sul concetto di calcolabilità

Ricordiamo che il nostro obiettivo è andare a definire la regione delle funzioni calcolabili e, di conseguenza, anche quella delle funzioni non calcolabili. Abbiamo visto che RAM e WHILE permettono di calcolare le stesse cose.

Possiamo definire ciò che è calcolabile a prescindere dalle macchine che usiamo per calcolare?

Come disse **Kleene**, vogliamo definire ciò che è calcolabile in termini più astratti, matematici, lontani dall'informatica. Se riusciamo a definire il concetto di calcolabile senza che siano nominate macchine calcolatrici, linguaggi e tecnologie ma utilizzando la matematica potremmo usare tutta la potenza di quest'ultima.

10.3. Chiusura di insiemi rispetto alle operazioni

10.3.1. Operazioni

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

$$\text{op} : \underbrace{U \times ... \times U}_{k} \longrightarrow U.$$

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

10.3.2. Chiusura

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

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

In poche parole, *se opero in A rimango 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** operazioni in Ω.

10.3.3. Chiusura di un insieme

Problema: siano $A \subseteq U$ e $\text{op} : U^k \longrightarrow U$, voglio espandere l'insieme A per trovare il più piccolo sottoinsieme di U tale che:
1. contiene A;
2. chiuso per op.

Quello che voglio fare è espandere A il minimo possibile per garantire la chiusura rispetto a op.

Due risposte ovvie a questo problema sono:
1. se A è chiuso rispetto a op, allora A stesso è l'insieme cercato;
2. sicuramente U soddisfa le due richieste, *ma è il più piccolo?*

Teorema Siano $A \subseteq U$ e $\text{op} : U^k \longrightarrow U$. Il più piccolo sottoinsieme di U contenente A e chiuso rispetto all'operazione op si ottiene calcolando la **chiusura di A rispetto a** op, e cioè l'insieme A^{op} definito **induttivamente** come:
1. $\forall a \in A \implies a \in A^{\text{op}}$;
2. $\forall a_1, ..., a_k \in A^{\text{op}} \implies \text{op}(a_1, ..., a_k) \in A^{\text{op}}$;
3. nient'altro sta in A^{op}.

Vediamo una 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. se trovi un risultato che non è già in A^{op} allora aggiungilo ad A^{op};
4. ripeti i punti (2) e (3) fintantoché A^{op} cresce.

11. Lezione 11

11.1. Chiusura di un insieme rispetto ad un insieme di operazioni

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

- $\forall a \in A \implies a \in A^\Omega$;
- $\forall i \in \{1, ..., t\}, \quad \forall a_1, ..., a_{ki} \in A^\Omega \implies \text{op}_i(a_i, ..., a_{ki}) \in A^\Omega$;
- nient'altro è in A^Ω.

11.2. Definizione teorica di calcolabilità

11.2.1. Roadmap

Seguiremo la seguente roadmap:

1. **ELEM**: definiamo un insieme di tre funzioni che **qualunque** idea di calcolabile si voglia proporre deve considerare calcolabili. ELEM non può esaurire il concetto di calcolabilità, quindi lo espanderemo con altre funzioni;
2. **Ω**: definiamo insieme di operazioni su funzioni che costruiscono nuove funzioni. Le operazioni in Ω sono banalmente implementabili e, applicandole a funzioni calcolabili, riesco a generare nuove funzioni calcolabili.
3. $\mathbf{ELEM}^{\mathbf{\Omega}} = \mathcal{P}$: definiamo la classe delle **funzioni ricorsive parziali**. Questa sarà la nostra idea astratta della classe delle funzioni calcolabili secondo Kleene.

Quello che faremo dopo la definizione di $\mathcal{P}$ sarò chiederci se questa idea di calcolabile che abbiamo *"catturato"* coincide o meno con le funzioni presenti in $F(\text{RAM}) = f(\text{WHILE})$.

11.2.2. Primo passo: ELEM

Definiamo l'insieme ELEM con le seguenti funzioni:

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

Questo insieme è un *onesto punto di partenza*: sono funzioni basilari che qualsiasi idea data teoricamente non può non considerare come calcolabile.

Ovviamente, ELEM non può essere considerato come l'idea teorica di *TUTTO* ciò che è calcolabile: infatti, la funzione $f(x) = x + 2$ non appartiene a ELEM ma è sicuramente calcolabile.

Quindi, ELEM è troppo povero e deve essere ampliato.

11.2.3. Secondo passo: Ω

Definiamo ora un insieme Ω di operazioni che amplino le funzioni di ELEM per permetterci di coprire tutte le funzioni calcolabili.

11.2.3.1. Composizione

Il primo operatore (*banale*) che ci viene in mente di utilizzare è quello di **composizione**.

Siano:

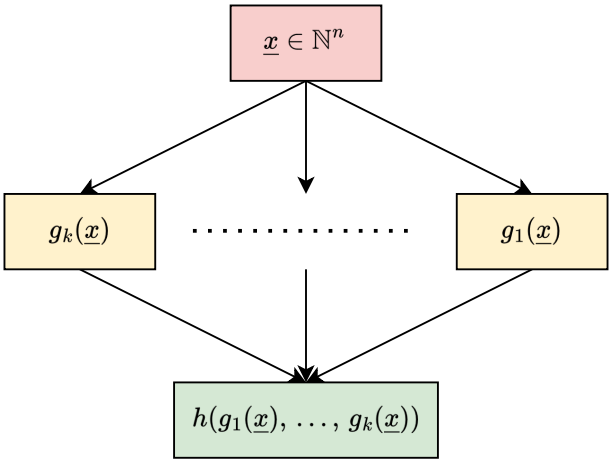
- $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ **funzione di composizione**,
- $g_1, ..., g_k : \mathbb{N}^n \longrightarrow \mathbb{N}$ *"funzioni intermedie"* e
- $\underline{x} \in \mathbb{N}^n$ input.

Allora definiamo

$$\mathrm{COMP}(h, g_1, ..., g_k) : \mathbb{N}^n \longrightarrow \mathbb{N}$$

la funzione tale che

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



COMP è una funzione *intuitivamente calcolabile* se parto da funzioni calcolabili: infatti, prima eseguo le singole funzioni $g_1, ..., g_k$ e poi le applico alla funzione h.

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

Vediamo subito come la funzione $f(x) = x + 2$ appartenga a questo insieme perché

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

Che altre funzioni ci sono in questo insieme? Sicuramente tutte le funzioni lineari del tipo $f(x) = x + k$, ma la somma?

La funzione

$$\mathrm{somma}(x, y) = x + y$$

non appartiene a questo insieme perché il valore y non è prefissato e noi non sappiamo *iterare* un certo numero di volte una certa funzione (in questo caso la funzione successore).

DObbiamo ampliare ancora $\mathrm{ELEM}^{\mathrm{COMP}}$ con altre operazioni.

11.2.3.2. Ricorsione primitiva

Definiamo un'operazione che ci permetta di **iterare** sull'operatore di composizione. Questa operazione è la **ricorsione primitiva** e viene usata per definire **funzioni ricorsive**, ovvero funzioni definite tramite loro stesse.

Siano:

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

Allora 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.

Come prima, chiudiamo $\mathrm{ELEM}^{\mathrm{COMP}}$ rispetto a RP, ovvero calcoliamo l'insieme $\mathrm{ELEM}^{\{\mathrm{COMP,RP}\}}$.

Questo insieme che otteniamo tramite la chiusura lo chiamiamo

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

ed è 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 che stanno in RICPRIM sono:

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

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

11.2.3.2.1. RICPRIM vs WHILE

L'insieme RICPRIM contiene molte funzioni, ma abbiamo raggiunto l'insieme $F(\mathrm{WHILE})$? Mostriamo che $\mathrm{RICPRIM} \subseteq F(\mathrm{WHILE})$ per induzione strutturale.

Prima di tutto, vediamo come è definita RICPRIM:

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

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

Dimostrazione
Dimostriamo prima di tutto il caso base. Le funzioni di ELEM sono ovviamente while programmabili, le avevamo mostrate in precedenza.

Passiamo ora al passo induttivo.

Per COMP, assumiamo per ipotesi induttiva che $h, g_1, ..., g_k \in \mathrm{RICPRIM}$ siano while programmabili, allora esistono $H, G_1, ..., G_k \in W\text{-PROG}$ tali che $\Psi_H = h, \Psi_{G_1} = g_1, ..., \Psi_{g_k} = g_k$. Mostro allora un programma WHILE che calcola COMP.

```
begin
  x0 := G1(x1);          # In x1 inizialmente ho <a1,...,an>
  x0 := [x0, G2(x1)];
  ...
  x0 := [x0, Gk(x1)];
  x1 := H(x0);
end
```

Per RP, assumiamo per ipotesi induttiva 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).

```
input(x,y)                # In x1 inizialmente ho <x,y>
begin
  t := G(x);              # t contiene f(x,y)
  k := 1;
  while k <= y do
    begin
      t := H(t, k-1, x);
      k := k + 1;
    end
end
```

□

Abbiamo quindi dimostrato che RICPRIM $\subseteq F(\text{WHILE})$, ma questa inclusione è propria?

11.2.3.2.2. Considerazioni

Notiamo subito che nel linguaggio WHILE posso fare dei cicli infiniti, mentre in RICPRIM questo non posso farlo: infatti, RICPRIM contiene solo funzioni totali (si dimostra per induzione strutturale) mentre WHILE contiene anche delle funzioni parziali.

Abbiamo quindi dimostrato che

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

Visto che le funzioni in RICPRIM sono tutte totali possiamo di dire che ogni ciclo in RICPRIM ha un inizio e una fine ben definiti: infatti, il costrutto usato per dimostrare che RP $\in F(\text{WHILE})$ nella dimostrazione precedente ci 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 che viene utilizzato è il *FOR giusto*, cioè quel costrutto che si serve di una **variabile di controllo** che parte da un preciso valore e arriva ad un valore limite (senza toccarlo) senza la possibilità di poterla modificare in corso d'opera. Questo costrutto nel linguaggio Pascal veniva implementato mettendo la variabile di controllo in un registro particolare per permettere la sua non modifica da esterni.

Il FOR language è quindi un linguaggio WHILE dove l'istruzione di loop è un FOR.

Possiamo quindi dire che $\mathrm{FOR} = \mathrm{RICPRIM}$, e quindi che $F(\mathrm{FOR})$ non raggiunge $F(\mathrm{WHILE})$.

Vediamo però che questo confronto sia un po' impari: WHILE vince su RICPRIM solo perché ha i loop infiniti. Restringiamo allora WHILE imponendo dei loop non infiniti: creiamo l'insieme

$$\widetilde{F}(\mathrm{WHILE}) = \{\Psi_W : W \in W\text{-PROG} \wedge \Psi_W \text{ totale}\}.$$

Dove si posizione questo insieme rispetto a RICPRIM? L'inclusione è propria?

La risposta è sì: vince ancora WHILE perché ci sono funzioni in $\widetilde{F}(\mathrm{WHILE})$ che non sono scrivibili come funzioni in RICPRIM. Ad esempio, la funzione di Ackermann del 1928 definita come

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

è una funzione che non appartiene a RICPRIM perché con la doppia ricorsione questa funzione cresce troppo in fretta, e quindi non può appartenere a RICPRIM.

Dobbiamo quindi ampliare RICPRIM.