Indice

1 Numerabilità e funzioni **3**
- 1.1 Finito vs. Infinito . . . 3
- 1.2 Numerabilità . . . 4
  - 1.2.1 Slide 6 . . . 4
  - 1.2.2 Slide 7 . . . 5
  - 1.2.3 Slide 8 . . . 5
  - 1.2.4 Slide 9 . . . 5
- 1.3 Diagonalizzazione . . . 6
- 1.4 Definibilità vs. Calcolabilità . . . 7
  - 1.4.1 Paradosso di Russel . . . 8
  - 1.4.2 Paradosso di Berry . . . 8

2 Il formalismo primitivo ricorsivo **10**
- 2.1 Ricorsione primitiva . . . 10
  - 2.1.1 Slide 21 . . . 11
  - 2.1.2 Esempi di funzioni definibili nel formalismo primitivo ricorsivo 11
- 2.2 Test di primalità . . . 13
- 2.3 Minimizzazione . . . 14
- 2.4 Generazione numeri primi . . . 15
- 2.5 Ricorsione multipla . . . 15
  - 2.5.1 Slide 33 . . . 17
- 2.6 Funzione di Ackermann . . . 17
  - 2.6.1 Slide 37 . . . 17

3 Altri formalisimi e Macchine di Turing **19**
- 3.1 Formalismi totali e problema dell'interprete . . . 19
  - 3.1.1 Slide 47 . . . 19
  - 3.1.2 Slide 48 . . . 20
  - 3.1.3 Slide 46 . . . 20
- 3.2 Tesi di Church . . . 20
  - 3.2.1 Slide 50 . . . 20
- 3.3 Alcuni formalismi Turing completi . . . 21
  - 3.3.1 Slide 52 . . . 21
- 3.4 Macchine di Turing . . . 22

3.4.1 Slide 54 22
3.4.2 Slide 55 23
3.4.3 Slide 56 23
3.5 Macchine universali 25

4 Problemi indecidibili **27**
4.1 Note introduttive su programmi e funzioni 27
4.2 Il problema della fermata 27
4.3 Proprietà s-m-n 28
4.4 Il predicato T di Kleene 31

Capitolo 1

Numerabilità e funzioni

Questi appunti riguardano le lezioni introduttive sulla numerabilità.

1.1 Finito vs. Infinito

Siamo interessati a questa problematica. Perché? Se avessimo solo da calcolare funzioni di input finiti con output finiti non avremmo troppe difficoltà. I nostri programmi diventano interessanti quando andiamo a lavorare su degli input infiniti. Ad esempio, potremmo avere una struttura dati dinamica come input. Ancora, un programma che lavora su una lista dovrebbe ragionevolmente funzionare per tutte le liste. I nostri algoritmi dovrebbero essere in grado di accettare una certa quantità di input infiniti diversi.

L'infinito è anche interessante a livello di complessità computazionale. Nella parte di complessità computazionale guarderemo al comportamento asintotico del programma al crescere della dimensione dell'input.

Un'altra problematica interessante è la seguente: a volte l'input stesso dei nostri programmi è infinito. Ad esempio, il nostro programma potrebbe essere un automa per un linguaggio libero da contesto. Il nostro input, un linguaggio, può essere un insieme infinito.

(Figura 1 sul quaderno)

Come facciamo a dare al programma un input infinito? L'unico modo è fornire al programma una descrizione finita dell'input. Creiamo quindi una struttura generativa finita che, da un insieme finito di oggetti, ci permette di crearne dinamicamente di nuovi.

Questo esempio ci mostra come in questo ambito abbiamo un doppio livello di discussione:

- uno descrittivo: a questo livello parlo di quello che mi serve per descrivere ciò di cui voglio parlare (e.g. la grammatica per un linguaggio). Vi sono una serie di problematiche legate a questo livello: la descrizione che do è giusta? È unica? Ce n'è una canonica? Ad esempio, date due grammatiche posso decidere se queste definiscono lo stesso linguaggio?

- uno denotazionale: a questo livello parlo veramente del mio oggetto in questione, a prescindere da qualsiasi descrizione se ne possa dare (e.g. un linguaggio). Abbiamo le stesse problematiche già viste per il livello descrittivo

Il livello descrittivo è anche detto intensionale, quello denotazionale invece estensionale.

Anche per i programmi vale questa distinzione: la funzione calcolata da un programma fa parte del livello denotazionale. Una funzione è descritta mediante il suo grafo. Non avendo modi diretti, ovvero finiti, di descrivere la funzione usiamo un programma, che fa parte del livello descrittivo. Una problematica interessante è la seguente: esiste una maniera di descrivere una funzione che non mi dia anche un metodo effettivo per calcolarla (non un programma insomma)?

Siamo costretti ad avere questi due tipi di livelli? Sì se l'oggetto di cui vogliamo parlare è infinito. La comunicazione presuppone una descrizione finita dell'oggetto infinito.

Tutta la nostra intuizione è basata sul finito. Tuttavia molte cose che non valgono per il finito valgono per l'infinito. Ad esempio è possibile mettere in corrispondenza biunivoca un sottoinsieme stretto di un insieme infinito con l'insieme stesso, a differenza del caso finito. Questo è fuori dalla nostra intuizione immediata di insiemi.

A volte vogliamo fare ricerche infinite, e.g. su strutture dati infinite. Ad esempio potrei voler verificare che una stringa x appartenga ad un linguaggio $\mathcal{L}$ generando tutte le stringhe da una grammatica per $\mathcal{L}$ e decidere "Si"' se x compare. Ho un modo per dire se in una ricerca troverò quello che cerco? In generale no. Questo problema è fondamentale perché è frequentissimo. Parleremo in questi casi qui di semi-decisione. Notiamo inoltre che sul complementare, ovvero $x \notin \mathcal{L}$, non possiamo dire niente.

Noi considereremo funzioni da $\mathbb{N}$ a $\mathbb{N}$ o, al massimo, da $\mathbb{N}^k$ a $\mathbb{N}$. Questo perché i numeri naturali sono la più semplice struttura infinita. Questo non ci pone limitazioni di alcun tipo. Se noi siamo interessati ad aspetti di calcolabilità tutto è alla fine riconducibile ad un numero binario. Questa traduzione da oggetto qualsiasi ad un numero naturale prende il nome di Gödelizzazione. Ai tempi ($\sim$ 1930) fu un'idea rivoluzionaria, ma dal punto di vista di un informatico dovrebbe sembrare più naturale.

1.2 Numerabilità

Le seguenti sono note alle slide

1.2.1 Slide 6

La funzione f ci dà anche un metodo di enumerazione degli elementi del codominio di f.

1.2.2 Slide 7

Aggiungere un elemento non intacca la numerabilità di un insieme. Basta uno shift per avere una nuova enumerazione di A. La numerabilità è resistente all'operazione di estensione.

1.2.3 Slide 8

Dimostrazione del secondo corollario sul quaderno azzurro.

// DA RIPORTARE QUI

1.2.4 Slide 9

Possiamo enumerare $\mathbb{N} \times \mathbb{N}$? Vediamo com'è fatto questo insieme. Avremo $(0,0), (0,1), (0,2), \ldots$, dopodiché, sulla prossima riga, avremo $(1,0), (1,1), (1,2), \ldots$, e così via all'infinito. Il modo più conveniente per visualizzarlo è pensare a dei punti nel piano.

Possiamo enumerarli? Sarebbe sbagliato numerare per righe o per colonne. Questo perché le righe e le colonne sono infinite, iterando su una riga non passerò mai alla prossima. Che tecnica usiamo? Dove tailing. Numeriamo per diagonali. Ce ne sono altre di numerazioni possibili. Va bene qualsiasi "gioco dell'oca" che passa per ogni coppia una e una sola volta.

Importante: Il dove tailing non è la diagonalizzazione.

È facile dimostrare che posso avere delle biezioni tra $\mathbb{N}^k$ e $\mathbb{N}$, in modo da codificare delle tuple di numeri con un numero solo. In altre parole c'è una procedura algoritmica che mi permette di passare da una tupla al corrispondente numero n e da n alla corrispondente tupla. Di conseguenza $\mathbb{N}^k$, per k naturale, è ancora numerabile.

L'unione numerabile di insiemi numerabili è ancora numerabile.

$$\bigcup_{i \in \mathbb{N}} A_i = \{< i, a > |\ i \in \mathbb{N}, a \in A_i\}$$

Consideriamo l'operatore di Kleene sull'alfabeto Σ: Σ^*. Anche questo insieme è numerabile, perché è l'unione numerabile di insiemi numerabili (ricordiamo che A^k è numerabile).

Un altro insieme numerabile interessante è l'insieme delle parti finite:

$$\mathscr{P}_{fin}(\mathbb{N}) = \{A \mid A \subseteq \mathbb{N}, A \ \textit{è finito}\}$$

Abbiamo due piani di infinità nei programmi: l'infinità dell'input e l'infinità del tempo di calcolo, in quanto il mio programma può divergere su un dato input.

L'insieme di funzioni da A a B ha cardinalità B^A.

Possiamo caratterizzare l'insieme delle coppie su A, $A \times A = A^2$, come una funzione dall'insieme $\mathrm{BOOL} = \{0,1\}$ all'insieme A. $f(x)$ può essere così definita: data una coppia $< a_1, a_2 >$, $f(x)$: $x \to$ if x then a_1 else a_2. Al posto dei booleani potremmo avere un qualsiasi insieme di cardinalità 2.

A^K è isomorfo all'insieme delle funzioni $f : K \to A$.

La funzione è utile sia come strumento di calcolo che come strumento di codifica dei dati.

Lo spazio delle funzioni da un insieme finito ad un insieme numerabile è ancora numerabile.

1.3 Diagonalizzazione

Consideriamo lo spazio delle funzioni da un insieme numerabile ad un altro insieme numerabile. Ancora più semplicemente, possiamo considerare le funzioni da $\mathbb{N}$ a 2 (o BOOL).

L'insieme delle funzioni da A a BOOL è isomorfo all'insieme delle parti di A.

Per denotare un sottoinsieme si può dare una funzione caratteristica, che restituisce 1 se un elemento fa parte del sottoinsieme e 0 altrimenti.

Teorema 1.1. *Per ogni insieme A non esiste una funzione suriettiva da A in $\mathscr{P}(A)$.*

Dimostrazione. Sia $f : A \to \mathscr{P}(A)$ una funzione suriettiva da A all'insieme delle parti di A. Data f posso costruire parametricamnente Δ_f così fatto:

$$\Delta_f = \{a \mid a \in A, a \notin f(a)\}$$

Essendo f suriettiva esiste a tale che $f(a) = \Delta_f$. Ci chiediamo ora, $a \in f(a)$?

$$a \in f(a) \iff a \in \Delta_f \iff a \in A \wedge a \notin f(a)$$

Ma questo è assurdo. □

La tecnica con cui si dimostra questo teorema è detta tecnica diagonale, o diagonalizzazione. Un'idea intuitiva che giustifica il nome è la seguente: consideriamo una tabella indicizzata per colonne dagli $a \in A$ e sulle colonne dagli $f(a)$ per ogni $a \in A$. Nella cella (i, j) della tabella avrò 1 se $a_j \in f(a_i)$ e 0 altrimenti. Abbiamo per ogni riga la funzione caratteristica di $f(a)$, per ogni $a \in A$. Abbiamo ora che la funzione caratteristica di Δ_f è costruita andando sulla diagonale e prendendo l'elemento j in Δ se in corrispondenza sulla diagonale, ovvero alla riga i ho 0, lasciandolo fuori altrimenti. Di conseguenza la funzione caratteristica che vado a costruire per definire Δ è diversa da tutte le funzioni caratteristiche sulle righe almeno per un elemento.

La diagonalizzazione è un metodo semplice per creare un nuovo elemento diverso da tutti quelli di una lista sotto certe ipotesi.

Dimostriamo che i numeri nell'intervallo $[0, 1[$ sono una quantità non numerabile. Come possiamo descrivere i numeri reali? Possiamo farlo, ad esempio, con la loro rappresentazione decimale: una infinita successione delle cifre del numero, per ogni numero.

Supponiamo di avere una enumerazione dei numeri reali $r_0, r_1, r_2, \ldots$. Disegnamo una tabella con le righe indicizzate dai numeri reali nell'ordine dato

dall'enumerazione e con le colonne indicizzate dai numeri naturali. Per ogni numero della enumerazione possiamo scrivere, in corrispondenza della colonna j, la sua cifra j, in ordine da sinistra verso destra.

Consideriamo la diagonale della tabella. Consideriamo il mapping che per gli elementi dell'insieme $\{0, \ldots, 4\}$ restituisce 7 mentre per gli elementi dell'insieme $\{5, \ldots, 9\}$ restituisce 3. Se prendiamo la diagonale e alle sue cifre applichiamo il mapping creato otteniamo un numero che è diverso da tutti gli elementi dell'enumerazione almeno per la cifra sulla diagonale. Di conseguenza non fa parte dell'enumerazione. Ma questo è assurdo, dato che avevo supposto di poter enumerare tutti i numeri reali dell'intervallo $[0, 1[$.

Le funzioni da $\mathbb{N}$ a 2, a $\{0, \ldots, 9\}$, a $\mathbb{N}$ hanno la stessa cardinalità, quella del continuo, che non è numerabile.

Esistono quindi numeri reali per i quali non esiste una maniera per descriverli.

1.4 Definibilità vs. Calcolabilità

Le funzioni da $\mathbb{N}$ a $\mathbb{N}$ di cui possiamo parlare hanno cardinalità numerabile. Anche i programmi sono una quantità numerabile.

Non è però questo il problema che vogliamo affrontare. È evidente per ragioni di cardinalità che ci sono funzioni che non potremo calcolare. Il problema a cui siamo interessati è: tutte le funzioni che possiamo definire sono calcolabili o ci sono funzioni definibili ma non calcolabili?

Abbiamo inizialmente il problema di cosa significa definibile. La nozione di definibilità dipende dal contesto e dal potere espressivo del linguaggio che uso (e.g. linguaggio dell'aritmetica, linguaggio del secondo ordine, ecc.).

Possiamo formalizzare la nozione di definibilità? No. È possibile dimostrare che l'idea di definibilità non è definibile. Non è possibile dire quando una cosa è ben definita.

Supponiamo di avere un criterio che scansiona le stringhe e decide se una è una buona definizione o no. Possiamo quindi definire una sequenza di funzioni e quindi dare una numerazione delle funzioni definibili. Si può dimostrare che questa definizione di definibilità è incompleta, non cattura bene il nostro senso intuitivo di definibilità.

Intuitivamente, possiamo creare una tabella con le righe indicizzate dalle funzioni definite e per colonne dai numeri naturali. La casella (i, j) contiene il risultato della funzione f_i sul numero j. Supponiamo inoltre di avere funzioni binarie, ovvero che restituiscono 1 o 0. Questa semplificazione non fa perdere di generalità.

Con un procedimento diagonale possiamo definire una nuova funzione che non sta nell'enumerazione. Ad esempio, sia $f(n) = 1 - d_n(n)$. Se questa definizione intuitivamente valida non è presente nell'enumerazione allora il mio criterio è incompleto. Supponiamo che nella mia enumerazione la mia funzione f compaia in posizione m. Allora avremmo $d_m(m) = f(m) = 1 - d_m(m)$. Ma questo è assurdo.

Se fissiamo un linguaggio l'insieme delle funzioni definite su quel linguaggio è numerabile. Esiste quindi una funzione, per quel linguaggio, che mi dica se una definizione è valida. Questa funzione, tuttavia, non è calcolabile.

1.4.1 Paradosso di Russel

Questo è un vero paradosso, che mise in crisi la matematica agli inizi del XX secolo.

Possiamo pensare ad un insieme come ad una collezione di cose che rispetta certe proprietà. Data una proprietà $P(x)$ possiamo costruire un insieme che la rispetti:

$$\frac{P(t)}{t \in \{x \mid P(x)\}}$$

//TODO USE PROVING PACKAGE OR SOME OTHER PACKAGE FOR RULE INFERENCE Vale anche l'implicazione inversa

Pensiamo alla proprietà $P(x) = x \notin x$. Possiamo costruire l'insieme $U = \{x \mid x \notin x\}$. Ci chiediamo ora: $U \in U$? Abbiamo che $U \in U \iff U \notin U$. Ma questo è paradossale.

Il problema è che all'inizio del secolo si era tentato di basare la teoria insiemistica sul principio di comprensione. Per quanto comodo e bello questo, nella forma che abbiamo visto, è causa di paradossi. Dobbiamo rinunciare all'idea che basti pensare ad una proprietà per poter creare un insieme che la rispetti.

1.4.2 Paradosso di Berry

Berry era un bibiliotecario che si era appassionato un pò di matematica. C'è un principio importante nei numeri naturali che dice che dato un sottoinsieme finito dei numeri naturali esiste un elemento dei naturali che è il più piccolo numero naturale che non appartiene a questo sottoinsieme.

Supponiamo di definire i numeri con stringhe di lunghezza d. Abbiamo quindi un insieme di numeri che posso definire con al più d caratteri. Possiamo definire n_d come il più piccolo numero non definibile con meno di d caratteri, e sappiamo che esiste per il principio precedentemente riportato.

Prendiamo ad esempio $d = 100$. Abbiamo che n_{100} non è definibile con meno di 100 caratteri. Eppure è definito dalla stringa: "n_{100} non è definibile con meno di 100 caratteri", che ha meno di 100 caratteri. Questo è assurdo.

// DA RIVEDERE Più formalmente, supponiamo che la definizione sia data dalla stringa s e prendiamo $d_s = |s|$. n_{d_s} è il più piccolo numero non definibile con meno di d_s caratteri. Ma n_{d_s} è definito da s. Assurdo.

Ci sono paradossi e paradossi. Alcune sono cose che sembrano strane ma in realtà non lo sono così tanto. Altri sono proprio cattivi, mettono in questione aspetti fondamentali della nostra intuizione.

Dove sta il problema del paradosso di Berry? Nella stringa s, perché la nozione di definibilità non è definibile, ed s la usa. La definizione data non è una buona definizione.

Ci chiediamo: data una enumerazione θ delle sentenze dell'aritmetica è possibile generare una formula che mi dice il valore di verità di una sentenza dell'enumerazione?

Ad ogni sentenza dell'enumerazione associamo un numero, detto di Gödel. Questo perché non possiamo parlare delle sentenze in sè in aritmetica, ma possiamo parlare solo di numeri. Quindi ci serve un numero per parlare della sentenza.

La verità è intesa sul modello dei numeri naturali.

Vogliamo una formula *Vera* tale che

$$\mathcal{N} \models A \iff \mathcal{N} \models \mathit{Vera}(g(A))$$

dove $g(A)$ rappresenta il numero di Gödel di A.

Per i numeri naturali c'è un lemma detto lemma di diagonalizzazioe. Questo dice che dato un predicato è sempre possibile trovare una formula S tale che $\mathcal{N} \models S \iff \mathcal{N} \models P(g(S))$. È possibile, per una sentenza, dire "quel predicato P vale per me". La dimostrazione è interessante perché costruttiva.

Prendiamo come $P(x)$ la formula $\neg \mathit{Vero}(x)$: $\mathcal{N} \models P(x) \iff \mathcal{N} \models \neg \mathit{Vero}(x)$. Possiamo allora costruire S tale che $\mathcal{N} \models S \iff \mathcal{N} \models \neg Vero(S)$. Avremmo quindi $\mathcal{N} \models S \iff \mathcal{N} \models \neg S$. Ma questo è paradossale. Quindi non è definibile, nel linguaggio dell'aritmetica, una formula che, data una sentenza dell'aritmetica, mi dice se questa è vera, in senso aritmetico. Questo risultato è soprendentemente forte e va sotto il nome di teorema di Tarski.

Un'idea simile viene usata nel teorema di Gödel non con la nozione di verità matematica ma con la nozione di dimostrabilità. Il risultato è che la formula che afferma la propria indimostrabilità non è dimostrabile. Da ciò il sistema logico è incompleto, ovvero c'è una formula indimostrabile.

Capitolo 2

Il formalismo primitivo ricorsivo

2.1 Ricorsione primitiva

Nei linguaggi di programmazione siamo abituati all'autoreferenzialità, o ricorsione. In generale bisogna fare attenzione.

Esistono forme esplicite ed implicite di autoreferenziazione. Per le seconde ho un oggetto che ha una sezione universale nel suo scope e tale che l'oggetto stesso ci rientra. È una sorta di ordine superiore.

Ad esempio:

> "Tutte le sentenze universali sono false"

Questa frase è implicitamente autoreferenziale. Nel suo raggio di applicabilità include se stessa.

Per le formule aperte non ha senso parlare di verità. Si può parlare solo di verità in caso di formule chiuse, ovvero sentenze. I teoremi sono tutti chiusi. La sentenza sopra è dimostrabilmente falsa.

Ci chiediamo se la definibilità di una funzione implichi la sua calcolabilità e anche se vale il viceversa. Sappiamo già che la nozione di definibilità è legata al linguaggio che utilizzo.

Possiamo dare una definizione precisa di calcolabilità? È una problematica delicata, legata al sistema di calcolo che utilizzo e alla definizione formale di algoritmo, che non è banale.

Ci chiediamo, vogliamo davvero avere un loop infinito? In certi casi sì, ad esempio per stampare una lista di numeri primi ed interromperla quando voglio. Ma non è comune, molto spesso ci basta una iterazione determinata.

Le funzioni dell'informatica sono di una categoria particolare: sono funzioni parziali. Le funzioni totali calcolabili sono un sottoinsieme delle funzioni parziali.

Ci poniamo il problema di giustificare l'inclusione, nei nostri linguaggi di programmazione, di costrutti che possono causare divergenza. Hanno un'utilità che giustifica il rischio di non terminazione dei programmi.

Noi siamo abituati a scrivere funzioni ricorsive. Ad esempio, $Fact(n+1) = (n+1) * Fact(n)$, assieme al caso base. Dal punto di vista matematico questo oggetto è strano. Non si può definire una cosa in funzione di se stessa. C'è modo di definire in una maniera più sensata a livello matematico una funzione ricorsiva, che sottintende una procedura? È una problematica interessante.

La ricorsione primitiva è un tipo di ricorsione simile a quella finora vista ma con dei limiti semantici. In particolare possiamo dare il vincolo: nel corpo della definizione di $f(n+1)$ ci si può richiamare ricorsivamente solo su n. È equivalente ma più debole, dal punto di vista dell'espressività, limitare le chiamate ricorsive a oggetti più piccoli di $n+1$.

La ricorsione primitiva, per vincoli strutturali, garantisce la terminazione. Siamo interessati a questa classe di funzioni perché c'è un teorema che dice che le funzioni definibili in questo modo sono esattamente quelle definibili con un for.

Il seguente è un commento alle slides.

2.1.1 Slide 21

Quando ho più strade di espansione nei sistemi di riscrittura il problema di capire se la strada che scelgo è influente è il problema della confluenza, legato alla terminazione dell'espansione.

Nell'esecuzione ci sono dei problemi che non abbiamo indicato esplicitamente, come la valutazione per valore e per nome, l'ordine di valutazione delle espressioni, i side effect, ecc.

Non è chiaro cosa calcoli la funzione. È possibile semplificare il codice mediante inlining.

—-

L'idea delle funzioni ricorsive primitive è che abbiamo un argomento su cui facciamo la ricorsione ed una serie di parametri aggiuntivi. Abbiamo due casi: $x = 0$ o $x = succ(y)$.

$$f(x, \vec{z}) = \begin{cases} f(0, \vec{z}) = g(\vec{z}) \\ f(y+1, \vec{z}) = h(y, f(y, \vec{z}), \vec{z}) \end{cases}$$

Lavorare su una "sottostruttura" dell'input in ricorsione garantisce la terminazione della chiamata. La ricorsione primitiva è un sottocaso della ricorsione strutturale. Con la seconda ci si richiama solo su sottostrutture strette.

2.1.2 Esempi di funzioni definibili nel formalismo primitivo ricorsivo

Vediamo ora alcuni esempi di definizioni di funzioni comuni nel formalismo primitivo ricorsivo.

$$add(x,y) = \begin{cases} add(0,y) = y \\ add(x+1,y) = succ(add(x,y)) \end{cases}$$

$$mult(x,y) = \begin{cases} mult(0,y) = 0 \\ mult(x+1,y) = add(mult(x,y),y) \end{cases}$$

Il meccanismo della definizione di funzioni per ricorsione primitiva è naturale. Molte funzioni sono strutturate in maniera ricorsiva.

$$fact(x) = \begin{cases} fact(0) = 1 \\ fact(x+1) = (x+1) * fact(x) \end{cases}$$

La seguente funzione mi restituisce 1 se l'input è 0 e 0 altrimenti.

$$test(x) = \begin{cases} test(0) = 1 \\ test(x+1) = 0 \end{cases}$$

Proviamo a definire la funzione differenza. Ricordiamo che abbiamo definito la calcolabilità sui numeri naturali, interi positivi. Di conseguenza non possiamo calcolare la differenza, ad esempio, tra 3 e 7. Noi vogliamo una funzione che calcoli $a - b$ se $a > b$. In tutti gli altri casi calcoliamo 0.

Una prima definizione naive di $a - b$ è la seguente:

$$sub(a,b) = \begin{cases} sub(0,b) = 0 \\ sub(a+1,b) = succ(sub(a,b)) \end{cases}$$

Questa è sbagliata rispetto alla nostra specifica. Per $sub(1,1)$, ad esempio, restituisce 1. Una scelta migliore sarebbe andare in ricorsione su b.

Una delle scelte da fare è su cosa andare in ricorsione. Può essere uno dei parametri oppure un nuovo valore costruito a partire dai parametri.

Proviamo con la seguente definizione:

$$sub(a,b) = \begin{cases} sub(a,0) = a \\ sub(a,b+1) = pred(sub(a,b)) \end{cases}$$

dove *pred* restituisce 0 per 0 e $x - 1$ in ogni altro caso.

Questa definizione è un pò borderline, perché andrebbe dimostrato che cambiando il parametro su cui vado in ricorsione ottengo un formalismo equivalente a quello introdotto. Tuttavia ciò è possibile perciò la definizione rientra nel formalismo primitivo ricorsivo.

Vogliamo ora una funzione che restituisca 1 se i parametri sono uguali, 0 altrimenti.

$$comp(n,m) = test(add(sub(n,m), sub(m,n)))$$

Benchè sembri limitante è veramente potente questo tipo di ricorsione.

Quando parliamo di predicati intendiamo funzioni che restituiscano un booleano.

Supponiamo di saper calcolare un certo predicato P. Possiamo calcolare anche la sua negazione.

Data la funzione caratteristica di P, c_P, possiamo calcolare la funzione caratteristica di $\overline{P}$: $c_{\overline{P}}(x) = 1 - c_P(x)$.

Date c_P e c_Q abbiamo che $c_{P \wedge Q}(x) = c_P(x) * c_Q(x)$.

È possibile definire $c_{P \vee Q}$ con le leggi di de Morgan: $A \vee B = \overline{\overline{A \vee B}} = \overline{\overline{A} \wedge \overline{B}}$. Un altro modo è usano la funzione normalizzazione, che normalizza i numeri a 0 e 1:

$$c_{P \vee Q}(x) = norm(c_P(x) + c_Q(x))$$

Dati dei predicati possiamo quindi calcolare i connettivi logici tra di loro nel nostro formalismo. Passiamo ora al discorso dei quantificatori.

Per calcolare il quantificatore esistenziale dovrei avere una procedura del genere:

```
x = 0
while ¬P(x):
    x = x + 1
```

Questo potrebbe ciclare all'inifinito se l'esiste non vale. Abbiamo un problema duale con il quantificatore universale.

È evidente che c'è un problema ma non è detto che questo non sia insormontabile.

Quello che sappiamo senza dubbio calcolare è la quantificazione limitata, o bound. L'idea è che ho un upper bound finito alla mia quantificazione. Calcoleremmo quindi $g(n) = \exists x \leq n, P(x)$.

Ovviamente possiamo calcolare la quantificazione limitata con la ricorsione primitiva. Nel caso del quantificatore universale:

$$f(0) = p(0)$$
$$f(n+1) = p(n+1) * f(n)$$

Tanti problemi aperti dell'aritmetica sono legati alla quantificazione: sapere se esiste un numero che ripetti tale proprietà o se una tale proprietà vale per tutti i numeri.

È anche una maniera per approcciarsi ai problemi. Domandarsi se c'è un bound permette di rendere l'algoritmo più efficiente e certamente terminante.

2.2 Test di primalità

Vediamo ora un altro predicato interessante, il test di primalità.

Perché escludiamo 1 dai numeri primi? Perché uno dei risultati più importanti dell'aritmetica è la fattorizzazione unica, che vale per tutti i numeri escludendo 1. Per mantenere quel teorema 1 viene escluso.

Definiamo la proprietà $P(x)$:

$$x > 1 \wedge \forall z(z|x \implies (z = 1 \vee z = x))$$

$z|x$ è notazione per z divide x (più precisamente, z è un fattore di x).

$$z|x := \exists a, z * a = x := \exists a \leq x, z * a = x$$

Possiamo porre un bound anche al test di primalità: possiamo fermarci ad x. Siamo ora in grado sia di definire sia di calcolare la funzione di primalità: // TODO DA SCRIVERE

2.3 Minimizzazione

Vediamo ora un'altra operazione che opera su domini limitati che useremo molto spesso: la minimiazzazione.

$$\mu x, P(x)$$

Possiamo vederla come uno snippet del genere:

```
x = 0
while ¬P(x):
    x = x + 1
return x
```

Fissiamo un ordinamento e cerchiamo il primo x che soddisfa $P(x)$. Quello che ci interessa alla fine del ciclo è il valore di x. Questo è, sostanzialmente, cosa l'operazione di minimizzazione fa.

While è un costrutto imperativo. Noi preferiamo, per la nostra teoria della calcolabilità, un costrutto funzionale, da cui l'introduzione del μ. Il risultato di questo operatore è x.

Al solito col while abbiamo il problema che il while potrebbe non fermarsi mai. Quello che possiamo certamente sperare di scrivere, e quindi calcolare, nel nostro formalismo è una forma limitata di μ. Cerchiamo il più piccolo x minore di un certo y su cui vale un predicato $P(x, \vec{z})$.

$$f(y, \vec{z}) = \mu x \leq y, P(x, \vec{z})$$

I parametri di $\vec{z}$ rappresentano parametri ulteriori che possono essere utili.

Cosa vogliamo restituire se non troviamo x nell'intervallo fissato? Dobbiamo restituire un valore e ne possiamo restituire uno qualunque. La cosa più naturale è restituire $y + 1$, se y è il mio upper bound.

Vogliamo trovare un modo per scrivere questa operazione, non necessariamente per calcolarla efficientemente.

Definiamo prima il predicato R:

$$R(y, \vec{z}) = \forall x \leq y, \neg P(x, \vec{z})$$

Questo mi dice "fino a y non ho trovato il minimo", con y compreso.

R, come funzione, è una costante di valore 1 fino all'y_0 minimo per cui vale $P(y_0, \vec{z})$. Da lì in poi il suo valore diventa costantemente 0.

Possiamo ora scrivere μx:

$$\mu x \leq y, P(x, \vec{z}) = \sum_{w \leq y} \neg R(w, \vec{z})$$

Possiamo definirlo in un altro modo. Prendiamo in considerazione il predicato M:

$$M(x, \vec{z}) = P(x, \vec{z}) \wedge \forall y < x, \not P(y, \vec{z})$$

Questo predicato mi dice "x è il più piccolo valore per cui vale P". Come faccio però a trovare x? Si può moltiplicare x per $M(x, \vec{z})$. Dato che dobbiamo testare tutti gli $x \leq y$, abbiamo che μ può essere espresso come:

$$\mu x \leq y, P(x, \vec{z}) = \sum_{x \leq y} x \cdot M(x, \vec{z})$$

2.4 Generazione numeri primi

Come possiamo trovare il più piccolo numero primo successivo ad un numero i? Con la seguente funzione, ad esempio:

$$\Pi(i) = \begin{cases} \Pi(0) = 2 \\ \Pi(x+1) = \mu p.prime(p) \wedge p > \Pi(x) \end{cases}$$

Cosa manca? Bisogna dare un bound alla minimizzazione. C'è un teorema che dice che è sempre possibile trovare un numero primo tra n e $2n$. Il bound che possiamo dare è $2\Pi(x)$.

L'importante è dare un bound. C'è un altro bound, molto più grande, che andrebbe bene comunque: $\Pi(x)!$.

Come si dimostra l'infinità dei numeri primi? Supponiamo che siano finiti e siano $p_1, p_2, \ldots, p_n$. Prendiamo il numero $p_1 \cdot p_2 \cdots p_n + 1$. Questo numero non è divisibile per nessun p_i della mia enumerazione. Quindi o è un numero primo oppure i suoi fattori non fanno parte di quella lista. In ogni caso ho un assurdo.

È una dimostrazione bella. La tecnica è analoga alla diagonalizzazione: costruisco un nuovo elemento da quelli di una lista che prova il mio assurdo.

2.5 Ricorsione multipla

Consideriamo una possibile codifica del piano e consideriamo la coppia $< n, m >$. Non siamo troppo interessati alla codifica della coppia in sè, ma alle funzioni che mi restituiscono le componenti della coppia.

Abbiamo che, in generale, le componenti sono $\leq$ della (codifica della) coppia: $n \leq < n, m >$ e $m \leq < n, m >$.

Supponiamo di voler calcolare $\pi_1(x)$, ovvero la prima proiezione della coppia x. Partiamo dalla seguente definizione:

$$\pi_1(x) = \mu n, \exists m, < n, m >= x$$

Manca un bound. Quale prendiamo? x:

$$\pi_1(x) = \mu n \leq x, \exists m \leq x, < n, m >= x$$

Talvolta la ricorsione può essere necessaria su più di un valore. Nel formalismo che abbiamo visto finora non abbiamo questa possibilità.

Consideriamo la sequenza di Fibonacci:

$$\begin{aligned} \mathit{fib}(0) &= 1 \\ \mathit{fib}(1) &= 1 \\ \mathit{fib}(x+2) &= \mathit{fib}(x+1) + \mathit{fib}(x) \end{aligned}$$

La funzione di Fibonacci è intrinsicamente esponenziale, ma questo è il peggior metodo per calcolarlo. Siamo esponenziali nel numero di chiamate oltre ad esserlo nell'input. Inoltre così com'e scritta ora non rispetta i vincoli del formalismo primitivo ricorsivo.

Qual è l'idea? Bisogna portarsi dietro un accumulatore.

Vogliamo definire la seguente funzione:

$$\mathit{fibo'}(x) =< \mathit{fib}(x), \mathit{fib}(x+1) >$$

Possiamo definirla nel formalismo primitivo ricosivo nel seguente modo:

$$\begin{aligned} \mathit{fibo'}(0) &=< 1, 1 > \\ \mathit{fibo'}(x+1) &=< \mathit{fib}(x+1), \mathit{fib}(x+2) > \\ &=< \pi_2(\mathit{fibo'}(x)), \pi_1(\mathit{fibo'}(x)) + \pi_2(\mathit{fibo'}(x)) > \end{aligned}$$

L'unica chiamata ricorsiva che faccio è $\mathit{fibo'}(x)$.

Questa funzione corrisponde all'incirca al seguente snippet:

```
acc_0 = 1
acc_1 = 1
for i in range(x+1):
    tmp = acc_0
    acc_0 = acc_1
    acc_1 = tmp + acc_1
return acc_1
```

I seguenti sono commenti alle slides.

2.5.1 Slide 33

Questa tecnica è generale è prende il nome di ricorsione di coda.

Tutte le funzioni ricorsive primitive possono essere espresse mediante un for. È possibile dimostrare anche il viceversa. Il formalismo primitivo ricorsivo è equivalente, a potere espressivo, ai programmi scrivibili con il for, ovvero senza while e senza ricorsione generale.

2.6 Funzione di Ackermann

2.6.1 Slide 37

Vediamo ora una funzione che non è possibile scrivere nel formalismo primitivo ricorsivo. Va immaginata come una famiglia di funzioni dove il primo parametro mi istanza la particolare funzione. Abbiamo ack_0, ack_1, etc.

I casi sono mutalmente disgiunti. Data una tripla qualsiasi di valori solo una riga si applica.

La funzione termina? Sì, ma non è banale. L'argomento più importante della funzione è il primo. Se il primo decresce bene. Altrimenti vado a vedere gli altri. Questo mi dà un ordinamento delle mie chiamate ricorsive che mi da un'idea del fatto che la funzione è terminante.

Cosa calcola? ack_0 è abbastanza banale: è la somma. ack_1 invece calcola il prodotto tra x e y. ack_2 calcola l'elevamento a potenza (y^x).

ack_i itera ack_{i-1}. Il prodotto itera la somma. L'elevamento a potenza itera la moltiplicazione la tetrazione itera l'elevamento a potenza.

La funzione, benchè terminante, ha una complessità spaventosa.

Perché non posso scriverla nel formalismo primitivo ricorsivo? Perché questa funzione cresce troppo velocemente. Cresce più velocemente di qualsiasi funzione esprimibile col formalismo primitivo ricorsivo. La funzione di Ackermann mi dà un bound computazionale alle funzioni esprimibili con il formalismo primitivo ricorsivo. Di conseguenza non può essere esprimibile nello stesso formalismo.

Se riesco ad esprimere un bound computazionale alla complessità di un programma so che non è possibile calcolare quel bound nello stesso formalismo in cui ho espresso il mio programma.

Tutte le singole istanze di Ackermann sono scrivibili nel formalismo primitivo. Ma non posso scrivere un programma che le esprima tutte. Il formalismo non è abbastanza parametrico.

È sufficiente aggiungere l'ordine superiore al formalismo primitivo per poter esprimere la funzione di Ackermann.

La funzione di Ackermann è una funzione chiaramente calcolabile, essendo esprimibile in un qualche linguaggio di programmazione, ma non è esprimibile nel formalismo primitivo ricorsivo.

Si può dimostrare che c'è un ordinamento ben fondato e quando facciamo le chiamate ricorsive nella funzione di Ackermann le variabili decrescono secondo questo ordine.

Un ordinamento si dice ben fondato se non esistono catene discendenti infinite.

Dire che ho un ordinamento ben fondato non è la stessa cosa di dire che di elementi minori di un dato elemento m ce ne sono una quantità finita.

Vediamo un esempio: l'ordinamento lessicografico. Considerando $\mathbb{N} \times \mathbb{N}$. Definiamo l'ordinamento $<_p$ nel seguente modo: $< n_1, m_1 ><_p< n_2, m_2 > \iff n_1 < n_2 \vee (n_1 = n_2 \wedge m_1 < m_2)$. Questo ordinamento è ben fondato. Vale che $\forall m, < 2, m ><_p< 3, 7 >$.

Non è possibile costruire catene discendenti infinite. Proviamo a costruirne una: $< 3, 7 >>_p< 3, 6 > ... < 3, 0 >>_p< 2, 10^4 >$

Arrivati qui posso ripetere il giochetto di prima finché non arrivo a 0, dopodiché dovrò decrementare la prima componente. Di queste sequenze ne posso fare di lunghezza arbitraria ma sempre finita. Questo ragionamento ci dà un'idea del perché la funzione di Ackermann termina (il principio alla base della dimostrazione è lo stesso).

Capitolo 3

Altri formalisimi e Macchine di Turing

3.1 Formalismi totali e problema dell'interprete

Ci sono varie estensioni possibili al formalismo primitivo ricorsivo. Un esempio è il sistema T di Gödel, che aggiunge l'ordine superiore. Questo ci permette di scrivere la funzione di Ackermann. Un'altra possibilità è quella di rilassare la ricorsione primitiva permettendo una ricorsione che permetta di andare in ricorsione su ordinamenti ben fondati. Questa condizione è verificabile in termini sintattici, entro certi limiti.

Esteso il mio linguaggio ottengo la possibilità di scrivere la funzione di Ackermann e le mie funzioni sono totali. Sono complete? O c'è qualche funzione totale che posso pensare ma non scrivere? C'è una dimostrazione che dice che sì, tutti questi formalismi saranno incompleti. Un formalismo che permette di scrivere solo programmi terminanti sarà sempre incompleto.

La dimostrazione è abbastanza semplice. L'idea è che un programma che non riesco a scrivere (tra i tanti) è l'interprete per il linguaggio stesso.

Cosa intendiamo per interprete? Qui parliamo sempre di funzioni da $\mathbb{N}$ a $\mathbb{N}$. Dovremmo quindi definire l'interprete in questi termini.

L'input dell'interprete non è un programma in senso astratto. Prende in input una stringa che esprime il programma. Questa stringa può essere letta come un numero. Dire che una funzione è calcolabile è equivalente a definire l'interprete per tale funzione, se vogliamo dare una definizione operativa.

I seguenti sono commenti alle slides.

3.1.1 Slide 47

φ_n è la funzione che calcola il programma P_n, n-esimo elemento di un ordinamento (ad esempio lessicografico) dei programmi esprimibili nel linguaggio $\mathcal{L}$.

P_n è un livello intensionale, una descrizione di una funzione. φ_n è la funzione calcolata dal programma n-esimo.

Data una numerazione effettiva l'interprete è intuitivamente calcolabile. È fondamentale la numerazione: dato n devo poter sapere qual'è la funzione n-esima da interpretare.

3.1.2 Slide 48

L'argomento è sempre diagonale. Mi muovo sulla digaonale mentre sui lati ho due infinità (numero dei programmi e lunghezza dei programmi ad esempio).

L'interprete è un esempio di funzione intuitivamente calcolabile ma non esprimibile in un formalismo totale. Tipicamente è sempre possibile, dato un linguaggio che esprime funzioni totali, trovare un linguaggio più espressivo.

La completezza algoritmica è un concetto obsoleto. Gli algoritmi a cui siamo spesso interessati sono quelli polinomiali in complessità. A questo punto, perché non restringersi a linguaggi di programmazione che permettono di scrivere solo programmi con questa complessità? Si può fare, ce ne sono tanti di linguaggi del genere, basta imporre i giusti vincoli sul linguaggio.

Da un punto di vista operativo si perde in praticità. Ma questo è anche legato al capire la complessità computazionale dell'algoritmo. Se però si capisce bene perché un dato algoritmo ha una certa complessità allora possiamo strutturare il programma che lo realizza in modo che rispetti i vincoli del linguaggio. Il nostro approccio è al contrario: scriviamo i programmi e poi li analizziamo. Questo approccio non ci dà un granchè di informazioni. Non abbiamo alcun metodo generalizzato che ci dia informazioni o garanzie su programmi. Sarebbe più bello avere queste informazioni a priori.

Nei linguaggi di programmazione comuni (C, Python, ecc.) posso scrivere l'interprete per il linguaggio stesso. Perché? Perché non ho garanzie di terminazione ($\varphi_i(i)$ divergerebbe).

3.1.3 Slide 46

Abbiamo una gerarchia nota dei linguaggi totali. Un'interessante caratterizzazione del sistema T è che le funzioni calcolabili in questo sistema sono esattamente quelle calcolabili nell'aritmetica di Peano al primo ordine.

3.2 Tesi di Church

I seguenti sono commenti alle slides.

3.2.1 Slide 50

Che succede se rinunciamo a questa idea della totalità? Sappiamo che non daremo luogo a paradossi. Tuttavia rimane il problema: siamo nella situazione dei formalismi totali, e cioè esiste una gerarchia di formalismi più potenti, o

colpiamo un upper bound tale che in un formalismo del genere posso calcolare tutte le funzioni che posso calcolare in qualsiasi altro formalismo? Quest'ultima situazione sembrerebbe miracolosa dati i risultati visti finora.

La cosa che era ragionevole aspettarsi era la prima. Quello che sembra essere, ma che non è dimostrabile, è che la nozione di calcolabilità è indipendente dal formalismo che uso. Se ho un formalismo abbastanza espressivo posso esprimere qualsiasi funzione intuitivamente calcolabile. Questo è il succo della tesi di Church.

Quello che possiamo fare è confrontare formalismi a livello di potere espressivo. Più formalismi si sono considerati più si è avvalorata l'idea che la classe di funzioni che possiamo calcolare sia la stessa, ed in particolare quella delle funzioni calcolabili da una macchina di Turing.

Ci sono delle funzioni intuitivamente definibili ma non calcolabili? Non lo sappiamo. Rimane un problema aperto.

3.3 Alcuni formalismi Turing completi

3.3.1 Slide 52

Dimostrare la Turing-completezza dei linguaggi moderni è complesso. Molti formalismi sono stati studiati e sono stati trovati tutti equivalenti a potere espressivo.

Un sistema interessante è la logica combinatoria. Ho due costanti, che per ragione storiche si chiamano S e K. I programmi sono scritti come grosse combinazioni di S e K. Ad esempio:

$$(K((SK)K))$$

Abbiamo due regole di risctittura:

- $((KM)N) \to M$
- $(((SP)Q)R) \to (PR)(QR)$

È dimostrabile che questo sistema è Turing completo. Ovviamente c'è il problema di codificare i dati di input e output. Questi sono quelli che sono chiamati combinatori.

Questi formalismi sono semplici. Si può dimostrare quindi l'equivalenza tra due formalismi come meta-teorema in maniera agevole.

Il λ-calcolo è una di quelle cose dell'informatica che è così e non potrebbe essere altrimenti. Si giustifica intrinsicamente. È la cosa più naturale, in un certo senso. La logica combinatoria, ad esempio, non è così invece.

L'idea del λ-calcolo è che voglio un linguaggio per descrivere funzioni. Ho bisogno di:

- variabili

- un meccanismo per definire funzioni. Introduciamo quindi, data un termine M,$\lambda x.M$. Questa è l'operazione di introduzione (o costruzione) della funzione. Manca l'operazione di eliminazione (o distruzione)
- introduciamo quindi l'applicazione: (MN)

Posso partire da x e, perché no, applicare x a se stesso. Dopodiché introduco l'astrazione. Ottengo $\lambda x.(xx)$.

Qual è la regola di calcolo? Questa nasce dall'interazione tra costruttore e distruttore.

Cosa mi aspetto da $(\lambda x.M)N$? Mi aspetto M con tutte le occorrenze (libere) di x sostituite da N:

$$(\lambda x.M)N \to M[N/x]$$

L'operatore visto prima è noto comunemente come $\delta = \lambda x.(xx)$. Definiamo I come $I = \lambda x.x$. È un formalismo di alto livello. Non ci sono tipi, posso applicare espressioni ad altre espressioni senza limiti.

Consideriamo la seguente sequenza di calcolo dell'espressione (δI):

$$(\delta I) \to (II) \to I$$

I è un termine in forma normale: non c'è più alcuna riduzione possibile per questo termine.

Cosa succede con $(\delta\delta)$? Si riscrive se stesso all'infinito.

Un altro formalismo è quello delle funzioni μ-ricorsive.Cosa sono le funzioni μ-ricorsive? Si prende il formalismo primitivo ricorsivo e si aggiunge la minizzazione unbound. È Turing completo.

Quando definiamo un formalismo abbiamo due modi. Un modo è quello descrittivo, ovvero definisco un linguaggio ed eventualmente delle regole. È una descrizione ad alto livello. L'altro modo, un pò più simile alle macchine di Turing o alle macchine a registri, è di dare un'architettura di basso livello e scrivere i propri programmi utilizzando questa architettura.

3.4 Macchine di Turing

I seguenti sono commenti alle slides

3.4.1 Slide 54

Finché lavoro con linguaggi ad alto livello è difficile convicersi che abbiano lo stesso potere espressivo delle macchine di Turing. Inoltre rimane il dubbio: siamo sicuri di non aver tralasciato un costrutto che permette di fare un balzo nel potere espressivo?

La macchina di Turing che consideriamo ha un nastro solo. È un nastro di memoria infinito. Esiste? No, è un'astrazione matematica.

Un programma è composto da una lista infinita di operazioni che associano ad una coppia < carattere, stato interno > una tripla < nuovo carattere, nuovo stato, mossa >, dove mossa e' *dx* o *sx*.

3.4.2 Slide 55

La computazione deve essere deterministica. Dato un input alla funzione di transizione ci può essere un solo output.

3.4.3 Slide 56

Devo rispondere ad alcune considerazioni. Ad esempio, dove si trova la testina rispetto all'input? Noi supponiamo che la testina parta dall'inizio dell'input.

Se ho un nastro solo su quello scrivo l'input e alla fine su quello trovo l'output. Devo decidere come interpretarlo; ci sono vari modi standard.

// TODO SEE IF THERE IS A ASCII ART PACKAGE FOR THIS

```
-----------------
  |0|1|1|0|#|
-----------------
    @
```

Supponiamo di essere nello stato q_0 e di essere in posizione @. Consideriamo il seguente programma:

$$
\begin{aligned}
< q_0, 0 > &\rightarrow < q_1, 0, R > \\
< q_0, 1 > &\rightarrow < q_2, 1, R > \\
< q_1, 0 > &\rightarrow < q_1, 0, R > \\
< q_1, 1 > &\rightarrow < q_1, 1, R > \\
< q_1, \# > &\rightarrow < q_3, 0, R > \\
< q_3, 1 > &\rightarrow < q_2, 0, R > \\
< q_3, 0 > &\rightarrow < q_4, \#, L > \\
< q_3, 1 > &\rightarrow < q_4, \#, L > \\
< q_3, b > &\rightarrow < q_4, \#, L > \\
< q_2, 0 > &\rightarrow < q_2, 0, R > \\
< q_2, 1 > &\rightarrow < q_2, 1, R > \\
< q_2, \# > &\rightarrow < q_3, 1, R >
\end{aligned}
$$

Possiamo associare q_1 allo stato "ho letto uno zero".

Questo programma copia il primo carattere in input nella posizione di # e poi sposta # a destra.

Abbiamo $Q = \{q_0, \ldots, q_4\}$ e $F = \{q_4\}$

Ad ogni coppia stato simbolo viene associata una tripla nuovo stato, nuovo simbolo e mossa. Ci sono una infinità di varianti (tutte equivalenti dal punto di vista del potere formale).

La mossa da fare sarà unica perché la macchina è deterministica.

Una configurazione istantanea è una descrizione istantanea della configurazione della macchina. Non corrisponde allo stato interno della macchina, ma quest'ultimo ne fa parte. La si può pensare come ciò che devo ricordare per

riprendere una computazione interrotta più tardi. Si parla di configurazione solo in relazione ad un dato nastro di input.

Ho bisogno di salvare tre informazioni, avendo un nastro solo:

- lo stato interno
- il nastro
- la posizione della testina sul nastro

L'ultimo passaggio è delicato: avrei bisogno di un'origine per definire la posizione. Non è però chiaro definire dove sta questa origine. Avendo nastri infiniti non ho un'idea ben definita di origine. Potrei fissarne una ma poi dovrei separare il nastro in due parti. Con un seminastro sarebbe facile. La cosa più semplice è definire come origine il dove si trova la testina. A questo punto mi interessa memorizzare solo il seminastro a destra della testina e quello a sinistra.

La mia configurazione sarà quindi:

$$\alpha, q, \beta$$

α è il seminastro sinistro, β quello destro. Possiamo ora descrivere la comuputazione come una sequenza di configurazioni.

Descriviamo la computazione di prima:

$$\begin{gathered}\emptyset, q_0, 0110\# \\ 0, q_1, 110\# \\ 01, q_1, 10\# \\ 011, q_1, 0\# \\ 0110, q_1, \# \\ 01100, q_3, \emptyset \\ 0110, q_4, 0\#\end{gathered}$$

Per comodità è sempre utile far vedere il primo carattere a destra e a sinistra della testina. Se un seminastro è blank posso immaginare di avere b come primo carattere:

$$\alpha, q, \beta \equiv \alpha_1 a, q, b\beta_1$$

Supponiamo che questa sia la configurazione

$$\alpha a, q, b\beta$$

Supponiamo che questa sia l'istruzione

$$< q, b > \rightarrow < q', b', R >$$

Allora

$$< \alpha a, q, b\beta > \vdash < \alpha a b', q', \beta >$$

Analogamente, se $< q, b > \rightarrow < q', b', L >$ allora

$$< \alpha a, q, b\beta > \vdash < \alpha, q', ab'\beta >$$

Questa è la semantica della macchina di Turing.

Il processore è a tutti gli effetti una macchina a stati finiti

Perché è importante l'idea della macchina di Turing? Perché la macchina di Turing racchiude il concetto di operatore di calcolo più naturale che possiamo immaginare.

Quello che l'agente esecutore ha a sua disposizione è una memoria illimitata. L'agente è un qualcosa di finitistico, della memoria ha una visione finita. Quello che vede è la cella, di dimensione finita ma senza alcuna assunzione sulla dimensione della cella. Non ha una visione sinottica dell'intero nastro, ha una visione limitata dalla sua natura. L'agente può modificare una porzione finita del nastro. Per semplicità diciamo che può modificare solo la cella. Può spostarsi e modificare il suo stato interno. Di quanto può spostarsi? Di una porzione finita. Può ripetere queste azioni. Più di questo non può fare.

Perché è così potente questa nozione? Perché per andare oltre a questa nozione di calcolabilità dovrei visionare e realizzare un agente di calcolo con capacità maggiori di quello descritto.

3.5 Macchine universali

Un'altra macchina interessante è la macchina universale. Questa è una macchina capace di eseguire una qualsiasi macchina di Turing.

Perché ho bisogno di un nasto per memorizzare gli stati? Perché questi devono essere codificati, e non so a priori quanto grande sarà la mia codifica.

```
-------------------------------------------------------------------------------
    ||q_{0}|0|q_{1}|0|R||q_{1}|0|q_{1}|0|R||
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
    ||q_{0}|0||
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
    |0|1|1|0|#|
-------------------------------------------------------------------------------
```

Ho tre nastri: il nastro degli stati, il nastro che simula la macchina di Turing ed il nastro dell'input. Ognuno ha la sua testina.

Perché è interessante questa macchina? Perché questa è, in sostanza, la macchina di Von Neumann, eccetto per alcune differenze non significative. Ad esempio VN ha accesso random invece che un nastro sequenziale.

Con le macchine di Turing ogni automa definiva un'architettura: servirebbero tante macchine quante funzioni esprimibili. La macchina universale invece può emulare le macchine di Turing con un'unica architettura. Ci sono differenze

con la macchina di Turing ma sono dettagli, la struttura è simile e le capacità della macchina universale non sono maggiori: il potere espressivo è lo stesso. Prendiamo come input una macchina e l'input della funzione e la macchina universale fa da interprete.

Noi passeremo ad una nozione ancora più astratta. Questo perché vogliamo una teoria della calcolabilità che sia machine-independent. Non vogliamo essere costretti a ridurci sempre ad un modello computazionale particolare.

L'idea è che dobbiamo pensare di avere una enumerazione dei programmi. φ_i è la funzione calcolata dall'i-esimo programma. Noi diremo la funzione i-esima.

Vogliamo assicurarci che la numerazione dei programmi sia effettiva. Ad esempio, dato 100 voglio sapere qual è il programma 100. Vogliamo quindi una funzione universale μ tale che:

$$\mu(i, x) = \varphi_i(x)$$

Questa è la macchina universale o interprete. Possiamo vedere i come la descrizione del programma.

Possiamo riformulare la tesi di Church in questo contesto come:

"f è intuitivamente calcolabile $\implies \exists i.\varphi_i = f$"

Capitolo 4

Problemi indecidibili

Ci chiediamo ora se ci sono dei problemi che non sono decidibili, ovvero non calcolabili in un formalismo Turing completo. Introduciamo l'argomento con uno dei problemi più noti in letteratura, l'*Halting problem*.

Prima però chiariamo meglio i concetti di totalità, calcolabilità e la relazione tra i due, oltre a introdurre i concetti di divergenza e convergenza.

4.1 Note introduttive su programmi e funzioni

Una qualsiasi funzione φ_i della nostra enumerazione delle funzioni calcolabili è una funzione parziale calcolabile: su alcuni input può non essere definita.

La calcolabilità non coincide con la totalità: esistono funzioni parziali calcolabili e funzioni totali non calcolabili.

Le funzioni possono essere non definite in un punto. Se questo è il caso per una funzione calcolabile i ed un punto x, avremo allora la divergenza del *programma* i: $\varphi_i(x) \uparrow \iff \varphi_i$ non è definita su input x. Divergenza e convergenza sono più proprietà del programma che della funzione che calcola. Nonostante qui usiamo il simbolo φ sia per i programmi che per le funzioni è bene ricordare che può avere un doppio ruolo e una semantica diversa associata ad esso. La relazione tra i due è: la funzione i nella mia enumerazione di funzioni calcolabili è quella calcolata dal programma i della mia enumerazione dei programmi.

La divergenza non è una proprietà generale delle funzioni. Non ha senso dire $\varphi_i \uparrow$ senza specificare dove diverge; questo perché la convergenza e la divergenza sono proprietà puntuale: valgono per un dato input. Al massimo $\varphi_i(x)$ per un qualche x può divergere.

4.2 Il problema della fermata

Il problema che ci interessa è il cosiddetto "problema della terminazione". Le funzioni che stiamo calcolando sono funzioni parziali: i programmi corrispondenti possono potenzialmente divergere. Noi vorremmo calcolare la seguente

funzione:

$$Term(i,x) = \begin{cases} 1 & \text{se } \varphi_i(x) \downarrow \\ 0 & \text{se } \varphi_i(x) \uparrow \end{cases}$$

Questà è la specifica della mia funzione. È una funzione totale. Ci chiediamo a questo punto, è anche calcolabile? La risposta, come vedremo, sarà negativa.

Per dimostrarlo supponiamo che *Term* sia calcolabile e prendiamo in considerazione ora la funzione intermedia g:

$$g(x) = \begin{cases} 1 & \text{se } Term(x,x) = 0 \\ \uparrow & \text{se } Term(x,x) = 1 \end{cases}$$

Se *Term* fosse calcolabile allora anche g sarebbe calcolabile. Se g è calcolabile deve esistere un k tale che $\varphi_k = g$. Ci può essere più di un programma che calcola g, ma a me ne basta uno.

Ci chiediamo ora, legittimamente, qual è il comportamento di $\varphi_k(k)$? Converge?

Abbiamo che $\varphi_k(k) = g(k)$. Quindi $\varphi_k(k) \uparrow \iff Term(k,k) = 0 \iff g(k) = 1 \iff g(k) \downarrow \iff \varphi_k(k) \downarrow$. Questo è contraddittorio. Verrebbe da concludere che $\varphi_k(k)$ converge. È tuttavia vero che $\varphi_k(k) \downarrow \iff Term(k,k) = 1 \iff g(k) \uparrow \iff \varphi_k(k) \uparrow$. Ma anche questo è contradditorio. L'ipotesi da cui siamo partiti è che *Term* fosse calcolabile. Concludiamo quindi che *Term* non è calcolabile.

È il primo caso di una funzione che possiamo pensare ma che non riusciamo a calcolare, almeno con questa formulazione qui.

La dimostrazione è un semplice ragionamento diagonale. Esistono quindi funzioni ben definite ma non calcolabili: non esiste un algoritmo che mi calcoli questo problema. Nella sua forma generale il problema delle terminazione non è algoritmico.

Cosa vuol dire nella sua forma generale? Significa che valga per tutti gli i e per tutti gli x. Per alcuni programmi e alcuni input è possibile dimostrare che il programma termina. Ci sono dei casi particolari che sono gestibili, ma non esiste un unico algoritmo che in modo uniforme su tutti gli i e tutti gli x sappia decidere se il programma i termini su input x.

Quali erano le nostre ipotesi? La calcolabilità dell'interprete e qualche piccola proprietà di chiusura sul mio formalismo. Non molto.

Questa funzione non è esprimibile in un formalismo Turing completo. Non è tuttavia assurda l'idea che esista un agente di calcolo con più capacità della macchina di Turing e che sia in grado di calcolare *Term*. È difficile da immaginare. Nella calcolabilità relativa si parte immaginando un oracolo che sia capace di calcolare *Term* e ci si chiede da lì cosa si possa calcolare (e cosa no).

4.3 Proprietà s-m-n

L'ipotesi della calcolabilità dell'interprete è un'ipotesi importante. Supponiamo, nella nostra teoria della calcolabilità, che esista un modo per calcolarla. Que-

sta proprietà è detta proprietà UTM, Universal Turing Machine: $\exists u \varphi_u(i, x) = \varphi_i(x)$. È dimostrabile in tutti i formalismi Turing completi.

La proprietà che andiamo ora a considerare è la cosiddetta proprietà s-m-n. Supponiamo di avere una funzione calcolabile $g(x, y)$. Cosa posso dire delle sue istanze?. Supponiamo di fissare x, ad esempio a 0. Ottengo $g(0, y)$, che è una funzione unaria che dipende solo da y. Possiamo indicare $g(0, y)$ come $f_0(y)$. In generale posso fare questo per $f_k(y) = g(k, y)$.

// TODO Da rivedere l'esistenza di h Se tutte queste funzioni sono calcolabili per ognuna c'è una qualche funzione, delle mia enumerazione, che la calcola. Esisterà quindi un programma che la calcola. Esiste quindi $\varphi_{h(k)}(y) = f_k(y) = g(k, y)$ che mi calcola $g(k, y)$. h è una funzione che mi calcola l'indice del programma che calcola la g che mi interessa. L'esistenza di h è praticamente ovvia: se esiste un indice k' per la funzione $g(k, y)$ allora questo k' è calcolabile (mediante h). La domanda ora è: h è calcolabile? La risposta è sì.

Noi supporremo la proprietà s-m-n, ma questa è facilmente dimostrabile in tanti formalismi.

Teorema 4.1. ***Proprietà s-m-n.*** *$\forall g$ calcolabile $\exists h$ totale e calcolabile tale che*

$$\forall x, y. \varphi_{h(x)}(y) = g(x, y)$$

Quello che stiamo facendo è una curryficazione. C'è un importante isomorfismo a livello di funzioni: lo spazio delle funzioni $(A \times B) \to C$ è isomorfo allo spazio delle funzioni $A \to (B \to C)$. L'operazione alla base della dimostrazione di questa affermazione e che mi permette di passare da una funzione del primo spazio alla sua corrispondente nel secondo si chiama curryficazione. L'idea è: data $g(x, y)$ fissiamo x. In questo modo ottengo $g(x, y)$, con x fissato, ovvero una funzione unaria in y da B a C.

Si può dimostrare anche con un argomento sulla cardinalità. Sappiamo che la cardinalità del primo spazio è $|C|^{|A \times B|} = |C|^{|A| \cdot |B|} = (|C|^{|B|})^{|A|}$, che è la cardinalita di $A \to B \to C$. Questo mi garantisce l'esistenza dell'isomorfismo. La dimostrazione precedente è un pò più strutturale (e costruttiva se così si vuol dire).

h fondamentalmente mi dà l'indice della funzione curryficata.

In un certo senso s-m-n mi dice che il mio formalismo è chiuso rispetto alle curryficazioni/λ-astrazioni. UTM mi dice che il mio formalismo è chiuso rispetto alle λ-applcazioni.

Possiamo generalizzare s-m-n a vettori x e y di parametri: $\forall g \forall m \forall n \exists s$ totale calcolabile tale che

$$\varphi_{s(\vec{x}_m)}(\vec{y}_n) = g(\vec{x}_m, \vec{y}_n)$$

s-m-n serve per calcolare un numero come indice di un programma. Se non bisogna calcolare un indice di un programma in funzione di qualcos'altro non mi serve s-m-n. Noi vedremo molti casi in cui avremo proprio bisogno di quello.

Vediamo un esempio di applicazione di s-m-n nella dimostrazione della non calcolabilità di alcune funzioni.

La funzione che ci interessa indagare adesso è *Tot* che determina se un certo programma è totale o meno. È anche questo un problema di decisione. La specifica della mia funzione è:

$$Tot(i) = \begin{cases} 1 & \text{se } \forall x \varphi_i(x) \downarrow \\ 0 & \text{altrimenti} \end{cases}$$

Abbiamo prima però di un risultato intermedio. Un caso particolare caso del problema della terminazione è il problema della terminazione diagonale:

$$Term(i) = \begin{cases} 1 & \text{se } \varphi_i(i) \downarrow \\ 0 & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Se la terminazione diagonale non è calcolabile la terminazione generabile non è calolabile. Non è vera l'implicazione inversa. Ha senso quindi chiedersi se la terminazione diagonale è calcolabile. La risposta è no, e la dimostrazione è analoga a quella della terminazione generale.

Dimostrare che una funzione non sia calcolabile non è banale. Devo dimostrare che non esiste un algoritmo tale che la calcoli. È molto più difficile rispetto a dimostrare la calcolabilità di una funzione.

Abbiamo due strade. Assumiamo la calcolabilità di *Tot* e troviamo un assurdo. Oppure usiamo un procedimento di riduzione: se è calcolabile *Tot* deve essere calcolabile $Term_i$. Da lì poi dimostriamo la non calcolabilità di $Term_i$.

Prendiamo $g(i, x) = \varphi_i(i)$. Abbiamo due casi: o converge o diverge. Nel primo caso se fisso i la funzione curryficata che ottengo è totale. Nell'altro caso no.

Per s-m-n abbiamo h totale e calcolabile tale che

$$\varphi_{h(i)}(x) = g(i, x) = \varphi_i(i)$$

Ora mi chiedo: quanto vale $Tot(h(i))$? Abbiamo che

$$Tot(h(i)) = \begin{cases} 1 & \text{se } \varphi_i(i) \downarrow \\ 0 & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Componendo *Tot* con h risolverei il problema della terminazione diagonale. Ma questo è assurdo. Da cui *Tot* non è calcolabile.

In questa dimostrazione parto da funzioni calcolabili che vado a comporre con la mia funzione di partenza (*Tot*) e ottengo una funzione che mi potrebbe calcolare qualcosa che so non essere calcolabile. Da ciò concludo che la mia funzione di partenza non è calcolabile. È una dimostrazione diversa da quella del problema della terminazione.

È interessante vedere questa dimostrazione anche "al contrario", in versione bottom up. Come faccio a dimostrare la non calcolabilità di *Tot*? Assumo che sia calcolabile e la uso per risolvere un problema indecidibile. In questo caso vogliamo ridurci alla terminazione diagonale. Come facciamo? Possiamo farlo

cercando una funzione h tale che, per ogni i, $\varphi_{h(i)}$ è totale sse $\varphi_i(i)\downarrow$. Esiste questa funzione h? Sì, e possiamo dimostrarlo in due passaggi.

Per prima cosa definiamo una funzione calcolabile binaria $g(i,x)$ che, in base alla terminazione di o meno di $\varphi_i(i)$, ha un comportamento diverso che rispetti le condizioni che abbiamo posto su $\varphi_{h(i)}$. Ad esempio

$$g(i,x) = \begin{cases} 1 & \text{se } \varphi_i(i)\downarrow \\ \uparrow & \text{se } \varphi_i(i)\uparrow \end{cases}$$

Questa funzione è calcolabile? Sì, è facilmente scrivibile in un linguaggio di programmazione. Ma a questo punto ho trovato la h che cercavo, grazie alla proprietà s-m-n: $\exists h$ tot. e calc.$\varphi_{h(i)}(x) = g(i,x)$. A questo punto, se mi chiedo se la funzione $\varphi_{h(i)}$ è totale, utilizzando la funzione *Tot*, mi sono ridotto al problema della terminazione diagonale.

Consideriamo un caso particolare e poi proviamo a generalizzare. Stiamo analizzando programmi. Mi chiedo se un mio programma calcola la funzione identità. Si potrebbe generalizzare al capire se il mio programma ha un certo comportamento rispetto ad una funzione di riferimento.

Vogliamo quindi

$$ID(i) = \begin{cases} 1 & \text{se } \forall x, \varphi_i(x) = x \\ 0 & \text{altrimenti} \end{cases}$$

Possiamo dimostrare che questa funzione non è calcolabile, con la stessa tecnica di prima. Ci riduciamo al problema della terminazione.

Costruiamo g tale che

$$g(i,x) = \begin{cases} x & \text{se } \varphi_i(i)\downarrow \\ \uparrow & \text{se } \varphi_i(i)\uparrow \end{cases}$$

Per s-m-n esiste h tale che $\varphi_{h(i)}(x) = g(i,x)$. Mi chiedo ora, $h(i)$ è la funzione identita?

$$ID(h(i)) = \begin{cases} 1 & \text{se } \varphi_i(i)\downarrow \\ 0 & \text{se } \varphi_i(i)\uparrow \end{cases}$$

Bisogna partire da una funzione calcolabile, altrimenti non si può applicare s-m-n. È parte fondamentale della dimostrazione mostrare che g è calcolabile.

4.4 Il predicato T di Kleene

Ci chiediamo ora se esista un predicato calcolabile $T(i,x,t)$ che mi dice se il programma i termina la computazione su input x entro il tempo t.

C'è il problema di come definiamo il tempo; tuttavia l'idea è che tutti i programmi di cui parliamo sono di tipo discreto, hanno un passo discreto di calcolo.

È calcolabile? Intuitivamente sì. Immaginiamo di avere un interprete. Interpretiamo il programma i, seguendolo passo per passo per input x. Se dopo t passi la computazione è terminata allora ho una risposta positiva; in caso contrario no.

Esistono tante varianti del predicato T di Kleene. Questa è una forma "ternaria". Possiamo pensare ad una versione "quaternaria" $T^4(i, x, y, t)$ che mi dice se il mio programma termina su output y in t o meno passi su input x.

La forma originale di Kleene era un predicato ternario $T(i, x, tr)$, dove tr è una traccia computazionale. Si può vedere come una sequenza di configurazioni istantanee. Non deve essere necessariamente completa, deve essere corretta. È ancora chiaramente decidibile se la traccia seguita sia quella passata in input. Questa forma in un certo senso comprende le prime due. L'idea è che la lunghezza della traccia è il tempo passato dall'inizio della computazione.

Si può addirittura dimostrare che T è primitivo ricorsivo in generale. Inoltre ha una complessità computazionale relativamente bassa. La complessità è lineare in t e nelle altre componenti.

C'è un corollario del predicato T di Kleene. Se supponiamo questo predicato come primitivo possiamo scrivere, sulla base di questo, la macchina universale, la cui esistenza smette di essere primitiva nella nostra teoria della calcolabilità.

Infatti possiamo definire u nel seguente modo:

$$u(i, x) = fst(\mu < y, t >, T(i, x, y, t))$$

Questo corrisponde a scrivere $fst(\mu w, T(i, x, fst(w), snd(w)))$. Il fst più esterno serve perché a me non interessa t, interessa y.

Questo è un risultato importante e si chiama forma normale di Kleene. C'è un corollario importante. Si potrebbe limitare sintatticamente un programma ad un while con all'interno un for e questo non diminuirebbe il potere espressivo del formalismo, poichè l'interprete è scrivibile in questi termini.

Un altro problema non calcolabile è il problema del raggiungimento di codice. Ovvero, dato un programma e un'istruzione il problema di decidere se il programma raggiungerà mai quell'istruzione non è calcolabile. Esistono delle tecniche ma queste non sono generali. Sono tipicamente usate dai compilatori per ottimizzare il codice oggetto ed eliminare parti di codice inutile.