

Informatica teorica

Indice

1. Lezione 01	5
1.1. Introduzione	5
1.1.1. Definizione	5
1.1.2. Cosa e come	5
1.2. Richiami matematici	5
1.2.1. Funzioni	5
2. Lezione 02	7
2.1. Richiami matematici	7
2.1.1. Funzioni totali e parziali	7
2.1.2. Totalizzare una funzione	7
2.1.3. Prodotto cartesiano	7
2.1.4. Insieme di funzioni	7
2.1.5. Funzione di valutazione	8
2.2. Teoria della calcolabilità	8
2.2.1. Sistema di calcolo	8
2.2.2. Potenza computazionale	9
3. Lezione 03	11
3.1. Relazioni di equivalenza	11
3.1.1. Definizione	11
3.1.2. Partizione	11
3.1.3. Classi di equivalenza e insieme quoziente	11
3.2. Cardinalità	11
3.2.1. Isomorfismi	11
3.2.2. Cardinalità finita	12
3.2.3. Cardinalità infinita	12
3.2.3.1. Insiemi numerabili	12
3.2.3.2. Insiemi non numerabili	12
4. Lezione 04	15
4.1. Cardinalità	15
4.1.1. Insieme delle parti	15
4.1.2. Insieme delle funzioni	15
4.2. Potenza computazionale	16
4.2.1. Validità dell'inclusione $F(\mathcal{C}) \subseteq \text{DATI}_{\perp}^{\text{DATI}}$	16
5. Lezione 05	18
5.1. $\text{DATI} \sim \mathbb{N}$	18
5.1.1. Funzione coppia di Cantor	18
5.1.1.1. Definizione	18
5.1.1.2. Forma analitica della funzione coppia	19
5.1.1.3. Forma analitica di sin e des	20
5.1.1.4. $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$	21
5.1.2. Dimostrazione	21
5.1.2.1. Strutture dati	21

5.1.2.1.1. Liste	21
5.1.2.1.2. Array	23
5.1.2.1.3. Matrici	23
5.1.2.1.4. Grafi	23
5.1.2.2. Applicazioni	24
5.1.2.2.1. Testi	24
5.1.2.2.2. Suoni	24
5.1.2.2.3. Immagini	24
5.1.3. Conclusioni	24
6. Lezione 06	25
6.1. $\text{PROG} \sim \mathbb{N}$	25
6.1.1. Sistema di calcolo RAM	25
6.1.1.1. Introduzione	25
6.1.1.2. Macchina RAM	25
6.1.1.3. Fasi dell'esecuzione	26
6.1.1.4. Definizione formale semantica	26
7. Lezione 07	29
7.1. Aritmetizzazione di un programma	29
7.2. Osservazioni	30
7.3. Sistema di calcolo WHILE	30
7.3.1. Macchina WHILE	30
7.3.2. Linguaggio WHILE	30
8. Lezione 08	32
8.1. Semantica di un programma while	32
8.2. Confronto macchina RAM e macchina WHILE	33
8.3. Traduzioni	34
9. Lezione 09	35
9.1. Introduzione	35
9.2. Compilatore per il linguaggio WHILE	35
9.2.1. Forma del compilatore	35
9.2.1.1. Assegnamento	35
9.2.1.2. Comando composto	36
9.2.1.3. Comando while	36
9.2.2. Risultati ottenuti	36
9.3. $F(\text{RAM}) \subseteq F(\text{WHILE})$	37
9.3.1. Interprete	37
9.3.1.1. Macro-WHILE	37
10. Lezione 10	39
10.1. Interprete WHILE di programmi RAM	39
10.1.1. Codice dell'interprete I_w	39
10.1.2. Conseguenza dell'esistenza di I_w	40
10.1.3. Altre conseguenze e osservazioni	41
10.1.4. Compilatore universale	41
10.2. Riflessioni sul concetto di calcolabilità	42
10.3. Chiusura di insiemi rispetto alle operazioni	42
10.3.1. Operazioni	42

10.3.2. Chiusura	42
10.3.3. Chiusura di un insieme	42
11. Lezione 11	44
11.1. Chiusura di un insieme rispetto ad un insieme di operazioni	44
11.2. Definizione teorica di calcolabilità	44
11.2.1. Roadmap	44
11.2.2. Primo passo: ELEM	44
11.2.3. Secondo passo: Ω	44
11.2.3.1.1. RICPRIM vs WHILE	46
11.2.4. Considerazioni	47
12. Lezione 12	49
12.1. Introduzione	49
12.2. Minimalizzazione	49
12.3. Classe \mathcal{P} delle Funzioni Ricorsive Parziali	50
12.4. Tesi di Church-Turing	53
12.4.1. Conseguenze della tesi	53
13. Lezione 13	54
13.1. Sistemi di programmazione	54
13.1.1. Introduzione	54
13.1.2. Assiomatizzazione di “sistema di programmazione buono”	54
13.1.2.1. Potenza computazionale	54
13.1.2.2. Interprete universale	54
13.1.2.3. Teorema S_1^1	55
13.1.3. Sistemi di programmazione accettabili (SPA)	56
13.1.4. Esistenza di compilatori tra SPA	56
14. Lezione 14	58
14.1. Teorema di Rogers	58
14.2. Introduzione	58
14.3. Esistenza di certi programmi negli SPA	58
14.4. Teorema di ricorsione	58
14.4.1. Definizione	58
14.4.2. Primo quesito: Quine	59
14.4.3. Secondo quesito: compilatori completamente errati	59
14.4.4. Dimostrazione	60
14.5. Equazioni su SPA	60
14.5.1. Strategia	60
14.5.2. Esercizi	61
14.5.2.1. Esercizio 01	61
14.5.2.2. Esercizio 02	61
14.5.2.3. Esercizio 03	62
14.5.2.4. Esercizio 04	62
14.5.2.5. Esercizio 05	62
14.5.2.6. Esercizio 06	62
15. Lezione 15	63
15.1. Problemi di decisione	63
15.1.1. Introduzione	63
15.1.2. Esempi	63

15.1.3. Decidibilità	64
15.1.3.1. Applicazione agli esempi	65
15.1.4. Problemi indecidibili	65
15.1.4.1. Problema dell'arresto ristretto	66
15.1.4.2. Problema dell'arresto	68

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 $f : A \rightarrow B$, 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 :

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

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 $f : A \rightarrow B$ si dice *iniettiva* se e solo se:

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

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

- **suriettività**: una funzione $f : A \rightarrow B$ 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**:

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

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

Infine, una funzione $f : A \rightarrow B$ 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 $f : A \rightarrow B$ è una funzione biiettiva, si definisce **inversa** di f la funzione $f^{-1} : B \rightarrow A$ 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 $f : A \rightarrow B$ e $g : B \rightarrow C$, la funzione f *composto* g è la funzione $g \circ f : A \rightarrow C$ 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 $i_A : A \rightarrow A$ 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 $f : A \rightarrow B$ biiettiva, la sua inversa è l'unica funzione $f^{-1} : B \rightarrow A$ che soddisfa:

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

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 $\exists a \in A \mid f(a) \downarrow$.

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

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

Notiamo che:

- $\text{Dom}_f \subsetneq A \implies f$ parziale;
- $\text{Dom}_f = A \implies f$ totale.

2.1.2. Totalizzare una funzione

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

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

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

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

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 $\pi_i : A_1 \times \dots \times A_n \longrightarrow A_i$ tale che

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

Per comodità chiameremo $\underbrace{A \times \dots \times A}_n = A^n$

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

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

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

$$\omega : B_{\perp}^A \times A \rightarrow B$$

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

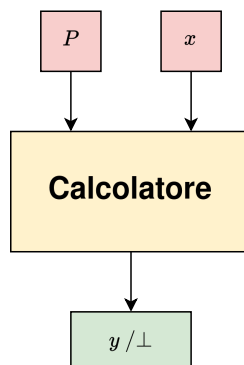
2.2.1. Sistema di calcolo

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

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

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

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

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

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

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

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

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

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

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

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

- $F(\mathcal{C}) \subsetneq \text{DATI}_\perp^{\text{DATI}}$ allora esistono compiti **non automatizzabili**;
- $F(\mathcal{C}) = \text{DATI}_\perp^{\text{DATI}}$ 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 associa 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**: $\forall a, b, c \in A \quad aRb \wedge bRc \implies aRc$.

3.1.2. Partizione

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

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

Diremo che R definita su A^2 induce una partizione A_1, A_2, \dots 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 $[a]_R \cap [b]_R = \emptyset$ 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*: $A \sim B \wedge B \sim C \implies A \sim C$ 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:

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

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:

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

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:

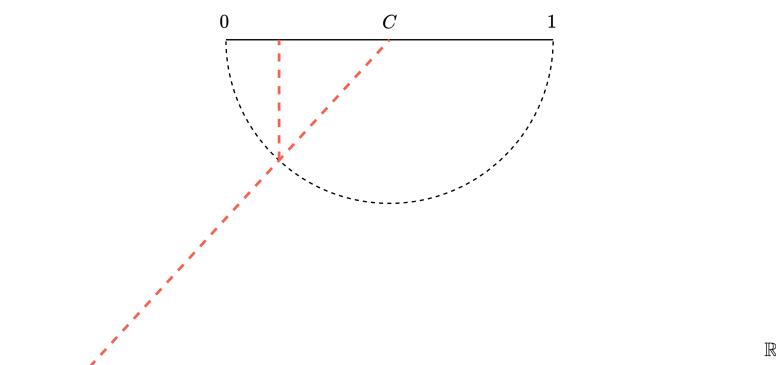
1. dimostriamo che $\mathbb{R} \sim (0, 1)$;
2. dimostriamo che $\mathbb{N} \sim (0, 1)$;
3. dimostriamo che $\mathbb{R} \sim \mathbb{N}$.

[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” $\mathbb{R}^\circ = \mathbb{R} \cup \{\pm\infty\}$ di \mathbb{R} , mappando 0 su $-\infty$ e 1 su $+\infty$.

[2] Continuiamo dimostrando che $\mathbb{N} \sim (0, 1)$: 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:

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

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

Costruisco il “numero colpevole” $c = 0.c_0c_1c_2\dots$ tale che

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

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 $N \approx (0, 1)$.

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

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

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 $P(\mathbb{N}) \approx \mathbb{N}$.

Dimostrazione

Dimostriamo questo teorema con la diagonalizzazione.

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

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

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

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

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

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

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 $P(\mathbb{N}) \approx \mathbb{N}$. □

Visto questo teorema possiamo affermare che:

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

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

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

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

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 $\{f_0, f_1, f_2, \dots\}$.

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

Scriviamo un colpevole $\varphi : \mathbb{N} \rightarrow \mathbb{N}_{\perp}$ 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 $P(\mathbb{N}) \approx \mathbb{N}$. □

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:

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

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

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

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

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

$$\text{sin} : \mathbb{N}^+ \longrightarrow \mathbb{N}, \quad \text{sin}(n) = x$$

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

Vediamo una rappresentazione grafica della funzione di Cantor.

$\begin{array}{c c} x & y \end{array}$	0	1	2	3	4	...
0	1	3	6	10		...
1	2	5	9			...
2	4	8				...
3	7					...
4	11					...
...

$\langle x, y \rangle$ 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.

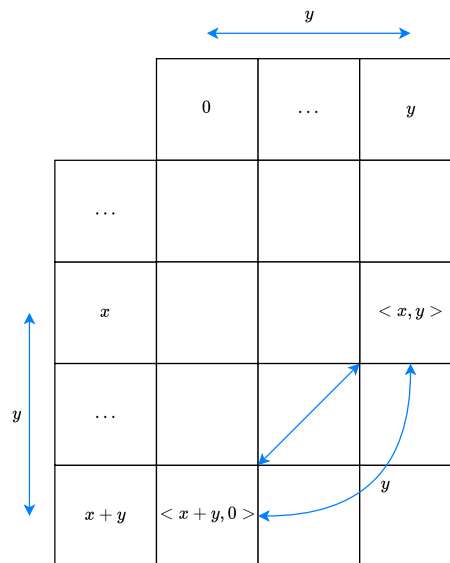
Questa richiesta è soddisfatta in quanto:

- 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

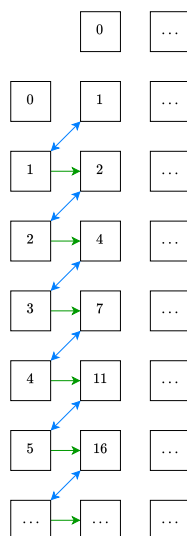
Quello che vogliamo fare ora è cercare una forma analitica della funzione coppia, questo perché non è molto comodo costruire ogni volta la tabella sopra. Nella successiva immagine notiamo come valga la relazione

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



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

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



Ogni cella $\langle z, 0 \rangle$ la si può calcolare come la somma di z e $\langle z - 1, 0 \rangle$, ma allora

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

Questa forma è molto più compatta ed evita il calcolo di tutti i singoli $\langle z, 0 \rangle$.

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

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

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

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

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

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

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

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

Risolviamo quindi la disequazione

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

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

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

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 biettiva tra l'insieme $\mathbb{N} \times \mathbb{N}$ e l'insieme \mathbb{N}^+ , quindi i due insiemi sono isomorfi. \square

Estendiamo adesso il risultato all'intero 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

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

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

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 $[x_1, \dots, x_n]$ con

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

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:

$$\begin{aligned} \langle 1, 2, 5 \rangle &= \langle 1, \langle 2, \langle 5, 0 \rangle \rangle \rangle \\ &= \langle 1, \langle 2, 16 \rangle \rangle \\ &= \langle 1, 188 \rangle \\ &= 18144. \end{aligned}$$

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 $\langle x_n, 0 \rangle$. Questo ci indica che quando $\text{des}(M)$ 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 delle implementazioni di queste funzioni. Assumiamo che:

- 0 codifichi la lista nulla;
- esistano delle routine per \langle, \rangle , *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 $\{x_1, \dots, x_n\}$ viene codificato con:

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

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

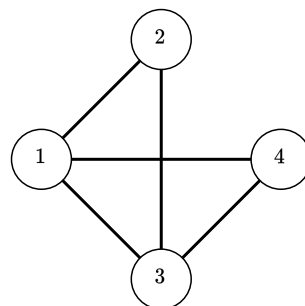
$$\begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

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,

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

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 $f : \text{DATI} \rightarrow \text{DATI}_\perp$ con delle funzioni $f' : \mathbb{N} \rightarrow \mathbb{N}_\perp$. 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:** $R_k \leftarrow R_k + 1$;
- **decremento:** $R_k \leftarrow R_k \dot{-} 1$;
- **salto condizionato:** IF $R_k = 0$ THEN GOTO m , con $m \in \{1, \dots, |P|\}$.

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 $P \equiv \text{Istr}_1, \dots, \text{Istr}_n$ 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 $\varphi_P : \mathbb{N} \rightarrow \mathbb{N}_\perp$ 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 operativa** 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 (StatoPrima, StatoDopo) rappresenta la semantica operativa 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

$$S_0, S_1, \dots, S_i, S_{i+1}, \dots$$

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:

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

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

7. Lezione 07

7.1. Aritmetizzazione di un programma

Per verificare che vale $\text{PROG} \sim \mathbb{N}$ dobbiamo trovare un modo per codificare i programmi in numeri in modo biunivoco. Notiamo che, data la lista di istruzioni semplici $P \equiv \text{Istr}_1, \dots, \text{Istr}_m$, 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

$$[\text{Istr}_1, \dots, \text{Istr}_n] \xrightarrow{\text{Ar}} [c_1, \dots, c_n] \xrightarrow{\langle \rangle} n.$$

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

$$n \xrightarrow{\langle \rangle} [c_1, \dots, c_n] \xrightarrow{\text{Ar}^{-1}} [\text{Istr}_1, \dots, \text{Istr}_n].$$

La nostra funzione “*complessiva*” è biunivoca se dimostriamo la biunivocità della funzione Ar , avendo già dimostrato questa proprietà per $\langle \rangle$.

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

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

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

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

La codifica del programma P è quindi

$$\text{cod}(P) = \langle \text{Ar}(\text{Istr}_1), \dots, \text{Ar}(\text{Istr}_n) \rangle.$$

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

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

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

quindi anche nel sistema di calcolo RAM esistono funzioni non 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 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:

$$\begin{aligned}x_k &:= 0, \\x_k &:= x_j + 1, \\x_k &:= x_j \div 1.\end{aligned}$$

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

$$\begin{aligned}R_k &\longleftarrow R_k + 1 \\R_k &\longleftarrow R_k \div 1\end{aligned}$$

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

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

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

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

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

Per dimostrare una proprietà P di un programma $W \in W\text{-PROG}$ 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, \dots, C_n e la dimostro vera per begin $C_1; \dots; C_n$ 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 $\Psi_W : \mathbb{N} \rightarrow \mathbb{N}_\perp$ 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 $\underline{x} = (c_0, \dots, c_{20})$ rappresenta il nostro stato, con c_i contenuto della variabile i ;
2. **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

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

5. **semantica operativa**: 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

$$\llbracket () \rrbracket : W\text{-COM} \times W\text{-STATI} \rightarrow W\text{-STATI}_\perp$$

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

$$\llbracket C \rrbracket(\underline{x}) = \underline{y},$$

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**:

$$\begin{aligned} \llbracket x_k := 0 \rrbracket(\underline{x}) = \underline{y} &= \begin{cases} x_i & \text{se } i \neq k \\ 0 & \text{se } i = k \end{cases}, \\ \llbracket x_k := x_j \pm 1 \rrbracket(\underline{x}) = \underline{y} &= \begin{cases} x_i & \text{se } i \neq k \\ x_j \pm 1 & \text{se } i = k \end{cases}. \end{aligned}$$

Proseguiamo con il passo induttivo, quindi:

- **comando composto**: vogliamo calcolare

$$\llbracket \text{begin } C_1; \dots; C_n \text{ end} \rrbracket(\underline{x})$$

conoscendo ogni $\llbracket C_i \rrbracket$ per ipotesi induttiva. Calcoliamo allora la funzione:

$$\llbracket C_n \rrbracket (\dots (\llbracket C_2 \rrbracket (\llbracket C_1 \rrbracket (\underline{x}))) \dots) = (\llbracket C_n \rrbracket \circ \dots \circ \llbracket C_1 \rrbracket)(\underline{x}),$$

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

- **comando while:** vogliamo calcolare

$$\llbracket \text{while } x_k \neq 0 \text{ do } C \rrbracket (\underline{x})$$

conoscendo ogni $\llbracket C_i \rrbracket$ per ipotesi induttiva. Calcoliamo allora la funzione:

$$\llbracket C \rrbracket (\dots (\llbracket C \rrbracket (\underline{x})) \dots).$$

Dobbiamo capire quante volte eseguiamo il loop: dato $\llbracket C \rrbracket^e$ (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:

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

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

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

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

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

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

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

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

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 $C_1\text{-PROG}$ e $C_2\text{-PROG}$ e potenze computazionali

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

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

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

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

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

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

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

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

con le seguenti proprietà:

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

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

dove Ψ rappresenta la semantica dei programmi in $C_1\text{-PROG}$ e φ rappresenta la semantica dei programmi in $C_2\text{-PROG}$.

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

Dimostrazione

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

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

Ho trovato un programma $P_2 \in C_2\text{-PROG}$ 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

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

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 $W\text{-PROG}$ 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 $\text{Comp}(C_1), \dots, \text{Comp}(C_m)$ e mostro come compilare il comando composto $\text{begin } C_1; \dots; C_n \text{ end}$;
 2. per ipotesi induttiva, assumo di sapere $\text{Comp}(C)$ e mostro come compilare il comando $\text{while } x_k \neq 0 \text{ do } C$.

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 $x_k := 0$.

$$\text{Comp}(x_k := 0) = \text{LOOP} : \text{IF } R_k = 0 \text{ THEN GOTO EXIT}$$

$$R_k \leftarrow R_k \div 1$$

$$\text{IF } R_{21} = 0 \text{ THEN GOTO LOOP}$$

$$\text{EXIT} : R_K \leftarrow R_K \div 1 \quad .$$

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 $x_k := x_j \div 1$.

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

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

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 .

```

Comp( $x_k := x_j \pm 1$ ) = LOOP : IF  $R_j = 0$  THEN GOTO EXIT1
                         $R_j \leftarrow R_j \div 1$ 
                         $R_{22} \leftarrow R_{22} + 1$ 
                        IF  $R_{21} = 0$  THEN GOTO LOOP
EXIT1 : IF  $R_k = 0$  THEN GOTO EXIT2
         $R_k \leftarrow R_k \div 1$ 
        IF  $R_{21} = 0$  THEN GOTO EXIT1
EXIT2 : IF  $R_{22} = 0$  THEN GOTO EXIT3
         $R_k \leftarrow R_k + 1$ 
         $R_j \leftarrow R_j + 1$ 
         $R_{22} \leftarrow R_{22} \div 1$ 
        IF  $R_{21} = 0$  THEN GOTO EXIT2
EXIT3 :  $R_k \leftarrow R_k \pm 1$  .

```

9.2.1.2. Comando composto

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

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

9.2.1.3. Comando while

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

$$\text{Comp}(\text{while } x_k \neq 0 \text{ do } C) = \text{LOOP : IF } R_k = 0 \text{ THEN GOTO EXIT} \\ \text{Comp}(C) \\ \text{IF } R_{21} = 0 \text{ THEN GOTO LOOP} \\ \text{EXIT : } R_k \leftarrow R_k \div 1 \quad .$$

9.2.2. Risultati ottenuti

La funzione

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

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 ci 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 $\langle x, n \rangle$.

La semantica di I_W diventa

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

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:

- $x_k := x_j + x_s$;
- $x_k := \langle x_j, x_s \rangle$;
- $x_k := \langle x_1, \dots, x_n \rangle$;
- 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;
- $x_k := \sin(x_j)$;
- $x_k := \text{des}(x_j)$;

- costruito 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, \dots 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, \dots, R_{n+2} . Usiamo i registri fino 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 $< R_0, \dots, R_{n+2} >$, 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>);
x2 := sin(x1);
x3 := des(x1);
x0 := <0,x2,0,...,0>;
x1 := 1;

# Esecuzione
while (x1 != 0) do:
  if (x1 > length(x2)) then
    x1 := 0;
  else
    x4 := Pro(x1, x3);
    if (x4 mod 3 == 0) then
      x5 := x4 / 3;
      x0 := incr(x5, x0);
      x1 := x1 + 1;
    if (x4 mod 3 == 1) then
      x5 := (x4 - 1) / 3;
      x0 := decr(x5, x0);
      x1 := x1 + 1;
    if (x4 mod 3 == 2) then
      x5 := sin((x4 + 1) / 3);
      x6 := des((x4 + 1) / 3);
      if (Proj(x5, x0) == 0) then
        x1 := x4;
      else
        x1 := x1 + 1;

# Finalizzazione
x0 := sin(x0);

```

In x1
 # Dato x
 # Programma n
 # Memoria iniziale
 # Program counter
 # Se x1 = 0 HALT
 # Sono fuori dal programma?
 # Vado in HALT
 # Fetch istruzione
 # Incremento?
 # Trovo k
 # Incremento
 # Istruzione successiva
 # Decremento?
 # Trovo k
 # Decremento
 # Istruzione successiva
 # GOTO?
 # Trovo k
 # Trovo m
 # Devo saltare?
 # Salto a m
 # Istruzione successiva
 # Oppure Pro(0,x0)

10.1.2. Conseguenza dell'esistenza di I_w

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

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

tale che

$$\begin{aligned} \text{Comp}(P \in \text{PROG}) &\equiv n \leftarrow \text{cod}(P) \\ &\quad x_1 := \langle x_1, n \rangle \\ &\quad I_W \end{aligned}$$

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 $P \in \text{PROG} \mapsto \text{Comp}(P) \in W\text{-PROG}$, 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 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 (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

$$\begin{aligned} F(\text{WHILE}) &\subseteq F(\text{RAM}) \\ F(\text{RAM}) &\subseteq F(\text{WHILE}) \\ &\Downarrow \\ F(\text{RAM}) &= F(\text{WHILE}) \\ &\Downarrow \\ F(\text{RAM}) &= F(\text{WHILE}) \sim \text{PROG} \sim \mathbb{N} \approx \mathbb{N}_{\perp}^{\mathbb{N}}. \end{aligned}$$

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

$$\text{op} : \underbrace{U \times \dots \times U}_k \longrightarrow U.$$

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

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

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. $\forall a_1, \dots, a_k \in A^{\text{op}} \implies \text{op}(a_1, \dots, a_k) \in A^{\text{op}}$;
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 $\Omega = \{\text{op}_1, \dots, \text{op}_t\}$ un insieme di operazioni su U di arietà rispettivamente k_1, \dots, k_t e $A \subseteq U$. Definiamo **chiusura di A rispetto a Ω** il più piccolo sottoinsieme di U contenente A e chiuso rispetto a Ω , cioè l'insieme A^Ω definito come:

- $\forall a \in A \implies a \in A^\Omega$;
- $\forall i \in \{1, \dots, t\}, \quad \forall a_1, \dots, a_{k_i} \in A^\Omega \implies \text{op}_i(a_1, \dots, a_{k_i}) \in A^\Omega$;
- 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 esanderemo 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. **$\text{ELEM}^\Omega = \mathcal{P}$** : 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} sarà chiederci se questa idea di calcolabile che abbiamo “catturato” coincida 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:

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

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.

1. Composizione

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

Siano:

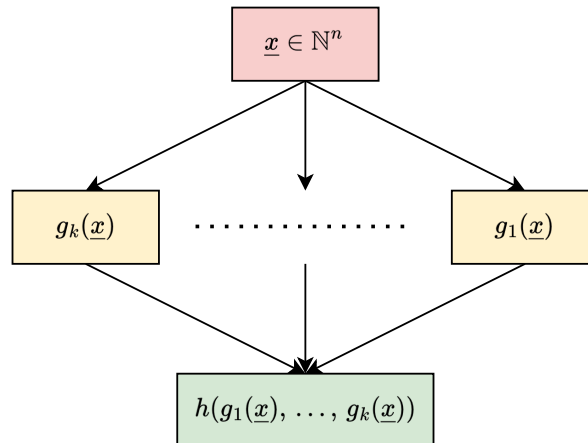
- $h : \mathbb{N}^k \longrightarrow \mathbb{N}$ **funzione di composizione**,
- $g_1, \dots, g_k : \mathbb{N}^n \longrightarrow \mathbb{N}$ “funzioni intermedie” e
- $\underline{x} \in \mathbb{N}^n$ input.

Allora definiamo

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

la funzione tale che

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



COMP è una funzione *intuitivamente calcolabile* se parto da funzioni calcolabili: infatti, prima eseguo le singole funzioni g_1, \dots, g_k e poi applico la funzione h sui risultati delle funzioni g_i .

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

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

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

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

La funzione

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

non appartiene a questo insieme perché il valore y non è prefissato e non abbiamo ancora definito il concetto di iterazione di una funzione (in questo caso la funzione successore).

Dobbiamo ampliare ancora $\text{ELEM}^{\text{COMP}}$ con altre operazioni.

2. Ricorsione primitiva

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

Siano:

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

Definiamo

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

funzione che generalizza la definizione ricorsiva di funzioni.

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

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

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

In questo insieme abbiamo la somma: infatti,

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

Altre funzioni che stanno in RICPRIM sono:

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

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

11.2.3.1.1. RICPRIM vs WHILE

L'insieme RICPRIM contiene molte funzioni, *ma abbiamo raggiunto l'insieme $F(WHILE)$?*

Vediamo come è definita RICPRIM:

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

Teorema $RICPRIM \subseteq F(WHILE)$.

Dimostrazione

Passo base:

Le funzioni di ELEM sono ovviamente while programmabili, le avevamo mostrate in precedenza.

Passo induttivo:

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

```

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

```

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

Per RP, assumiamo che $h, g \in \text{RICPRIM}$ siano while programmabili, allora esistono $H, G \in W\text{-PROG}$ 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

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

```

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
end

```

Quindi $\Psi_w(<x, y>) = \text{RP}(h, g)(\underline{x}, y)$. □

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

11.2.4. Considerazioni

Notiamo subito che nel linguaggio WHILE posso fare dei cicli infiniti, mentre in RICPRIM no: RICPRIM contiene solo funzioni totali (si dimostra per induzione strutturale) mentre WHILE contiene anche delle funzioni parziali.

Di conseguenza

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

Per poter raggiungere $F(\text{WHILE})$ dovremo ampliare nuovamente RICPRIM.

Visto che le funzioni in RICPRIM sono tutte totali, possiamo dire che ogni ciclo in RICPRIM ha un inizio e una fine ben definiti: il costrutto utilizzato per dimostrare che $\text{RP} \in F(\text{WHILE})$ 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 quello *originale*, cioè quel costrutto che si serve di una **variabile di controllo** che parte da un preciso valore e arriva ad un valore limite, senza che la variabile di controllo venga toccata. In Pascal veniva implementato mettendo la variabile di controllo in un registro particolare, per non permettere la sua scrittura.

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

Possiamo quindi dire che $\text{FOR} = \text{RICPRIM}$, e quindi che $F(\text{FOR}) \subset F(\text{WHILE})$.

Dato che WHILE vince su RICPRIM solo per i loop infiniti, restringiamo WHILE imponendo dei loop finiti. Creiamo l'insieme

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

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

Anche in questo caso, “vince” ancora WHILE, perché ci sono funzioni in $\tilde{F}(\text{WHILE})$ che non sono scrivibili come funzioni in RICPRIM. Ad esempio, la funzione di Ackermann (1928), definita come

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

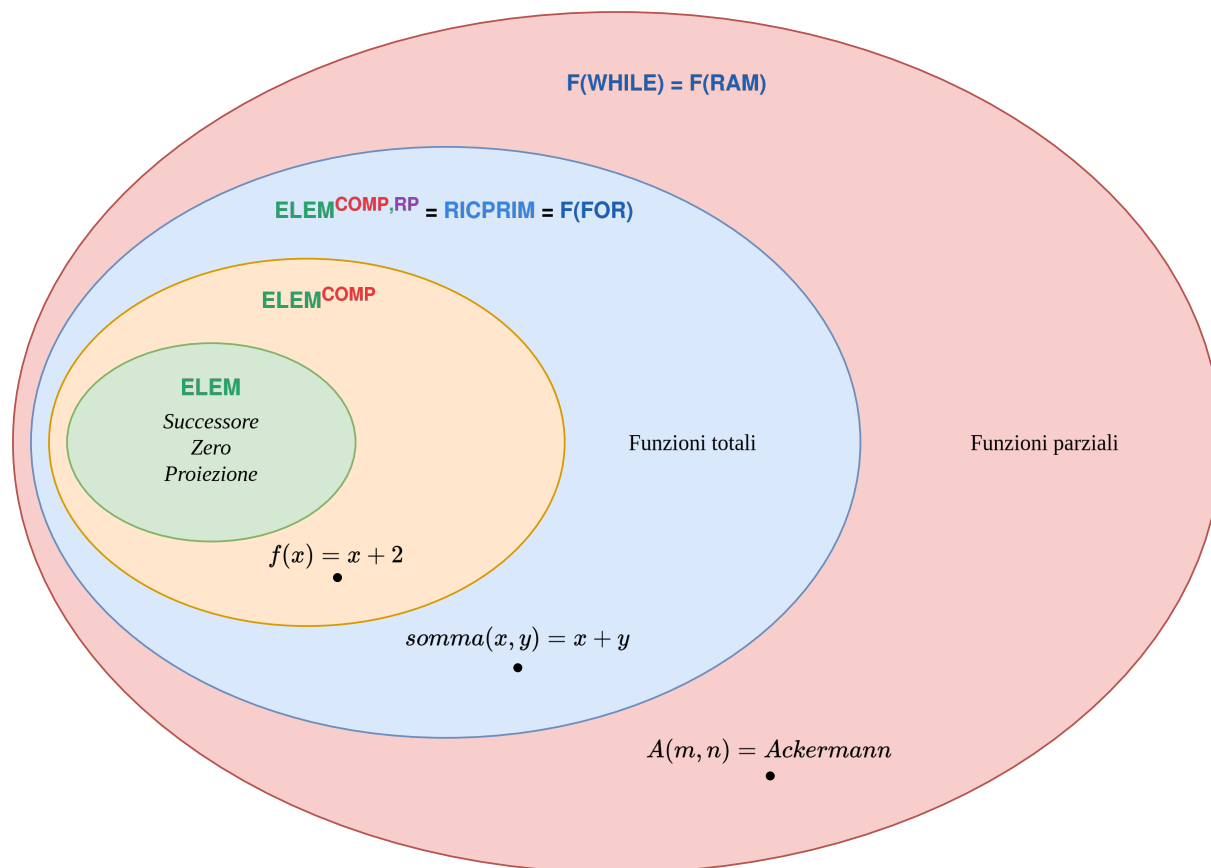
è una funzione che non appartiene a RICPRIM, perché a causa della doppia ricorsione cresce troppo in fretta.

Di conseguenza, vogliamo ampliare anche RICPRIM.

12. Lezione 12

12.1. Introduzione

Vogliamo una definizione di calcolabilità che non coinvolga i linguaggi, in modo da renderla il più generale possibile.



Siamo nella direzione giusta: non abbiamo catturato “cose strane” che non avrei catturato in $F(\text{RAM})$, ma questo non basta. Infatti, dobbiamo ampliare ancora RICPRIM perché non abbiamo ancora catturato le funzioni parziali.

12.2. Minimalizzazione

Introduciamo quindi un nuovo operatore per permettere la presenza di funzioni parziali.

L'operatore scelto è l'operatore di **minimalizzazione** di funzione.

Sia $f : \mathbb{N}^{n+1} \rightarrow \mathbb{N}$ con $f(\underline{x}, y)$ e $\underline{x} \in \mathbb{N}^n$, allora:

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

Un'altra definizione di MIN è

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

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

Vediamo alcuni esempi con $f : \mathbb{N}^2 \rightarrow \mathbb{N}$.

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

12.3. Classe \mathcal{P} delle Funzioni Ricorsive Parziali

Ampliamo RICPRIM chiudendolo con la nuova operazione MIN:

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

Sicuramente \mathcal{P} , che grazie a MIN ora contiene anche funzioni parziali, amplia RICPRIM, fatto solo di funzioni totali. *Ma come si pone rispetto a $F(\text{WHILE})$?*

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

Dimostrazione

\mathcal{P} è definito per chiusura, ma in realtà è definito induttivamente in questo modo:

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

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

- **passo base:** le funzioni elementari sono WHILE programmabili, lo abbiamo già dimostrato;
- **passi induttivi:**
 - siano $h, g_1, \dots, g_k \in \mathcal{P}$ WHILE programmabili per ipotesi induttiva, allora mostro che $\text{COMP}(h, g_1, \dots, g_k)$ è WHILE programmabile, ma questo lo abbiamo già fatto per RICPRIM;
 - siano $h, g \in \mathcal{P}$ WHILE programmabili per ipotesi induttiva, allora mostro che $\text{RP}(h, g)$ è WHILE programmabile, ma anche questo lo abbiamo già fatto per RICPRIM;
 - sia $f \in \mathcal{P}$ WHILE programmabile per ipotesi induttiva, allora mostro che $\text{MIN}(f)$ è WHILE programmabile. Devo trovare un programma WHILE che calcoli la minimizzazione: il programma WHILE

```

P ≡ input(x)
begin
  y := 0
  while f(x, y) ≠ 0 do
    y := y + 1
end
```

è un programma che calcola la minimizzazione: infatti, se non esiste un y che azzeri $f(x, y)$ il programma va in loop, quindi la semantica di P è \perp secondo MIN.

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

Viene naturale chiedersi se vale la relazione inversa, cioè se $F(\text{WHILE}) \subseteq \mathcal{P}$ oppure no.

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

Dimostrazione

Sappiamo che

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

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

Le funzione in $W\text{-PROG}$ sono nella forma

$$\Psi_W = \text{Pro}_0^{21}(\llbracket W \rrbracket(\text{w-in}(\underline{x}))),$$

con $\llbracket C \rrbracket(\underline{x}) = \underline{y}$ la funzione che calcola lo stato prossimo $\underline{y} \in \mathbb{N}^{21}$ a seguito dell'esecuzione del comando C a partire dallo stato corrente $\underline{x} \in \mathbb{N}^{21}$.

In sostanza, $\llbracket \cdot \rrbracket : \mathbb{N}^{21} \rightarrow \mathbb{N}^{21}$ rappresenta la funzione di stato prossimo definita induttivamente per via della struttura induttiva del linguaggio WHILE.

Abbiamo definito Ψ_W come composizione delle funzioni Pro_0^{21} e $\llbracket W \rrbracket(\text{w-in}(\underline{x}))$, ma allora:

1. $\text{Pro}_0^{21} \in \text{ELEM} \implies \text{Pro}_0^{21} \in \mathcal{P}$;
2. \mathcal{P} è chiuso rispetto alla composizione;
3. a causa delle due precedenti, se dimostro che la funzione di stato prossimo è ricorsiva parziale allora $\Psi_W \in \mathcal{P}$ per la definizione induttiva di \mathcal{P} .

La funzione di stato prossimo restituisce elementi in \mathbb{N}^{21} , mentre gli elementi in \mathcal{P} hanno codominio \mathbb{N} . Per risolvere questo piccolo problema tramite le liste di Cantor riesco a condensare il vettore in un numero.

Consideriamo quindi $f_C(x) = y$ **funzione numero prossimo**, con $x = [\underline{x}]$ e $y = [\underline{y}]$.

$$\begin{array}{c} f_C(x) = y \\ \swarrow \quad \downarrow \quad \searrow \quad \swarrow \quad \searrow \\ \llbracket C \rrbracket(\text{Pro}(0, x), \dots, \text{Pro}(20, x)) = (\text{Pro}(0, y), \dots, \text{Pro}(20, y)) \end{array}$$

Ovviamente

$$f_C \in \mathcal{P} \iff \llbracket C \rrbracket \in \mathcal{P}$$

dato che posso passare da una all'altra usando funzioni in \mathcal{P} quali Cantor e proiezioni.

Dimostriamo, tramite induzione strutturale, sul comando while C :

- **caso base:** i comandi base WHILE devono stare in \mathcal{P} .

- **azzeramento** $C \equiv x_k := 0$:

$$f_{x_k:=0}(x) = \left[\text{Pro}(0, x), \dots, \underbrace{0(x)}_k, \dots, \text{Pro}(20, x) \right].$$

Tutte le funzioni usate sono in \mathcal{P} , così come la loro composizione;

- **incremento/decremento** $C \equiv x_k := x_j \pm 1$:

$$f_{x_k:=x_j\pm 1}(x) = [\text{Pro}(x, 0), \dots, \text{Pro}(j, x) \pm 1, \dots, \text{Pro}(20, x)].$$

Tutte le funzioni usate sono in \mathcal{P} , così come la loro composizione;

- **passi induttivi:** i comandi “complessi” WHILE devono stare in \mathcal{P} .

- **comando composto** $C \equiv \text{begin } C_1; \dots; C_n \text{ end}$ sapendo che $f_{C_i} \in \mathcal{P}$:

$$f_C(x) = f_{C_n}(\dots(f_{C_2}(f_{C_1}(x)))\dots).$$

Ogni $f_{C_i} \in \mathcal{P}$ per ipotesi induttiva, così come la loro composizione;

- **comando while** $C' \equiv \text{while } x_k \neq 0 \text{ do } C$, sapendo che $f_C \in \mathcal{P}$:

$$f_{C'}(x) = f_C^{e(x)}(x),$$

con

$$e(x) = \mu_y(\text{Pro}(k, f_C^y(x)) = 0).$$

$f_C^{e(x)}$ è la composizione di f_C per $e(x)$ volte, che non è un numero costante dato che dipende dallo stato iniziale x , ma questo è problema: grazie all'operatore di composizione sappiamo comporre un numero predeterminato di volte, *ma come facciamo con un numero non costante?*

Rinominiamo

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

Come rappresento $T(x, y)$ in \mathcal{P} ?

Notiamo come $T(x, y)$ sia un operatore ottenuto tramite RP su una funzione $f_C \in \mathcal{P}$, di conseguenza anche lei starà in \mathcal{P} .

L'ultima cosa da sistemare è $e(x)$. Questa funzione è la minimizzazione di $T(x, y)$: infatti,

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

cerca il primo numero che azzerà il registro k , quindi $e(x) \in \mathcal{P}$.

In conclusione,

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

è composizione di funzioni in \mathcal{P} , quindi $f_{C'} \in \mathcal{P}$.

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

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

Abbiamo ottenuto che la classe delle funzioni ricorsive parziali, che dà un'idea di *calcolabile* in termini matematici, coincide con quello che noi intuitivamente consideriamo *calcolabile*, dove con “intuitivamente calcolabile” intendiamo tutti quei problemi di cui vediamo una macchina che li risolve.

12.4. Tesi di Church-Turing

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

- \mathcal{P} insieme delle funzioni ricorsive parziali;
- \mathcal{T} insieme delle funzioni ricorsive totali.

Il secondo insieme presentato contiene tutte le funzioni di \mathcal{P} che sono totali, ma allora

$$\mathcal{T} \subset \mathcal{P}.$$

Inoltre vale

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

perché, ad esempio, la funzione di Ackermann $\mathcal{A}(m, n)$ non sta in RICPRIM (già dimostrato) ma è sicuramente calcolabile e totale.

L'insieme \mathcal{P} cattura tutti i sistemi di calcolo esistenti: WHILE, RAM, Macchine Di Turing, Lambda-calcolo di Church, paradigma quantistico, grammatiche, circuiti, sistemi di riscrittura, eccetera. In poche parole, tutti i sistemi creati dal 1930 ad oggi.

Infatti, dal 1930 in poi sono stati proposti un sacco di modelli di calcolo che volevano catturare ciò che è calcolabile, ma tutti questi modelli individuavano sempre la classe delle funzioni ricorsive parziali. Visti questi risultati, negli anni 1930/1940 **Church** e **Turing** decidono di enunciare un risultato molto importante.

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

Questa tesi non è un teorema, è una **congettura**, un'opinione. Non può essere un teorema in quanto non è possibile caratterizzare i modelli di calcolo ragionevoli che sono stati e saranno proposti in maniera completa. Possiamo semplicemente decidere se aderire o meno a questa tesi.

12.4.1. Conseguenze della 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**, quindi che non va mai in loop.

13. Lezione 13

13.1. Sistemi di programmazione

13.1.1. Introduzione

Fin'ora, nello studio dei **sistemi di programmazione**, ci siamo concentrati su una loro caratteristica principale: la *potenza computazionale*. Con la tesi di Church-Turing abbiamo affermato che ogni sistema di programmazione ha come potenza computazionale \mathcal{P} , cioè l'insieme delle funzioni ricorsive parziali.

Oltre a questo, vorremmo sapere altro sui sistemi di programmazione, ad esempio la possibilità o l'impossibilità di scrivere programmi su certi compiti.

Vorremmo, come sempre, rispondere nel modo più rigoroso e generale possibile, quindi non considereremo un particolare sistema di programmazione, ma studieremo proprietà valide per tutti i sistemi di programmazione "ragionevoli".

Dobbiamo astrarre un sistema di calcolo generale che permetta di rappresentarli tutti.

13.1.2. Assiomatizzazione di "sistema di programmazione buono"

Assiomatizzare significa *dare un insieme di proprietà* che i sistemi di calcolo devono avere per essere considerati buoni.

Da qui in poi individueremo un sistema di programmazione con

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

ovvero l'insieme delle funzioni calcolabili con quel sistema, in altre parole l'insieme delle sue semantiche. Il pedice $i \in \mathbb{N}$ indica i programmi (*codificati*) di quel sistema.

Troveremo tre proprietà che un sistema di programmazione deve avere per essere considerato buono e lo faremo prendendo spunto dal sistema RAM.

13.1.2.1. Potenza computazionale

La prima proprietà che vogliamo in un sistema di programmazione riguarda la **potenza computazionale**. Dato il sistema $\{\varphi_i\}$ vogliamo che

$$\{\varphi\}_i = \mathcal{P}.$$

Questa proprietà è ragionevole, infatti non vogliamo considerare sistemi troppo potenti, che vanno oltre \mathcal{P} , o poco potenti, che sono sotto \mathcal{P} . Vogliamo la giusta potenza computazionale.

13.1.2.2. Interprete universale

La seconda proprietà che vogliamo in un sistema di programmazione riguarda la presenza di un **interprete universale**. Un interprete universale è un programma $\mu \in \mathbb{N}$ tale che

$$\forall x, n \in \mathbb{N} \quad \varphi_\mu(\langle x, n \rangle) = \varphi_n(x).$$

In sostanza è un programma scritto in un certo linguaggio, che riesce a interpretare ogni altro programma n scritto nello stesso linguaggio, su qualsiasi input x .

La presenza di un interprete universale permette un'**algebra** sui programmi, quindi permette la trasformazione di quest'ultimi.

13.1.2.3. Teorema S_1^1

L'ultima proprietà che vogliamo in un sistema di programmazione riguarda il soddisfacimento del teorema S_1^1 . Questo teorema afferma che è possibile costruire automaticamente programmi specifici da programmi più generali, ottenuti fissando alcuni degli input.

Supponiamo di avere

$$P \in \text{PROG} : \varphi_P(< x, y >) = x + y.$$

Un programma RAM per questa funzione potrebbe essere

$$\begin{aligned} P &\equiv R_2 \leftarrow \sin(R_1) \\ R_3 &\leftarrow \text{des}(R_1) \\ R_0 &\leftarrow R_2 + R_3 \quad . \end{aligned}$$

Siamo in grado di produrre automaticamente un programma \overline{P} che riceve in input solo x e calcola, ad esempio, $x + 3$ a partire da P e 3 ?

$$(P, 3) \rightsquigarrow S_1^1 \in \text{PROG} \rightsquigarrow \overline{P}.$$

Per generare \overline{P} , potrei ad esempio fare

$$\begin{aligned} \overline{P} &\equiv R_0 \leftarrow R_0 + 1 \\ R_0 &\leftarrow R_0 + 1 \\ R_0 &\leftarrow R_0 + 1 \\ R_1 &\leftarrow < R_1, R_0 > \\ R_0 &\leftarrow 0 \\ P &\quad . \end{aligned}$$

Vediamo come questo programma segua principalmente quattro fasi:

1. si fissa il valore y in R_0 ;
2. si calcola l'input $< x, y >$ del programma P ;
3. si resetta la memoria alla situazione iniziale, tranne per il registro R_1 ;
4. si chiama il programma P .

In generale, il programma S_1^1 implementa la funzione

$$S_1^1(n, y) = \overline{n},$$

con n codifica di P e \overline{n} codifica del nuovo programma \overline{P} , tale che

$$\varphi_{\overline{n}}(x) = \varphi_n(< x, y >).$$

Questo teorema è molto comodo perché permette di calcolare facilmente la codifica \overline{n} : avendo n devo solo codificare le istruzioni iniziali di fissaggio di y , la funzione coppia di Cantor per creare l'input e l'azzeramento dei registri utilizzati. In poche parole,

$$S_1^1(n, y) = \overline{n} = < \underbrace{0, \dots, 0}_y, s, t, n > ,$$

con s codifica dell'istruzione che calcola la funzione coppia di Cantor e t codifica dell'istruzione di azzeramento.

S_1^1 è una funzione totale e programmabile, quindi $S_1^1 \in \mathcal{T}$ (funzione **ricorsiva totale**).

In sintesi, per RAM, esiste una funzione S_1^1 **ricorsiva totale** che accetta come argomenti

1. il codice n di un programma che ha 2 input;
2. un valore y cui fissare il secondo input

e produce il codice $\bar{n} = S_1^1(n, y)$ di un programma che si comporta come n nel caso in cui il secondo input è fissato ad essere y .

Teorema Dato φ_i sistema RAM, esiste una funzione $S_1^1 \in \mathcal{T}$ tale che

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

Questo teorema ci garantisce un modo di usare l'algebra sui programmi.

Inoltre, ha anche una forma generale S_n^m che riguarda programmi a $m + n$ input in cui si prefissano n input e si lasciano variare i primi m .

Teorema Dato φ_i sistema RAM, esiste una funzione $S_n^m \in \mathcal{T}$ tale che per ogni programma $k \in \mathbb{N}$, $\underline{x} \in \mathbb{N}^m$ e $\underline{y} \in \mathbb{N}^n$ vale

$$\varphi_k(< \underline{x}, \underline{y} >) = \varphi_{S_n^m(k, \underline{y})}(< \underline{x} >).$$

13.1.3. Sistemi di programmazione accettabili (SPA)

Le tre caratteristiche che abbiamo identificato formano gli **assiomi di Rogers** (1953). Questi caratterizzano i sistemi di programmazioni su cui ci concentreremo che chiameremo *Sistemi di Programmazione Accettabili*.

Questi assiomi non sono restrittivi: tutti i modelli di calcolo ragionevoli sono di fatto SPA.

13.1.4. Esistenza di compilatori tra SPA

Sappiamo che esiste un compilatore da WHILE a RAM, *ma è l'unico?*

Dati i SPA $\{\varphi_i\}$ e $\{\Psi_i\}$, un compilatore dal primo al secondo è una funzione $t : \mathbb{N} \rightarrow \mathbb{N}$ che soddisfa le proprietà di:

1. **programmabilità**: esiste un programma che implementa t ;
2. **completezza**: t compila ogni $i \in \mathbb{N}$;
3. **correttezza**: $\forall i \in \mathbb{N}$ vale $\varphi_i = \Psi_{t(i)}$.

I primi due punti ci dicono che $t \in \mathcal{T}$.

Teorema Dati due SPA, esiste sempre un compilatore tra essi.

Dimostrazione

Consideriamo $\{\varphi_i\}$ e $\{\Psi_i\}$ due SPA. Valgono i tre assiomi di Rogers:

1. $\{\varphi_i\} = \mathcal{P}$;
2. $\exists u : \varphi_u(< x, n >) = \varphi_n(x)$;
3. $\exists S_1^1 \in \mathcal{T} : \varphi_n(< x, y >) = \varphi_{S_1^1(e, i)}(x)$;

Voglio trovare un compilatore $t \in \mathcal{T}$ che sia corretto. Ma allora

$$\varphi_i(x) \stackrel{(2)}{=} \varphi_u(\langle x, i \rangle) \stackrel{(1)}{=} \Psi_e(\langle x, i \rangle) \stackrel{(3)}{=} \Psi_{S_1^1(e,i)}(x)$$

In poche parole, il compilatore cercato è la funzione $t(i) = S_1^1(e, i)$ per ogni $i \in \mathbb{N}$.

Infatti:

1. $t \in \mathcal{T}$ in quanto $S_1^1 \in \mathcal{T}$;
2. t corretto perché $\varphi_i = \Psi_{t(i)}$.

□

Notiamo la portata molto generale del teorema: non ci dice quale è il compilatore, ma ci dice che sicuramente esiste.

Corollario Dati gli SPA A, B, C esiste sempre un compilatore da A a B scritto nel linguaggio C .

Dimostrazione

Per il teorema precedente esiste un compilatore $t \in \mathcal{T}$ da A a B .

C è un SPA, quindi contiene programmi per tutte le funzioni ricorsive parziali, dunque ne contiene uno anche per t , che è una funzione ricorsiva totale.

□

In pratica, ciò vuol dire che per qualunque coppia di linguaggi, esistenti o che verranno progettati in futuro, sarò sempre in grado di scrivere un compilatore tra essi nel linguaggio che più preferisco. È un risultato assolutamente generale.

14. Lezione 14

14.1. Teorema di Rogers

Un risultato più potente del teorema precedente è dato dal **teorema di Rogers**.

Teorema (*Teorema di isomorfismo tra SPA*) Dati due SPA $\{\varphi_i\}$ e $\{\Psi_i\}$, esiste $t : \mathbb{N} \rightarrow \mathbb{N}$ tale che:

1. $t \in \mathcal{T}$;
2. $\forall i \in \mathbb{N} \quad \varphi_i = \Psi_{t(i)}$;
3. t è invertibile, quindi t^{-1} può essere visto come un decompilatore.

I primi due punti sono uguali al teorema precedente e ci dicono che il compilatore t è programmabile e completo (punto 1) e corretto (punto 2).

14.2. Introduzione

Vogliamo trovare dei risultati che abbiano la stessa portata dei precedenti, quindi che valgano per *tutti i sistemi di programmazione accettabili*.

14.3. Esistenza di certi programmi negli SPA

Ci poniamo due quesiti riguardo gli SPA:

1. **programmi auto-replicanti**: dato un SPA, *esiste all'interno di esso un programma che stampa se stesso (il proprio listato)?*

Ovviamente, questa operazione deve essere fatta senza aprire il file che contiene il listato.

Questi programmi sono detti **Quine**, in onore del filosofo e logico Willard Quine (1908-2000) che li descrisse per la prima volta.

La risposta è positiva per molti linguaggi: ad esempio, in Python il programma

```
| a='a=%r;print(a%%a)';print(a%a)
```

stampa esattamente il proprio listato. Noi, però, vogliamo rispondere tramite una dimostrazione rigorosa, quindi ambientiamo la domanda nel sistema di programmazione RAM, che diventa

$$\exists j \in \mathbb{N} \mid \varphi_j(x) = j \text{ per ogni input } x \in \mathbb{N}?$$

2. **compilatori completamente errati**: dati due SPA $\{\varphi_i\}$ e $\{\Psi_j\}$, *esiste un compilatore completamente errato?*

Un compilatore dal primo SPA al secondo SPA è una funzione $t : \mathbb{N} \rightarrow \mathbb{N}$ tale che:

- $t \in \mathcal{T}$ programmabile e totale;
- $\forall i \in \mathbb{N} \quad \varphi_i = \Psi_{t(i)}$ corretta.

Invece, un *compilatore completamente errato* è una funzione $t : \mathbb{N} \rightarrow \mathbb{N}$ tale che:

- $t \in \mathcal{T}$ programmabile e totale;
- $\forall i \in \mathbb{N} \quad \varphi_i \neq \Psi_{t(i)}$ errata.

14.4. Teorema di ricorsione

14.4.1. Definizione

Il **teorema di ricorsione** ci fornisce una risposta precisa a entrambi i quesiti che ci siamo posti.

Teorema Dato un SPA $\{\varphi_i\}$, per ogni $t : \mathbb{N} \rightarrow \mathbb{N}$ ricorsiva totale vale

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

Diamo una chiave di lettura a questo teorema:

- consideriamo t come un programma che prende in input un programma n e lo cambia nel programma $t(n)$, anche nella maniera più assurda;
- il teorema dice che qualsiasi sia la natura di t , esisterà sempre almeno un programma il cui significato **non sarà stravolto** da t .

Prima di vedere la sua dimostrazione, torniamo a considerare le due domande che ci siamo posti poco fa e vediamo la risposta usando proprio il teorema di ricorsione.

14.4.2. Primo quesito: Quine

Consideriamo il programma RAM

$$\begin{aligned} P &\equiv R_0 \leftarrow R_0 + 1 \\ &\quad R_0 \leftarrow R_0 + 1 \\ &\quad \dots \\ &\quad R_0 \leftarrow R_0 + 1 \end{aligned}$$

che ripete l'istruzione di incremento di R_0 un numero j di volte. La semantica di questo programma è esattamente j : infatti, dopo la sua esecuzione avremo j nel registro di output R_0 .

Calcoliamo la codifica di P come

$$\text{cod}(P) = \underbrace{\langle 0, \dots, 0 \rangle}_{j\text{-volte}} = Z(j) \in \mathcal{T}.$$

Questa funzione è ricorsiva totale in quanto programmabile e totale, visto che sfrutta solo la funzione di Cantor. Vale quindi

$$\varphi_{Z(j)}(x) = j.$$

Per il teorema di ricorsione

$$\exists j \in \mathbb{N} \mid \varphi_j(x) = \varphi_{Z(j)}(x) = j,$$

quindi effettivamente esiste un programma j la cui semantica è proprio quella di stampare sé stesso.

La risposta alla prima domanda è SI per RAM, ma lo è in generale per tutti gli SPA che ammettono una codifica per i propri programmi.

14.4.3. Secondo quesito: compilatori completamente errati

Supponiamo di avere in mano una funzione $t \in \mathcal{T}$ che “maltratta” i programmi.

Vediamo la semantica del programma “maltrattato” $t(i)$:

$$(*) \quad \Psi_{t(i)}(x) \stackrel{(2)}{=} \Psi_u(x, t(i)) \stackrel{(1)}{=} \varphi_e(x, t(i)) \stackrel{(3)}{=} \varphi_{S_1^1(e, t(i))}(x).$$

Chiamiamo $g(i)$ la funzione $S_1^1(e, t(i))$ che dipende solo da i , essendo e un programma fissato. Notiamo come questa funzione sia composizione di funzioni ricorsive totali, ovvero $t(i)$ per ipotesi e S_1^1 per definizione, quindi anch'essa è ricorsiva totale.

Per il teorema di ricorsione

$$(*) \quad \exists i \in \mathbb{N} \mid \varphi_i = \varphi_{g(i)}.$$

Unendo i risultati $(*)$ e $(**)$, otteniamo

$$\exists i \in \mathbb{N} \mid \Psi_{t(i)} \stackrel{(*)}{=} \varphi_{g(i)} \stackrel{(**)}{=} \varphi_i \quad \forall t \in \mathcal{T}.$$

Di conseguenza, la risposta alla seconda domanda è *NO*.

14.4.4. Dimostrazione

Dimostrazione

Siamo in un SPA $\{\varphi_i\}$ quindi valgono i tre assiomi di Rogers.

D'ora in avanti, per semplicità, scriveremo $\varphi_n(x, y)$ al posto di $\varphi_n(< x, y >)$.

Dobbiamo esibire, data una funzione t , uno specifico valore di n .

Partiamo con il mostrare che

$$\varphi_{\varphi_i(i)}(x) \stackrel{(2)}{=} \varphi_{\varphi_u(i, i)}(x) \stackrel{(2)}{=} \varphi_u(x, \varphi_u(i, i)) \rightsquigarrow f(x, i) \in \mathcal{P}.$$

Infatti, la funzione $f(x, i)$ è composizione di funzioni ricorsive parziali, quindi anch'essa lo è.

Continuiamo affermando che

$$f(x, i) \stackrel{(1)}{=} \varphi_e(x, i) \stackrel{(3)}{=} \varphi_{S_1^1(e, i)}(x).$$

Consideriamo ora la funzione $t(S_1^1(e, i))$: essa è ricorsiva totale in i perché composizione di t e di S_1^1 ricorsive totali, quindi

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

Abbiamo quindi mostrato che

$$(A) \quad \varphi_{\varphi_i(i)}(x) = \varphi_{S_1^1(e, i)}(x);$$

$$(B) \quad \varphi_m(i) = t(S_1^1(e, i)).$$

Fissiamo $n = S_1^1(e, m)$ e mostriamo che vale $\varphi_n = \varphi_{t(n)}$, ovvero il teorema di ricorsione.

$$\varphi_n(x) \stackrel{\text{def}}{=} \varphi_{S_1^1(e, m)}(x) \stackrel{(A)}{=} \varphi_{\varphi_m(m)}(x).$$

$$\varphi_{t(n)}(x) \stackrel{\text{def}}{=} \varphi_{t(S_1^1(e, m))}(x) = \varphi_{\varphi_m(m)}(x).$$

Ho ottenuto lo stesso risultato, quindi il teorema è verificato. □

Le conseguenze di questo teorema sono le due proprietà mostrate in precedenza.

14.5. Equazioni su SPA

14.5.1. Strategia

La portata del teorema di ricorsione è molto ampia: infatti, ci permette di risolvere **equazioni su SPA** in cui si chiede l'esistenza di certi programmi in SPA.

Ad esempio, dato uno SPA $\{\varphi_i\}$ ci chiediamo se

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x(n + \varphi_{\varphi_n(0)}(x))?$$

La **strategia** da seguire per risolvere questo tipo di richieste è analoga a quella usata per la dimostrazione del teorema di ricorsione e può essere riassunta nei seguenti passaggi:

1. trasforma il membro di destra dell'equazione in una funzione $f(x, n)$;
2. mostra che $f(x, n)$ è ricorsiva parziale e quindi che $f(x, n) = \varphi_e(x, n)$;
3. l'equazione iniziale diventa $\varphi_n(x) = \varphi_e(x, n) = \varphi_{S_1^1(e, n)}(x)$;
4. so che $S_1^1(e, n)$ è una funzione ricorsiva totale;
5. il quesito iniziale è diventato $\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_{S_1^1(e, n)}(x)$?
6. la risposta è *SI* per il teorema di ricorsione.

Riprendiamo in mano l'esempio appena fatto.

Cominciamo con il trasformare la parte di destra:

$$\begin{aligned}\varphi_n(x) &\stackrel{(2)}{=} \varphi_x(n + \varphi_{\varphi_u(0, n)}(x)) \\ &\stackrel{(2)}{=} \varphi_x(n + \varphi_u(x, \varphi_u(0, n))) \\ &\stackrel{(2)}{=} \varphi_u(n + \varphi_u(x, \varphi_u(0, n)), x) \\ &= f(x, n) \in \mathcal{P}.\end{aligned}$$

L'ultimo passaggio è vero perché $\varphi_u(n + \varphi_u(x, \varphi_u(0, n)), x)$ compone solamente funzioni ricorsive parziali quali somma e interprete universale. Di conseguenza, esiste un programma e che calcoli la funzione $f(x, n)$.

Continuando, riscriviamo l'equazione come

$$\varphi_n(x) = f(x, n) \stackrel{(1)}{=} \varphi_e(x, n) \stackrel{(3)}{=} \varphi_{S_1^1(e, n)}(x),$$

con $S_1^1(e, n) \in \mathcal{T}$ per l'assioma 3.

Per il teorema di ricorsione possiamo concludere che

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_{S_1^1(e, n)}(x) = \varphi_x(n + \varphi_{\varphi_u(0, n)}(x)).$$

14.5.2. Esercizi

In tutti gli esercizi viene dato un SPA $\{\varphi_i\}$.

14.5.2.1. Esercizio 01

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x(n) + \varphi_{\varphi_x(n)}(n)?$$

$$\begin{aligned}\varphi_n(x) &\stackrel{(2)}{=} \varphi_u(n, x) + \varphi_{\varphi_u(n, x)}(n) \\ &\stackrel{(2)}{=} \varphi_u(n, x) + \varphi_u(n, \varphi_u(n, x)) \\ &= f(x, n) \\ &\stackrel{(1)}{=} \varphi_e(x, n) \stackrel{(3)}{=} \varphi_{S_1^1(e, n)}(x) \\ &\stackrel{\text{TR}}{=} \text{OK}.\end{aligned}$$

14.5.2.2. Esercizio 02

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x(x) + n?$$

$$\begin{aligned}
\varphi_n(x) &\stackrel{(2)}{=} \varphi_u(x, x) + n \\
&= f(x, n) \\
&\stackrel{(1)}{=} \varphi_e(x, n) \stackrel{(3)}{=} \varphi_{S_1^1(e, n)}(x) \\
&\stackrel{\text{TR}}{=} \text{OK} .
\end{aligned}$$

14.5.2.3. Esercizio 03

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x(< n, \varphi_x(1) >)?$$

$$\begin{aligned}
\varphi_n(x) &\stackrel{(2)}{=} \varphi_u(< n, \varphi_u(1, x) >, x) \\
&= f(x, n) \\
&\stackrel{(1)}{=} \varphi_e(x, n) \stackrel{(3)}{=} \varphi_{S_1^1(e, n)}(x) \\
&\stackrel{\text{TR}}{=} \text{OK} .
\end{aligned}$$

14.5.2.4. Esercizio 04

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_{\varphi_x(\sin(n))}(\text{des}(n))?$$

$$\begin{aligned}
\varphi_n(x) &\stackrel{(2)}{=} \varphi_{\varphi_u(\sin(n), x)}(\text{des}(n)) \\
&\stackrel{(2)}{=} \varphi_u(\text{des}(n), \varphi_u(\sin(n), x)) \\
&= f(x, n) \\
&\stackrel{(1)}{=} \varphi_e(x, n) \stackrel{(3)}{=} \varphi_{S_1^1(e, n)}(x) \\
&\stackrel{\text{TR}}{=} \text{OK} .
\end{aligned}$$

14.5.2.5. Esercizio 05

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = n^x + (\varphi_x(x))^2?$$

$$\begin{aligned}
\varphi_n(x) &\stackrel{(2)}{=} n^x + (\varphi_u(x, x))^2 \\
&= f(x, n) \\
&\stackrel{(1)}{=} \varphi_e(x, n) \stackrel{(3)}{=} \varphi_{S_1^1(e, n)}(x) \\
&\stackrel{\text{TR}}{=} \text{OK} .
\end{aligned}$$

14.5.2.6. Esercizio 06

$$\exists n \in \mathbb{N} \mid \varphi_n(x) = \varphi_x(n+2) + \left(\varphi_{\varphi_x(n)}(n+3) \right)^2?$$

$$\begin{aligned}
\varphi_n(x) &\stackrel{(2)}{=} \varphi_u(n+2, x) + \left(\varphi_{\varphi_u(n, x)}(n+3) \right)^2 \\
&\stackrel{(2)}{=} \varphi_u(n+2, x) + (\varphi_u(n+3, \varphi_u(n, x)))^2 \\
&= f(x, n) \\
&\stackrel{(1)}{=} \varphi_e(x, n) \stackrel{(3)}{=} \varphi_{S_1^1(e, n)}(x) \\
&\stackrel{\text{TR}}{=} \text{OK} .
\end{aligned}$$

15. Lezione 15

15.1. Problemi di decisione

15.1.1. Introduzione

Un **problema di decisione** è una domanda cui devo decidere se rispondere *SI* o *NO*.

I problemi di decisione sono costituiti da tre elementi principali:

- **nome**: il nome del problema;
- **istanza**: dominio degli oggetti che verranno considerati;
- **domanda**: proprietà che gli oggetti del dominio possono soddisfare o meno.
Dato un oggetto del dominio, devo rispondere *SI* se soddisfa la proprietà, *NO* altrimenti.
In altre parole, è la **specificità** del problema di decisione.

Diamo ora una definizione più formale:

- **nome**: Π ;
- **istanza**: $x \in D \rightarrow \text{input}$;
- **domanda**: $p(x) \rightarrow \text{proprietà che } x \in D \text{ può soddisfare o meno}$.

Se, per esempio, mi viene chiesto “*questo polinomio ha uno zero?*” non devo dire *quale* zero ha, ma solo se lo ha o meno. Non devo esibire una **struttura** come risultato (cosa che avviene nei problemi di ricerca o di ottimizzazione), ma solo una risposta *SI/NO*.

15.1.2. Esempi

Vediamo qualche esempio di problemi di decisione noti. L’assegnamento di un valore a $x \in D$ genera un’**istanza particolare** del problema:

1. Parità

- Nome: Parità.
- Istanza: $n \in \mathbb{N}$.
- Domanda: n è pari?

2. Equazione diofantea

- Nome: Equazione Diofantea.
- Istanza: $a, b, c \in \mathbb{N}^+$.
- Domanda: $\exists x, y \in \mathbb{Z} \mid ax + by = c$?

Questa domanda nei reali avrebbe poco senso, perché ci sarebbero infiniti punti che soddisfano l’equazione della retta. Considerando punti interi, invece, ha più senso in quanto niente garantisce che la retta ne abbia.

Ad esempio, per $a = 3$, $b = 4$ e $c = 5$ rispondo *SI*, visto che la proprietà vale per $x = -1$ e $y = 2$.

Il nome di queste equazioni deriva dal matematico Diofanto, che per primo le trattò nel contesto dell’aritmetica di, appunto, Diofanto.

3. Fermat

- Nome: Ultimo Teorema di Fermat.
- Istanza: $n \in \mathbb{N}^+$.
- Domanda: $\exists x, y, z \in \mathbb{N}^+ \mid x^n + y^n = z^n$?

Questo problema è, in un certo senso, riconducibile al precedente.

Per $n = 1$ rispondo *SI*: è facile trovare tre numeri tali che $x + y = z$, ne ho infiniti.

Per $n = 2$ rispondo *SI*, i numeri nella forma $x^2 + y^2 = z^2$ rappresentano le **terne pitagoriche**.

Per $n \geq 3$ risponde **NO**, è stato dimostrato da Eulero.

Questo problema è rimasto irrisolto per circa 400 anni, fino a quando nel 1994 viene dimostrato il **teorema di Andrew-Wiles** (dall'omonimo matematico), come banale conseguenza di una dimostrazione sulla modularità delle curve ellittiche.

Si dice che il primo a risolvere questo problema sia stato Fermat, giurista che nel tempo libero giocava con la matematica, tanto da meritarsi il nome di *principe dei dilettanti*, lo dimostra il fatto che questo teorema non ha nessuna conseguenza pratica, è totalmente “inutile”.

4. Raggiungibilità

- Nome: Raggiungibilità.
- Istanza: grafo $G = (\{1, \dots, n\}, E)$.
- Domanda: $\exists \pi$ cammino dal nodo 1 al nodo n ?

5. Circuito hamiltoniano

- Nome: Circuito Hamiltoniano.
- Istanza: grafo $G = (V, E)$.
- Domanda: $\exists \gamma$ circuito hamiltoniano nel grafo G ?

Un **circuito hamiltoniano** è un circuito che coinvolge ogni nodo una e una sola volta.

6. Circuito euleriano

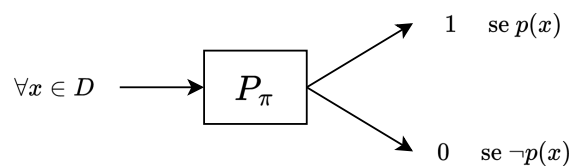
- Nome: Circuito Euleriano.
- Istanza: grafo $G = (V, E)$.
- Domanda: $\exists \gamma$ circuito euleriano nel grafo G ?

Un **circuito euleriano** è un circuito che coinvolge ogni arco una e una sola volta.

Questo quesito ha dato via alla teoria dei grafi.

15.1.3. Decidibilità

Sia Π problema di decisione con istanza $x \in D$ e domanda $p(x)$. Π è **decidibile** se e solo se esiste un programma P_Π tale che



Allo stesso modo, possiamo associare a Π la sua **funzione soluzione**:

$$\Phi_\Pi : D \longrightarrow \{0, 1\}$$

tale che

$$\Phi_\Pi(x) = \begin{cases} 1 & \text{se } p(x) \\ 0 & \text{se } \neg p(x) \end{cases}$$

Questa funzione deve essere sicuramente ricorsiva totale, perché deve essere programmabile e deve terminare sempre:

$$\Phi_\Pi \in \mathcal{T}.$$

I due approcci per definire la decidibilità sono equivalenti:

- il programma P_{Π} calcola $\Phi_{\Pi} \Rightarrow \Phi_{\Pi} \in \mathcal{T}$;
- se $\Phi_{\Pi} \in \mathcal{T}$ allora esiste un programma che la calcola e che ha il comportamento di P_{Π} .

Quindi, definire la decidibilità partendo da un programma o da una funzione ricorsiva parziale, è indifferente: una definizione implica l'altra.

Possiamo sfruttare questa cosa per sviluppare due tecniche di risoluzione del problema Decidibilità:

1. esibiamo un algoritmo di soluzione P_{Π} (anche inefficiente, basta che esista);
2. mostriamo che Φ_{Π} è ricorsiva totale.

15.1.3.1. Applicazione agli esempi

1. Parità:

$$\Phi_{PR}(n) = 1 \div (n \bmod 2) \in \mathcal{T}.$$

2. Equazione diofantea:

$$\Phi_{ED}(a, b, c) = 1 \div (c \bmod \text{mcd}(a, b)) \in \mathcal{T}.$$

3. Fermat:

$$\Phi_F(n) = 1 \div (n \div 2) \in \mathcal{T}.$$

Tipicamente, per problemi sui grafi si preferisce fornire degli algoritmi di decisione.

4. Raggiungibilità:

Sia $M_G \in \{0, 1\}^{n \times n}$ matrice di adiacenza tale che $M_G[i, j] = 1$ sse $(i, j) \in E$. Inoltre, M_G^k ha un 1 nella cella $[i, j]$ sse esiste un cammino lungo k da i a j .

$$\Phi_R(G) = \left(\bigvee_{k=0}^n M_G^k \right) [1, n] \in \mathcal{T}.$$

5. Circuito hamiltoniano:

Dovendo visitare ogni nodo una e una sola volta, il circuito genera una permutazione dei vertici in V . L'algoritmo di soluzione deve:

1. generare l'insieme P di tutte le permutazioni di V ;
2. data la permutazione $p_i \in P$, se è un circuito hamiltoniano rispondo *SI*;
3. se nessuna permutazione p_i è un circuito hamiltoniano rispondo *NO*.

L'algoritmo è inefficiente perché ci mette un tempo $O(n!)$, vista la natura combinatoria del problema, ma sicuramente questo problema è decidibile.

6. Circuito euleriano

Teorema (Teorema di Eulero (1936)) Un grafo $G = (V, E)$ contiene un circuito euleriano se e solo se ogni vertice in G ha grado pari.

Grazie a questo risultato, l'algoritmo di risoluzione deve solo verificare se il grado di ogni vertice in V è pari, quindi anche questo problema è decidibile.

15.1.4. Problemi indecidibili

Ma esistono dei **problemi indecidibili**?

PEFFORZA, se esistono programmi che non so scrivere allora esistono problemi per i quali non riesco a scrivere dei programmi che li risolvano.

15.1.4.1. Problema dell'arresto ristretto

Il **problema dell'arresto ristretto** per un programma P è un esempio di problema indecidibile.

Fissato P un programma, il problema è il seguente:

- Nome: AR_P .
- Istanza: $x \in \mathbb{N}$.
- Domanda: $\varphi_P(x) \downarrow?$

In altre parole, ci chiediamo se il programma P termina su input x .

La risposta dipende ovviamente dal programma P , che può essere decidibile o non decidibile.

Ad esempio, se

$$\begin{aligned} P_1 &\equiv \text{input}(x) \\ &\quad x := x + 1; \\ &\quad \text{output}(x) \end{aligned}$$

allora la funzione

$$\Phi_{\text{AR}_{P_1}}(x) = 1 \in \mathcal{T}$$

ci dice quando il problema P_1 termina o meno (*sempre*).

Se invece

$$\begin{aligned} P_2 &\equiv \text{input}(x) \\ &\quad \text{if } (x \bmod 2 \neq 0) \\ &\quad \quad \text{while } (1 > 0); \\ &\quad \text{output}(x) \end{aligned}$$

allora la funzione

$$\Phi_{\text{AR}_{P_2}}(x) = 1 \div (x \bmod 2) \in \mathcal{T}$$

ci dice quando il problema P_2 termina o meno.

Abbiamo quindi trovato due funzioni $\Phi_{\text{AR}_P} \in \mathcal{T}$ che ci dicono quando i programmi P_1, P_2 terminano o meno, *ma è sempre possibile?*

Prendiamo ora

$$\begin{aligned} \S \\ P &\equiv \text{input}(x) \\ &\quad z := U(x, x); \\ &\quad \text{output}(z), \end{aligned}$$

con U interprete universale tale che

$$\varphi_U(x, n) = \varphi_x(x).$$

Vale allora

$$\varphi_{\frac{\S}{P}}(x) = \varphi_U(x, x) = \varphi_x(x).$$

Abbiamo, quindi, un programma x che lavora su un altro programma (se stesso). Questo non è strano: compilatori, debugger, interpreti sono programmi che lavorano su programmi.

Come prima mi chiedo se φ_x su input x termina. A differenza di prima, ora il programma P non è fissato e dipende dall'input, essendo x sia input che programma.

Teorema Dato

$$\begin{aligned} \frac{\S}{P} &\equiv \text{input}(x) \\ z &:= U(x, x); \\ \text{output}(z), \end{aligned}$$

$\text{AR}_{\frac{\S}{P}}$ è indecidibile.

Dimostrazione

Per assurdo assumiamo $\text{AR}_{\frac{\S}{P}}$ decidibile.

Dunque esiste

$$\Phi_{\text{AR}_{\frac{\S}{P}}}(x) = \begin{cases} 1 & \text{se } \varphi_{\frac{\S}{P}}(x) = \varphi_x(x) \downarrow \\ 0 & \text{se } \varphi_{\frac{\S}{P}}(x) = \varphi_x(x) \uparrow \end{cases} \in \mathcal{T}$$

calcolabile da un programma che termina sempre.

Visto che $\Phi_{\frac{\S}{P}} \in \mathcal{T}$, anche la funzione

$$f(x) = \begin{cases} 0 & \text{se } \Phi_{\text{AR}_{\frac{\S}{P}}}(x) = 0 \equiv \varphi_x(x) \uparrow \\ \varphi_x(x) + 1 & \text{se } \Phi_{\text{AR}_{\frac{\S}{P}}}(x) = 1 \equiv \varphi_x(x) \downarrow \end{cases}$$

è ricorsiva totale. Infatti, il programma

$$\begin{aligned} A &\equiv \text{input}(x) \\ &\text{if } \left(\Phi_{\text{AR}_{\frac{\S}{P}}}(x) == 0 \right) \\ &\quad \text{output}(0) \\ &\text{else} \\ &\quad \text{output}(U(x, x) + 1) \end{aligned}$$

calcola esattamente la funzione $f(x)$.

Sia $\alpha \in \mathbb{N}$ la codifica del programma A , allora $\varphi_\alpha = f$. Valutiamo φ_α in α :

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

Tale funzione non può esistere, infatti:

- nel primo caso ho $\varphi_\alpha(\alpha) = 0$ se non termino \rightarrow contraddizione perché $A \in \mathcal{T}$;
- nel secondo caso ho $\varphi_\alpha(\alpha) = \varphi_\alpha(\alpha) + 1$ se termino \rightarrow contraddizione perché questa relazione non vale per nessun naturale.

Siamo ad un **assurdo**, quindi concludiamo che $AR_{\frac{s}{P}}$ deve essere **indecidibile**. \square

15.1.4.2. Problema dell'arresto

La versione generale del problema dell'arresto ristretto è il **problema dell'arresto**, posto nel 1936 da Alan Turing.

Teorema Dati $x, y \in \mathbb{N}$ rispettivamente un dato e un programma, il problema dell'arresto AR con domanda $\varphi_y(x) \downarrow$ è indecidibile.

Dimostrazione

Assumiamo per assurdo che AR sia decidibile.

Allora, esiste un programma $P_{AR}(x, y)$ che lo risolve, quindi restituisce 1 se $\varphi_y(x) \downarrow$, 0 altrimenti.

Il seguente programma decide $AR_{\frac{s}{P}}$:

$$P \equiv \text{input}(x) \\ \text{output}(P_{AR}(x, x)) \quad .$$

Il risultato precedente dimostra che questo problema è indecidibile, quindi non possono esistere programmi per la soluzione di AR. Siamo ancora ad un **assurdo**, quindi AR è **indecidibile**. \square