Indice

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: $A \vee B = \overline{\overline{A} \wedge \overline{B}} = \overline{\overline{A} \wedge \overline{B}}$. 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:

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

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:

$$fibo'(x) =< fib(x), fib(x+1) >$$

Possiamo definirla nel formalismo primitivo ricosivo nel seguente modo:

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

L'unica chiamata ricorsiva che faccio è $fibo'(x)$.

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:

Teorema 4.2. *$\forall g \forall m \forall n \exists s$ 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.

Capitolo 5

Insiemi ricorsivi e ricorsivamente enumerabili

Supponiamo di voler trasmettere un insieme. Se questo è finito non c'è nessun problema, basta mandare i suoi elementi. Se ho un insieme infinito posso trasmettere il programma che calcola la funzione caratteristica del mio insieme. Non posso immaginare di poter trasmettere tutti gli insiemi in questo modo, altrimenti potrei risolvere il problema della terminazione diagonale. Quelli per cui è possibile sono detti ricorsivi.

Definizione 5.1. Un insieme A è ricorsivo sse c_A è calcolabile.

Un altro modo per trasmettere il mio insieme in maniera effettiva è dare un metodo di calcolo, detto funzione enumerativa. Supponiamo di avere un insieme $a_0, a_1, a_2, \ldots$. Diamo una funzione f tale che $f(0) = a_0$, $f(1) = a_1$, ecc. Abbiamo che $A = cod(f)$.

Quando posso dare l'insieme in questa maniera ed f è calcolabile si dice che l'insieme è ricorsivamente enumerabile. Ricorsivamente è terminologia vecchia, degli anni trenta del XX secolo. In queste due accezioni ricorsivo va visto come sinonimo di calcolabile.

Definizione 5.2. A è ricorsivamente enumerabile sse esiste una funzione di enumerazione f per A calcolabile: $A = cod(f)$, per f calcolabile.

5.1 Relazione tra insiemi ricorsivi e r.e.

Esistono quindi sostanzialmente due modi per descrivere in maniera effettiva un insieme A. Vogliamo capire che relazione c'è tra queste due nozioni: qual è più potente? Come si rapportano?

Per descrivere una classe è spesso utile capire rispetto a quali operazioni la classe è chiusa. In particolare, se A è ricorsivo cosa possiamo dire del complementare di A, o dell'unione/intersezione/differenza di A con un altro insieme ricorsivo?

Ricordiamo che $A \subseteq \mathbb{N}$. Abbiamo che:

- $\emptyset$ è ricorsivo: la funzione costante **0** è calcolabile;
- $\mathbb{N}$ è ricorsivo: la funzione costante **1** è calcolabile;
- ogni insieme finito è ricorsivo: D è finito $\implies$ D è ricorsivo:

$$c_D(x) = \begin{cases} 1 & \text{se } x = d_1 \vee x = d_2 \vee \ldots \vee x = d_n \\ 0 & \text{altrimenti} \end{cases}$$

  Sarebbe esprimibile anche con una serie di *if*. Qualunque sia D ho un modo per scrivere la mia funzione caratteristica.

Con lo stesso ragionamento posso dimostrare che molte altre funzioni sono calcolabili. Ad esempio una funzione con *Range* finito è sicuramente calcolabile; è scrivibile con un *case* ad esempio. Tutte le funzioni scrivibili con un *case* sono sicuramente calcolabili. Queste funzioni hanno sempre un numero finito di elementi su cui hanno un comportamento interessante e nei restanti hanno lo stesso comportamento.

L'insieme dei numeri pari è ricorsivo. Tutti gli insiemi primitivi ricorsivi, ovvero con funzione caratteristica primitiva ricorsiva, sono sicuramente ricorsivi. Infatti le funzioni primitive ricorsive sono un caso particolare di algoritmo.

Esistono insiemi non ricorsivi. Ad esempio $K = \{i \mid \varphi_i(i) \downarrow\}$ non è ricorsivo.

Supponiamo che A,B siano ricorsivi. Abbiamo che:

- $\overline{A}$ è ricorsivo. Questo perche $c_{\overline{A}}(x) = 1 - c_A(x)$
- $A \cup B$ è ricorsivo. Questo perche $c_{A \cup B}(x) = \max(c_A(x), c_B(x))$
- $A \cap B$ è ricorsivo. Questo perche $c_{A \cap B}(x) = \min(c_A(x), c_B(x))$

Gli insiemi ricorsivi formano un'algebra di Boole, essendo chiusi per queste tre operazioni. Sono una sottostruttura interessante degli insiemi.

Data una funzione caratteristica è facile costruire una funzione di enumerazione. In altri termini, se un insieme è ricorsivo è anche ricorsivamente enumerabile. C'è un piccolo inghippo però a cui bisogna fare attenzione.

La funzione di enumerazione potrebbe essere costruita nel seguente modo ad esempio:

```
def e_A(i):
    c = 0
    j = 0
    while c ≤ i:
        if c_A(j):
            c++
        j++
    return j − 1
```

In versione funzionale potremmo scrivere scrivere e_A come $e_A(i+1) = \mu j, c_A(j) = 1 \wedge j > e_A(i)$, con $e_A(0) = \mu j, c_A(j) = 1$. Il piccolo dettaglio a cui fare attenzione è che la funzione di enumerazione la vorremmo totale e calcolabile. Tuttavia questo non è un vincolo così restringente.

Diamo la seguente definizione: A è r.e. se esiste f totale calcolabile tale che $A = cod(f)$ oppure A è vuoto.

Il problema della nostra funzione di enumerazione è che può divergere con insiemi finiti. Se, ad esempio, ho un A con 7 elementi e cerco l'ottavo con la mia funzione avrò che divergerà.

Il teorema rimane, se A è ricorsivo è ricorsivamente enumerabile. Bisogna giusto ricordarsi che per gli insiemi finiti va fatto un caso speciale che gestisca input maggiori della cardinalità di A.

Più precisamente, se A è finito, con $A = a_0, \ldots, a_n$, allora definiamo e_A nel seguente modo:

$$e_A(x) = \begin{cases} x = 0 \Rightarrow a_0 \\ x = 1 \Rightarrow a_1 \\ \vdots \\ x = n \Rightarrow a_n \\ default \Rightarrow a_n \end{cases}$$

Abbiamo quindi che Ricorsivo $\implies$ R.E. Vale il viceversa? La risposta è no, ma non è del tutto ovvia.

Sia A un insieme r.e. Potremmo pensare di calcolare la funzione caratteristica di A con la sua funzione di enumerazione, nel seguente modo:

```
def c_A(x):
    i = 0
    while e_A(i) ≠ x:
        i++
    return 1
```

Questa c_A pero' non calcola la funzione caratteristica di A, bensì la funzione semicaratteristica di A,che indicheremo con s_A.

$$s_A(x) = \begin{cases} 1 & \text{se } x \in A \\ \uparrow & \text{altrimenti} \end{cases}$$

Questa funzione è calcolabile e parziale. Per ora abbiamo dimostrato che la funzione semicaratteristica di A è calcolabile, non abbiamo ancora dimostrato che r.e. $\not\implies$ ricorsivo. Dobbiamo mostrare un esempio di insieme r.e. che non sia ricorsivo. K è non ricorsivo. K è r.e.?

Proviamo a scrivere una funzione di enumerazione per K:

```
def e_K(< i, t >):
    if T^3(i, i, t) = 1:
        return i
```

```
else :
    return   k_0
```

Nell'else è importante che restituisca un programma che sta in K. Ce ne sono tanti e di semplici; ad esempio le funzioni costanti. Diamo indice k_0 ad un tale programma.

È una buona funzione di numerazione? È sicuramente calcolabile. È suriettiva? È facile vedere che numero solo cose che stanno in K: k_0 sta in K e se restituisco i allora i stava in K. Se i sta in K vuol dire che esiste un t tale che $\varphi_i \downarrow$ in t passi. Di conseguenza vengono enumerate tutte le funzioni convergenti su i di indice i.

Il dovetailing è al variare dell'input. Mi muovo sugli input e sul tempo di computazione.

Si potrebbe fare anche nel seguente modo:

$$e_K(c) = fst(\mu < i, t >, < i, t > \geq c \wedge T(i, i, t))$$

Il risultato finale è che K è ricorsivamente enumerabile.

C'è un'altra relazione notevole tra insiemi r.e. e insiemi ricorsivi.

Teorema 5.1. *Sia A un insieme tale che sia A che $\overline{A}$ sono r.e. Allora A è ricorsivo.*

Dimostrazione. Supponiamo di aver e_A ed $e_{\overline{A}}$. Informalmente, possiamo far partire la ricerca per entrambe le funzioni. Prima o poi una terminerà, e in base a quale termina ho la mia risposta. La computazione parallela è assolutamente algoritmica e non c'è problema al riguardo.

Più formalmente, sia h così definita:

$$h(x) = \begin{cases} e_A(x) & \text{se } x = 2n \text{ per qualche } n \\ e_{\overline{A}}(x) & \text{se } x = 2n+1 \text{ per qualche } n \end{cases}$$

Possiamo scrivere $c_A(n) = pari(\mu i, h(i) = n)$ □

Come corollario di questo teorema abbiamo che esiste un insieme nè ricorsivo nè r.e. Quale? $\overline{K}$. Se infatti $\overline{K}$ fosse r.e. allora K sarebbe ricorsivo.

Abbiamo una gerarchia degli insiemi fatta in questo modo:

// TODO RICOPIARE LA FIGURA DELLA GERARCHIA (Figura sul quaderno azzurro)

5.2 Enumerazioni iniettive e monotone

Ci concentriamo ora sulle proprietà dell'enumerazione. Ci chiediamo in particolare cosa possiamo dire riguardo alla sua monotonia e alla sua iniettività.

Supponiamo di poter enumerare A con una funzione effettiva. Ci chiediamo se possiamo trasformare f in modo da non avere ripetizioni. Se pensiamo all'enumerazione come ad uno stream infinito di numeri in input ci chiediamo se possiamo creare un filtro che elimini i duplicati dallo stream.

Questo si può fare in maniera algoritmica tenendo una lista dei numeri ricevuti dallo stream e, ogni volta che ho un nuovo numero, controllo se è gia nella lista. In base al risultato decido se aggiungerlo o meno alla lista. La mia lista è ora il mio nuovo stream senza ripetizioni. Abbiamo quindi che non è restittivo richiedere che la mia enumrazione sia iniettiva.

Diamo quindi la seguente definizione:

Definizione 5.3. A è R.E. sse $\exists f$ iniettiva tot. calc. tale che $A = cod(f)$ oppure A è finito.

A livello formale data l'enumerazione f possiamo costruire una nuova enumerazione g per lo stesso insieme che sia iniettiva nel seguente modo:

$$g(x) = \begin{cases} f(0) & \text{se } x = 0 \\ f(\mu i, \forall y \leq x-1, f(i) \neq g(y)) & \text{altrimenti} \end{cases}$$

Questo ha senso per insiemi infiniti.

L'iniettività è quindi una proprietà accettabile per la mia enumerazione. Posso trasformarla anche in una funzione monotona crescente? La risposta sarà no, non posso enumerare gli elementi di A in maniera crescente se A è r.e.

Il problema è fondamentalmente che non posso assicurarmi che un dato elemento sia il più piccolo nello stream. La ricerca divergerebbe. Ogni volta che cerchiamo un minimo c'è un problema del genere.

L'idea è che se potessimo ordinare gli elementi di A allora A sarebbe ricorsivo. Dato che non tutti gli insiemi r.e. sono ricorsivi non si può fare in generale.

Sia f un'enumerazione strettamente crescente. Essendo discreta cresce strettamente più dell'identità. Se voglio sapere se x fa parte della mia numerazione posso controllare gli elementi fino ad x, oltre sicuramente non lo troverò.

Più formalmente, la ricerca di x può essere fatta col seguente algoritmo:

$i_x = \mu i, f(i) \geq x$
return $(f(i_x) == x)$

In questo modo sono sicuro, se f è strettamente crescente, che la mia ricerca termina. Per capire se x stava nel mio stream controllo che $f(i_x)$ sia uguale ad x.

Questo algoritmo funziona se la mia funzione è non decrescente? Se la mia funzione diventa indefinitamente costante da un certo punto in poi potrei andare avanti all'infinito. La mia minimizzazione non ha più garanzia di terminare.

Questo problema sorge quando $cod(f)$ è finito. Quando $cod(f)$ è infinito prima o poi la funzione assumerà un nuovo valore. Tuttavia se $cod(f)$ è finito allora l'insieme enumerato da f è ricorsivo.

Ogni insieme ricorsivo infinito è enumerabile mediante una funzione di enumerazione crescente. Ogni insieme ricorsivo non vuoto è enumerabile mediante una funzione di enumerazione non decrescente.

Se si è in grado di ordinare si è anche in grado di decidere. Gli approcci generativi sono generalmente semidecidibili e permettono di generare elementi in maniera disordinata.

Riprendiamo ora $K = \{i \mid \varphi_i(i) \downarrow\}$. Come decidiamo se un certo i sta in K? Possiamo far partire il mio interprete e aspettare. Se termina allora i appartiene a K. Se non appartiene a K non lo saprò mai, perché dovrei aspettare che il programma termini. Di K non possiamo quindi calcolare la funzione caratteristica, bensì la funzione semicaratteristica:

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

In generale la funzione semicaratteristica di A, con A r.e., è fatta nel seguente modo:

$$s_A(x) = \begin{cases} \downarrow & \text{se } x \in A \\ \uparrow & \text{se } x \notin A \end{cases}$$

In alcuni casi posso semidecidere la appartenenza di un elemento ad un insieme, in altri casi potrei non esserne in grado.

5.3 Insiemi r.e. e semidecisione

Ci chiediamo ora che legame esista tra la capacità di enumerare e la capacità di semidecidere. Ce lo chiediamo in generale, ricordando che per K l'abbiamo già visto.

Supponiamo che A sia vuoto oppure $A = cod(f)$, con f tot. calcolabile. Come faccio a costruire l'algoritmo di semidecisione per A? Semplicemente comincio a cercare x nell'enumerazione.

$$s_A(x) = (\mu i, (f(i) = x))$$

Come facciamo per il viceversa? Abbiamo la procedura di semidecisione per A e vogliamo costruire la funzione di enumerazione per A. f è delicata perché potrebbe divergere. Dobbiamo essere più astuti. L'idea è muoversi mediante dove tailing sull'input e sul tempo di computazione.

$$f(< x, t >) = \begin{cases} x & \text{se } s_A(x) \text{ termina nel tempo } t \\ a_0 & \text{altrimenti} \end{cases}$$

f è totale e calcolabile. Abbiamo quindi un'enumerazione per A. Non è iniettiva, ma questo non importa. a_0 è un elemento qualsiasi di A. Possiamo quindi vedere la semidecisione come enumerazione di un insieme r.e. e viceversa.

Non siamo sicuri di poter trovare a_0. Questo pezzo della dimostrazione è non costruttivo. Dal punto di vista classico però o l'insieme è vuoto, e quindi r.e., oppure non lo è e deve avere almeno un elemento a_0. Deve quindi esistere la funzione calcolabile f, anche se non è detto che possiamo trovarla.

Vediamo ora un'applicazione. Quali sono le proprietà di chiusura per gli insiemi r.e.? Sono chiusi per complementazione? No, altrimenti gli insiemi r.e.

sarebbero tutti ricorsivi, e sappiamo che questo non è vero. Ad esempio K è r.e. e $\overline{K}$ non lo è.

Cosa possiamo dire di $A \cup B$ e $A \cap B$? Per $A \cup B$ è abbastanza ovvio. Dati f_A e f_B possiamo costruire $f_{A \cup B}$ in modo che $f_{A \cup B}(2x) = f_A(x)$ e $f_{A \cup B}(2x+1) = f_B(x)$. Per $A \cap B$ è semplice costruire il semidecisore di $A \cap B$ dati quelli per A e per B. Basta concatenare i due: $s_{A \cap B}(x) = s_A(x); s_B(x)$.

Piccola nota sulla notazione. Noi consideriamo funzioni parziali da $\mathbb{N}$ a $\mathbb{N}$: $f : \mathbb{N} \multimap \mathbb{N}$. Indichiamo, in maniera impropria, $cod(f) = \{y \mid \exists x, y = f(x)\}$ e $dom(f) = \{x \mid f(x) \downarrow\}$. Il nostro codominio in realtà è il *Range* della funzione ed il nostro dominio è il dominio di congergenza (o co-*Range*). In termini del grafo abbiamo che $dom(f) = \{x \mid \exists y, < x, y > \in grafo(f)\}$.

5.4 Caratterizzazioni alternative di un insieme r.e.

Vediamo ora alcune caratterizzazioni, tutte equivalenti, degli insiemi r.e. che sono più adatte in certi contesti.

Se si può caratterizzare un insieme r.e. A come il dominio di convergenza di una funzione s abbiamo che s rappresenta la funzione di semidecisione di A.

Abbiamo visto che A è r.e. sse $A = \emptyset \vee A = cod(f)$, f tot. calc.

Abbiamo però che A può essere visto come $dom(g)$, con g parz. calc. È una definizione equivalente a quella appena citata.

Si può infine definire A anche come $A = cod(h)$, con h parz. calc.

Solitamente quando costruiamo una funzione di enumerazione la vogliamo totale, è meno problematica.

Le tre definizioni sono equivalenti appena viste sono equivalenti.

Teorema 5.2. *Le tre seguenti affermazioni sono equivalenti e definiscono tutte un insieme r.e. A:*

1. *$A = \emptyset \vee A = cod(f)$, con f totale calcolabile*
2. *$A = dom(g)$, con g parziale calcolabile*
3. *$A = cod(h)$, con h parziale calcolabile*

Dimostrazione. Lo dimostriamo facendo vedere che (1) $\implies$ (2), (2) $\implies$ (3) e (3) $\implies$ (1).

- (1) $\implies$ (2):

  Se $A = \emptyset$, allora abbiamo che $A = dom(f_\emptyset)$, se indichiamo con $f_\emptyset$ la funzione ovunque divergente.

  Se invece $A \neq \emptyset$ sia f tale che $A = cod(f)$. Definiamo g come

  $$g(x) = \mu y, f(y) = x$$

Si ha che $A = dom(g)$.

- (2) $\implies$ (3):

  Sia $A = dom(g)$, con g parziale calcolabile. Sia h la funzione calcolata dal seguente programma:

  ```
  def  h(x):
       g(x)
       return  x
  ```

  Se $g(x)$ termina restituisco x. Si ha che $A = cod(h)$.

- (3) $\implies$ (1):

  Abbiamo due casi: A è vuoto oppure no.

  Se $A = \emptyset$ l'asserto è già dimostrato.

  Se $A \neq \emptyset$ allora $\exists a \in A$. A questo punto definisco la mia f come:

  $$f(< x, t >) = \begin{cases} h(x) & \text{se } h(x) \text{ termina entro il tempo } t \\ a & \text{altrimenti} \end{cases}$$

  f è totale, perché do sempre in output un valore. È calcolabile, sotto l'ipotesi che h sia calcolabile. Infine $cod(f) = A$ perché f restituisce solo elementi in A e me li restituisce tutti, dato che $A = cod(h)$.

□

La definizione (3) ci permette di non doverci soffermare sul caso particolare dell'insieme vuoto, che non ha implicazioni tecniche importanti.

5.4.1 Teorema della proiezione

Un'ulteriore caratterizzazione degli insiemi r.e. è la seguente:

Teorema 5.3. *A è r.e. sse $\exists B$ ricorsivo tale che $A = \{x \mid \exists y, < x, y > \in B\}$.*

Possiamo vedere B come una codifica delle coppie oppure come un insieme di coppie, non è importante.

A è la proiezione esistenziale di B

// TODO RIPORTARE FIGURA (vedi figura sul quaderno azzurro).

Il teorema è importante. Se ho un insieme ricorsivo e ne faccio la proiezione esistenziale ottengo un insieme r.e. $x \in A$ sse $\exists y, (x, y) \in B$. Posso verificare che (x, y) sia in B perché B è ricorsivo, con un algoritmo del genere ad esempio: $\mu y, (x, y) \in B$.

Se io faccio una ricerca in un dominio ricorsivo non è detto che questa termini. Se potessi dare dei bound alla mia ricerca o avessi altre ipotesi potrei trasformare il mio algoritmo di semidecisione in un algoritmo di decisione, ma in generale non è questo il caso. Avremo in generale una funzione di semidecisione.

L'altra conseguenza importante è che qualunque problema semidecidibile può essere visto come una ricerca in uno spazio decidibile.

È utile vedere y come certificato di x, c_x. Il fatto che x appartenga ad A è cerficato da c_x. Se $x \in A$ sappiamo che questo ceritificato esiste ma non sappiamo quale sia. La sua ricerca non è però un problema decidibile.

Prendiamo per esempio la logica del prim'ordine. Se ho una formula in generale non è decidibile se questa sia dimostrabile. Tuttavia abbiamo un algoritmo che genera tutte le formule dimostrabili nella logica del prim'ordine. Dimostrare F con questo algoritmo significa andare a cercare, nell'insieme delle formule generate, un certificato per F.

Un altro esempio è: dato un programma datemi un certificato per la sua appartenenza a K. Questo certificato potrebbe essere il tempo t in cui $\varphi_x(x)$ termina.

La dimostrazione procede proprio in questo modo:

Dimostrazione. A è r.e. sse $A = dom(\varphi_i)$. Possiamo vedere $A = \{x \mid \exists t, T^3(i, x, t)\}$ □

Il certificato non è un concetto ben definito, sta a noi decidere cosa sia un certificato. Un certificato per un teorema potrebbe benissimo essere sia una prova che semplicemente la dimensione della prova.

Ritorneremo su questo concetto quando parleremo di $\mathbb{P}$ e $\mathbb{NP}$. Vedremo che $\mathbb{NP}$ è l'insieme dei linguaggi che sono proiezione esistenziale di un qualche linguaggio in $\mathbb{P}$.

La caratterizzazione $A = dom(\varphi_i)$ è così importante che abbiamo della notazione specifica per indicarlo: W_i. Possiamo con questa notazione definire K come $K = \{i \mid i \in W_i\}$. L'importanza di questa caratterizzazione è data dal fatto che essa induce una enumerazione di tutti gli insieri r.e., basata sui domini di convergenza delle funzioni della mia enumerazione delle funzioni calcolabili.

Capitolo 6

I teoremi di Rice e di Rice-Shapiro

Andiamo ora a vedere un teorema importante: il teorema di Rice.

Pensiamo alla proprietà $P(i) =$ "$\forall n, \varphi_i(n) \geq k$". Possiamo già intuire dalla quantificazione universale che la mia proprietà non è decidibile. Inoltre rimane il problema della divergenza. Un'altra proprietà interessante è analoga ma ha il quantificatore esistenziale al posto di quello universale. Questa è semidecidibile sicuramente, muovendosi con dove-tailing su input e tempo. È decidibile? Intuitivamente no, dovrei fare una ricerca su uno spazio infinito. Se sono fortunato in questi casi ho una proprietà semidecidibile.

Quello che andiamo a dimostrare adesso è che riguardo alle proprietà dei programmi non possiamo decidere niente. Non esiste alcuna proprietà decidibile, ad esclusione delle proprietà di falsità/verità costanti. Questo è dimostrabile in generale.

Questo qua è il succo del teorema di Rice.

6.1 Proprietà estensionali

Come caratterizziamo il comportamento delle funzioni calcolate dai programmi? Nel seguente modo: data una proprietà P ci chiediamo se $\varphi_i = \varphi_j$ implicihi $P(i) = P(j)$. Se questo è il caso allora dico che P è estensionale rispetto a φ. È importante il legame con l'enumerazione, che dato un numero mi restituisce la funzione calcolata dal programma con indice quel numero.

Possiamo caratterizzare un predicato con l'insieme degli indici delle funzioni per cui il predicato è vero. Parliamo in generale di insiemi estensionali. A è estensionale se c_A è tale che $\varphi_i = \varphi_j \implies c_A(i) = c_A(j)$. Gli insiemi estensionali sono chiusi rispetto a complementazione, unione e intersezione; formano quindi un'algebra di Boole.

φ è una funzione dai naturali alle funzioni parziali calcolabili: $\varphi : \mathbb{N} \to \mathcal{PC}$. Prendiamo $B \subseteq \mathcal{PC}$. A è estensionale se $A = \varphi^{-1}(B)$. Queste due caratterizzazioni sono equivalenti. La dimostrazione è lasciata al lettore.

6.2 Il teorema di Rice

Teorema 6.1. *Un insieme A estensionale è ricorsivo sse $A = \emptyset$ o $A = \mathbb{N}$ (o, equivalentemente, A è banale).*

Banale in matematica significa solitamente degenere.

Dimostrazione. L'implicazione inversa è banale. Dimostriamo solo quella diretta. Sia A estensionale. Supponiamo che A sia ricorsivo. Vogliamo dimostrare che A è vuoto oppure $A = \mathbb{N}$. Procediamo per assurdo. Supponiamo che $A \neq \emptyset$ e $A \neq \mathbb{N}$. Esistono allora $a_0 \in A$ e $a_1 \notin A$. Sia m un indice per la funzione che diverge sempre. Abbiamo due casi: o $m \in A$ o $m \in \overline{A}$. I due casi sono assolutamente simmetrici. Supponiamo, senza perdita di generalità, che $m \in \overline{A}$. A seconda che $\varphi_i(i)$ converga o diverga voglio costruire un programma che ha lo stesso comportamento di m in un caso e lo stesso comportamento di a_0 nell'altro. Vogliamo g tale che:

$$g(i,x) = \begin{cases} \varphi_{a_0}(x) & \text{se } \varphi_i(i) \downarrow \\ \varphi_m(x) & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Per s-m-n abbiamo $\varphi_{s(i)}(x) = g(i,x)$. Come calcoliamo $g(i,x)$? Col seguente algoritmo: $g(i,x) = \varphi_i(i); \varphi_{a_0}(x)$. Di conseguenza g è calcolabile.

Ci chiediamo ora: $s(i) \in A$? Dipende se $\varphi_i(i)$ converge. $s(i) \in A \iff \varphi_i(i) \downarrow$. Ma quindi anche K sarebbe ricorsivo. Ma questo è assurdo. □

6.3 Teorema di Rice-Shapiro

6.3.1 Proprietà compatte e monotone

Cerchiamo ora di generalizzare il teorema di Rice. Cosa possiamo semidecidere del comportamento dei programmi, tenuto vero quanto espresso dal teorema di Rice?

Prendiamo la proprietà essere totali: $P(i) =$ "φ_i è totale". Intuitivamente questa proprietà non è semidecidibile, perché dovrei esplorare tutti gli input prima di rispondere.

Proviamo con il complementare. Con un pò di abuso di notazione scriviamo $\overline{P}(i) =$ "φ_i non è totale" = "$\exists n, \varphi_i(n) \uparrow$". La divergenza non è testabile: si tratta sempre di una ricerca in uno spazio infinito, dove l'infinità è data dal tempo. Per questo motivo neanche questa proprietà è semidecibile, almeno a livello intuitivo.

Cosa posso sicuramente semidecidere? La convergenza puntuale ad esempio. L'algoritmo è semplice: lancio il programma i sul mio input x e aspetto. Se converge mi restituirà qualcosa, altrimenti divergerà.

Intuitivamente, le proprietà semidecidibili sono i test che riguardano la convergenza su un numero finito di input e i risultati relativi. Questo assomiglia un pò alla procedura di testing: osserviamo il comportamento del programma su un numero finito di input e verifichiamo se passa il mio test. È un testing semidecidibile. La cosa importante è che sia finito, non è neanche richiesto che sia determinato.

Ad esempio, prendiamo $P(i) =$ "$\exists n, \varphi_i(n) \downarrow$". Questo test è chiaramente semidecibie, mediante dove-tailing su n e tempo.

Vogliamo ora formalizzare l'idea di cosa è semidecidibile e cosa no. Cerchiamo alcune proprietà interessanti che sono soddisfatte da questi test.

Ci serve un ordinamento tra funzioni. Cerchiamo una nozione di ordinamento basata sul grafo delle funzioni. Diciamo che $\varphi_i \subseteq \varphi_j$ se $\mathit{grafo}(\varphi_i) \subseteq \mathit{grafo}(\varphi_j)$. In simboli questo corrisponde a $\forall n \forall m, \varphi_i(n) = m \implies \varphi_j(n) = m$. Quand'è che j può avere un comportamento diverso da i? Quando i diverge. In questo senso j è una "estensione" di i.

$\varphi_i \subseteq \varphi_i$, in base alla nostra definizione. φ_j è, in un certo senso, più informativa di φ_i, dato che dove φ_i è definita il suo valore è identico a quello di φ_j, e dove φ_i non è definita φ_j può darmi una nuova informazione.

Qualunque funzione è un'estensione della funzione che diverge sempre. Sia $P(i)$ una proprietà "testabile". Supponiamo che $P(i)$ e supponiamo che $\varphi_i \subseteq \varphi_j$. Cosa possiamo dire su φ_j rispetto a P? Per fissare le idee, prendiamo $P(i)$ $= \varphi_i$converge su tutti i numeri minori di 100. Supponiamo esista un a per cui vale P. Se ho che b estende a, ovvero $\varphi_a \subseteq \varphi_b$, allora necessariamente $P(b)$: qualunque estensione di un programma che converge su input minori di 100 continuerà a farlo.

Un predicato che rispetta la proprietà appena discussa è detto monotono. La monotonia è una proprietà del mio predicato P se per ogni estensione del mio programma a per cui vale P continua a valere la proprietà P.

Vediamo ora un'altra caratteristica che può avere un predicato estensionale. Supponiamo di avere il nostro test e che il test sia vero per un certo programma a. Allora, se esiste un b tale φ_b è una restrizione finita di φ_a e $P(b)$, diciamo che P è compatto. Equivalentemente possiamo esprimere la compattezza come $\exists b$ tale che $\varphi_b \subseteq \varphi_a$, φ_b è finito (come grafo) e $P(b)$. Questa proprietà dei predicati estensionali è detta compattezza.

Abbiamo dei massimi nell'ordinamento basato su estensione? Sì, tutte le funzioni totali, dette in questo contesto massimali. Non sono però in relazione tra loro, sono distinte.

// TODO RIPORTARE SCHEMA (Vedi schema sul quaderno azzurro).

Vediamo ora alcuni esempi di proprietà e chiediamoci quali sono compatte e quali monotone (e quali entrambe).

- $P(i) =$ "φ_i è totale". È monotona? Sì. È compatta? No.
- $P(i) =$ "φ_i non è totale". È monotona? No. È compatta? Sì.

- $P(i)$ = "$4 \in cod(\varphi_i)$". È monotona? Sì. È compatta? Sì.
- $P(i)$ = "$cod(\varphi_i) = 4$". È monotona? No. È compatta? Sì.

Dimostreremo che monotonia e compattezza sono condizioni necessarie ma non sufficienti per la semidecidibilità.

Prendiamo $P(i)$ = "$dom(\varphi_i)$ finito". È monotona? No. È compatta? Sì. Per il complementare? È monotono? Sì. È compatto? No.

Prendiamo $P(i)$ = "$dom(\varphi_i)$ infinito e $\overline{dom(\varphi_i)}$ finito (?)". È monotona? No. È compatta? No.

Una proprietà come $P(i)$ = "φ_i è primitiva ricorsiva" non è neppure semidecidibile, oltre a non essere decidibile per il teorema di Rice. Il motivo è che $P(i)$ non è compatta: il formalismo primitivo ricorsivo mi permette di definire solo funzioni totali.

6.3.2 Il teorema di Rice-Shapiro

I nomi di questo teorema variano nella letteratura. Per noi è il teorema di Rice-Shapiro.

Il simbolo dello yin-yang mi deve sottolineare l'impossibilità di separare le due aree, perché le due cose che sto tentando di separare sono processi in movimento che continuamente si trasformano nell'una e nell'altra. Non posso operare una distinzione tra le due.

I due cerchi rappresentano degli intorni che non cambiano, che appartengono all'altra area. Da un punto di vista topologico sono due aperti. Due aperti non possono dividere uno spazio, perché il complementare di un aperto è un chiuso.

Possiamo immaginare un cerchio come un intorno di tutti i programmi che hanno lo stesso comportamento di un certo programma. Se cerchiamo di dividere l'insieme di tutti i programmi in due spazi non possiamo aspettarci di poterlo fare in modo algoritmico. Questo è il succo del teorema di Rice. La proprietà di estensionalità, che è una proprietà di chiusura, non ci permette di fare questa divisione.

Nella mia dimostrazione cerco un s che mi permette di trovare una funzione che si comporta come a o m a seconda del comportamento di $\varphi_i(i)$. Questo lo posso fare, è più debole di chiedere che $s(i)$ mi restituisca a o m a seconda del comportamento di $\varphi_x(x)$ e sfrutta l'estensionalità. La prima cosa posso calcolarla, la seconda no.

Teorema 6.2. ***Teorema di Rice-Shapiro.*** *Sia A un insieme estensionale. Se A è r.e. allora:*

- *A è monotono*
- *A è compatto*

Dimostrazione. La monotonia mi dice che $\forall i, j$ se $i \in A$ e $\varphi_i \subseteq \varphi_j$ allora $j \in A$. Supponiamo che A sia estensionale e r.e. Supponiamo che A non sia monotono. Allora devono esistere i, j tali che $i \in A$, $\varphi \subseteq \varphi_j$ e $j \notin A$. Proveremo a

costruire un programma che si comporta come i in un caso e come j nell'altro caso. Costruiamo h totale e calcolabile che vogliamo si comporti come i se $\varphi_x(x) \uparrow$, come j se $\varphi_x(x) \downarrow$. Supponiamo di averla già: vedremo poi se possiamo calcolarla. Avremmo che $\varphi_x(x) \in \overline{K} \iff h(x) \in A$. L'implicazione inversa di questo $\iff$ può essere dimostrata dimostrando $\neg B \implies \neg A$. Ma questo vuol dire che se A fosse semidecidibile avrei un modo per calcolare, in modo semidecidibile, l'appartenenza a $\overline{K}$.

Tornando alla definizione del programma che calcola h, con s-m-n passiamo a $\varphi_{h(x)} = g(x,y) = \varphi_i(y)||(\varphi_x(x)\varphi_j(y))$. Questo programma ha lo stesso comportamento di j anche se $\varphi_x(x)$ converge e termina prima $\varphi_i(y)$ perché j è un'estensione di i.

Questa dimostrazione non è costruttiva. È costruttiva invece la dimostrazione per contraddizione: A non monotono $\implies$ A non r.e.

Questa dimostrazione implica, come corollario, il teorema di Rice.

Passiamo alla compattezza. Questa mi dice che $\forall i, i \in A \implies \exists j, \varphi_j \subseteq \varphi_i, \varphi_j$ è finita, $j \in A$.

Andiamo nuovamente per assurdo. Andiamo a negare la compattezza di A: $\exists i, i \in A \forall j, \varphi_j \subseteq \varphi_i, \varphi_j$ è finita $\implies j \notin A$.

Vogliamo costruire una funzione parametrica in x tale che si comporti come i se $\varphi_x(x) \uparrow$ e come una qualsiasi restrizione finita di i quando $\varphi_x(x) \downarrow$. La conclusione della dimostrazione è identica a quella della monotonia, quindi concentriamoci su h.

Utilizzando s-m-n abbiamo $\varphi_{h(x)} = g(x,y)$, con

$$g(x,y) = \begin{cases} \varphi_i(y) & \text{se } T^3(x,x,y) = \text{False} \\ \uparrow & \text{se } T^3(x,x,y) = \text{True} \end{cases}$$

Analizziamo il comportamento di questo programma. Ipotizziamo che $\varphi_x(x)$ sia divergente. Siamo nel primo caso, quindi il comportamento del mio programma è identico a quello di φ_i. Se $\varphi_x(x)$ converge invece ad un certo punto T mi restituirà True e da lì in poi la risposta rimarrà quella. A quel punto la mia funzione diverge. Questa corrisponde ad una restrizione finita di i: non so quanto lunga, so che esiste. Questa funzione è calcolabile, quindi h è calcolabile, da cui l'asserto. □

Capitolo 7

Teorema del punto fisso

7.1 Il teorema del punto fisso di Kleene

I seguenti sono commenti alle slide.

7.1.1 Slide 102

La parte più a destra dell'uguale è la definizione della mia funzione g.

Se prendiamo per f la funzione successore ho per forza che nel mio ordinamento avrò due programmi adiacenti che si comportano nella stessa maniera. In questo modo non è possibile creare un ordinamento tale che questo non si verifichi. La ragione fondamentale è che non ho controllo sul comportamento dei programmi quando vado ed enumerarli.

Punti fissi e ricorsione sono fondamentalmente la stessa cosa vista da due punti di vista diversi.

La ricorsione è legata alla possibilità che nel corpo del mio programma sia visibile la funzione che sto definendo.

Supponiamo di voler costruire un programma che stampi se stesso, ovvero che stampi il suo indice. Per semplicità lo vogliamo costante, ovvero tale che per ogni input stampi il suo indice: $\forall x, \varphi_p(x) = p$.

Prendiamo $g(i, x)$ così definita:

$$g(i, x) = i$$

Per s-m-n otteniamo una classe di programmi costanti $\varphi_{h(i)}(x)$. Questi però stampano sempre i, anche se $h(i)$ è diverso da i, quindi non vanno bene. Sfruttando il teorema del punto fisso abbiamo però che esiste $\varphi_p(x) = \varphi_{h(p)}(x) = p$. Di conseguenza è sufficiente prendere il punto fisso di h, p.

Se io voglio realizzare un comportamento ricorsivo posso farlo con il teorema del punto fisso di Kleene. Supponiamo di voler una funzione calcolabile φ_p tale che $\varphi_p(x) = f(\varphi_p(x))$, per qualche f. Si può? Sì. Abbiamo che $g(i, x) = f(\varphi_i(x))$. Ma ora per s-m-n abbiamo che questo corrisponde a $\varphi_{h(i)}(x)$. Ma ora

abbiamo che $\varphi_p(x) = \varphi_{h(p)}(x) = f(\varphi_p(x))$, se prendiamo come p il punto fisso di h.

In generale per punto fisso di una funzione intendiamo un input x tale che $f(x) = x$. Il nostro senso è un pò particolare perché è un punto fisso estensionale.

Il teorema dice fondamentalmente che $\forall f$ tot. calc. $\exists m, \varphi_{f(m)} = \varphi_m$.

Sfruttando questo risultato vogliamo costruire un programma che dato i mi restituisca qualcosa diverso dalla funzione φ_i per almeno un input, ovvero vogliamo i tale che:

$$\exists x, g(i, x) \neq \varphi_i(x).$$

Proviamo con $g(i, x) = \varphi_i(x)+1$. Per s-m-n esiste h tale che $\varphi_{h(i)}(x)$ identico a g. Funziona? No, se i è un indice per la funzione sempre divergente. Si può fare di meglio? No, per il teorema del punto fisso. Per qualunque trasformazione di programmi ho un punto fisso tale che $\varphi_{h(m)} = \varphi_m$. Non ho speranza di fare una modifica effettiva ed uniforme tale che il mio programma sia diverso da quello di partenza, e questo vale per ogni programma.

Vediamo una nuova dimostrazione del teorema di Rice basata sul teorema del punto fisso di Kleene. È concettualmente diversa da quella vista in precedenza.

Sia A un insieme estensionale. A è ricorsivo sse è banale. Supponiamo per assurdo che A sia ricorsivo ma non banale, ovvero esista $i \in A$ e $j \in \overline{A}$. Posso definire h tale che:

$$h(x) = \begin{cases} j & \text{se } x \in A \\ i & \text{se } x \notin A \end{cases}$$

Questa funzione sarebbe calcolabile per la ricorsività di A. Per il teorema del punto fisso, essendo h totle e calcolabile, deve esistere m tale che $\varphi_m = \varphi_{h(m)}$. Ci chiediamo ora, dove sta m? Se $m \in A$ allora stanno in A anche tutti i programmi che si comportano come m, tra cui anche $h(m)$. Ma se $m \in A$ allora $h(m) = j$, quindi $h(m) \notin A$. Se invece $m \notin A$ allora $h(m) \notin A$. Ma per definizione di h si ha che $h(m) = i \in A$. Ecco la mia contraddizione.

Tutte le volte che cerchiamo programmi che dipendono, intensionalmente o meno, dall'indice del programma che sto cercando devo fare riferimento al teorema del punto fisso.

7.2 Il secondo teorema del punto fisso

Pensiamo ora ad una funzione binaria $f(x, y)$. Possiamo generalizzare il teorema del punto fisso:

Teorema 7.1. *$\forall f^2$ totale calcolabile $\exists s$ totale calcolabile tale che $\forall y, \varphi_{f(s(y),y)} = \varphi_{s(y)}$*

Dimostrazione. La dimostrazione è nella slide 103 □

Capitolo 8

Riducibilità

8.1 *Many-to-one* reducibility

Abbiamo in precedenza visto dei procedimenti di "riduzione" da un insieme A a K per dimostare, ad esempio, che se A fosse ricorsivo anche K lo sarebbe. Andiamo ora a definire formalmente la nostra nozione di riducibilità. Ne esistono tante versioni diverse di questa nozione, noi ne vediamo una concettualmente semplice che va sotto il nome di m-riducibilità. La m sta per *many-to-one*; esiste ance la riduzione *one-to-one*.

Definizione 8.1. Siano $A, B \subseteq \mathbb{N}$. Un insieme A è m-riducibile a B ($A \leq_m B$) sse $\exists f$ totale calcolabile tale che $x \in A \iff f(x) \in B$.

Si parla di riducibilità perché possiamo ridurre la appartenenza ad A all'appartenenza a B. Per ogni x mi basta calcolare $f(x)$ e vedere se sta in B per sapere automaticamente se x appartiene ad A. Nella *one-to-one* reducibility si richiede anche che f sia iniettiva. È una nozione che troveremo molto simile nella parte di complessità, avremo solo degli ulteriori limiti sulla complessità di f.

Per dimostrare che $A \leq_m B$ devo fare due cose:

- per prima cosa devo definire $f : \mathbb{N} \to \mathbb{N}$ totale calcolabile;
- dopodiché va dimostrato che è una buona funzione di riduzione, ovvero va dimostrato il $\iff$ della definizione.

È un esercizio creativo, cambia per diversi A a B. La m-riducibilità è un ordine parziale. Infatti $A \leq_m A$, con la funzione identità, e se $A \leq_m B$ e $B \leq_m C$, allora, componendo le due funzioni di riduzione, otteniamo una funzione di riduzione da A a C:$A \leq_m C$.

Notazione: $A \leq_m^f B$ se A è riducibile a B attraverso la funzione f.

Se $A \leq_m B$ e $B \leq_m A$ allora scriveremo $A =_m B$. Intuitivamente sono equivalenti dal punto di vista della riducibilità.

La nozione di riducibilità ci dà un'idea di quanto complicato è un problema. Se $A \leq_m B$ allora B è tanto difficile quanto lo è A. Quanto fine è la mia misurazione dipende dalla potenza della mia nozione di riducibilità. Se la mia nozione di riducibilità non ha troppe pretese rischio di passare da un problema ricorsivo ad uno ricorsivamente enumerabile, e di avere molte riduzioni possibili non molto significative.

Tutti i problemi ricorsivi sono mutuamente riducibili ad esempio. Infatti supponiamo $x \in A \iff f(x) \in B$. A sto punto se B è ricorsivo ovviamente posso decidere l'appartenenza ad A, e quindi A è ricorsivo. Se invece B è r.e. allora, per lo stesso motivo, anche A risulta essere r.e.

L'altra implicazione non è vera. È possibile passare da un problema ricorsivo ad uno ricorsivamente enumerabile. Il minore o uguale ha il significato intuitivo di "A non è più difficile di B".

Supponiamo di avere due insiemi ricorsivi non banali. Mostriamo che sono sempre mutuamente riducibili l'uno all'altro. Dimostriamo solo $A \leq_m B$; il verso opposto usa le stesse ipotesi. Esistono allora $b_1 \in B$ e $b_2 \in \overline{B}$. È banale costruire ora $f(x)$ tale che:

$$f(x) = \begin{cases} b_1 & \text{se } x \in A \\ b_2 & \text{se } x \notin A \end{cases}$$

È una funzione totale calcolabile? Sì. Vale il sse della definizione di riducibilità? Anche, e si vede quasi subito. È importante l'ipotesi di non banalità di A e B. Non abbiamo usato l'ipotesi di ricorsività di B. Se B è un qualsiasi insieme non banale posso ridurre A a B. Senza altre ipotesi potrei non essere in grado di tornare indietro.

Prendiamo un insieme non ricorsivo come K e supponiamo che K sia riducibile ad A. Cosa abbiamo concluso? Che A non è ricorsivo. Analogamente se $\overline{K} \leq_m A$ allora A non è r.e.

Abbiamo infine che $A \leq_m B \iff \overline{A} \leq_m \overline{B}$.

8.2 Completezza

Passiamo ora alla nozione di completezza. Di solito è relativa ad una classe di problemi. Qua siamo interessati alla classe dei problemi semidecidibili.

Definizione 8.2. Diciamo che un certo problema B è completo (per i problemi r.e.) se

- B è r.e.
- $\forall A$, se $A \in \mathcal{RE}$, allora $A \leq_m B$

Potrei definire la stessa nozione per altre classi di problemi.

Il primo punto che ci interessa è, esistono dei problemi completi? La risposta è sì. Prendiamo ad esempio $K_1 = \{< x, y > |\ \varphi_x(y)\downarrow\}$. È banalmente semidecibile. È completo? Supponiamo che A sia r.e. Esiste quindi il semidecisore di A,

s_A. Supponiamo che sia il programma di indice a nella mia enumerazione dei programmi: $\varphi_a(x)$. Quindi $x \in A$ sse $\varphi_a(x) \downarrow$ sse $< a, x > \in K_1$. La funzione f che mi interessa è la funzione che mappa x nella coppia $< a, x >$: $x \mapsto < a, x >$.

Ne esistono altri? Si, ad esempio K è completo. Si può dimostrare direttamente. Noi però dimostreremo che $K \equiv_m K_1$. È facile vedere che $K \leq_m K_1$. La mia f è quella tale che $x \mapsto < x, x >$. Abbiamo che $x \in K \iff \varphi_x(x) \downarrow \iff < x, x > \in K_1$.

La parte $K_1 \leq_m K$ è un pò più tricky. Vogliamo catturare un comportamento puntuale sulla diagonale. Questo è difficile; possiamo piuttosto catturare un comportamento uniforme che valga ovunque, e che quindi in particolare valga anche per la diagonale.

Sia f tale che

$$f(x, y, z) = \begin{cases} 1 & \text{se } \varphi_x(y) \downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Utilizzo ora s-m-n sulla coppia $< x, y >$: esiste $\varphi_{s(x,y)}(z)$ con lo stesso comportamento. La mia funzione s è quella che cerco (questo è il mio claim).

Supponiamo che $(x, y) \in K_1$. Allora $\forall z \varphi_{s(x,y)}(z) \downarrow$. In particolare converge anche sulla diagonale. In particolare $\varphi_{s(x,y)}(s(x, y)) \downarrow$, e qindi $s(x, y) \in K$. Inoltre $(x, y) \notin K_1 \implies \forall z \varphi_{s(x,y)}(z) \uparrow \implies \varphi_{s(x,y)}(s(x, y)) \uparrow \implies s(x, y) \notin K$.

Come conseguenza abbiamo che tutti gli insiemi completi sono mutuamente riducibili: A e B completi $\implies A \equiv_m B$.

8.3 Insiemi produttivi e creativi

Cerchiamo ora di dare altre caratterizzazioni degli insiemi completi. Vediamo se hanno delle proprietà interessanti. Una proprietà curiosa è la creatività.

Definizione 8.3. Sia $A \subseteq \mathbb{N}$. A è produttivo se esiste f totale calcolabile, detta funzione di produzione, tale che comunque prendo un sottoinsieme $W_i \subseteq A$, si ha che $f(i) \in A \setminus W_i$.

Ricordiamo che W_i è la numerazione dei sottoinsiemi r.e. basata sul dominio di convergenza della funzione i della mia numerazione delle funzioni calcolabili.

Cos'è che produce la funzione di produzione? Uno potrebbe essere interessato a dare una approssimazione r.e. di questi insiemi. Cosa intendiamo? Dare un sottoinsieme di A ricorsivamente enumerabile. f mi produce una dimostrazione, per ogni approssimazione, dell'incompletezza della mia approssimazione in funzione dell'algoritmo con cui enumeriamo i programmi. f è unica, definita a prori dall'insieme A. Questo concetto è stato suggerito dai problemi di incompletezza logica.

C'è una caratterizzazione degli insiemi completi attraverso la produttività.

Possiamo pensare ad A come all'insieme delle formule vere dell'aritmetica. Quando costruiamo un sistema formale per ragionare sull'aritmetica, ovvero un sistema di assiomi fondamentalmente e le regole logiche per passare alle conseguenze logiche degli assiomi, avremo, per il teorema di incompletezza,

che ci saranno sempre formule vere che non sono però dimostrabili nel sistema formale.

È una tecnica simile alla diagonalizzazione.

Un insieme produttivo non è, per sua natura, r.e.; altrimenti potrei approssimarlo con l'insieme stesso.

Definizione 8.4. Un insieme $A \subseteq \mathbb{N}$ è creativo se A è r.e. e $\overline{A}$ è produttivo.

Non tutti gli insiemi produttivi hanno un complementare r.e.

K è un esempio di insieme creativo. Per dimostrarlo ci manca solo da dimostrare che $\overline{K}$ è produttivo. Questo è semplice però perché la funzione di produzione per $\overline{K}$ è la funzione identità.

Sia $W_i \subseteq \overline{K}$. Dobbiamo dimostrare che $i \in \overline{K}$ e $i \notin W_i$.

Per la prima appartenenza andiamo per assurdo. Supponiamo che $i \in K$. Allora $i \in W_i$. Ma essendo W_i sottinsieme di $\overline{K}$ si ha $i \in \overline{K}$, il che è assurdo. Quindi $i \in \overline{K}$.

Per la seconda appartenenza andiamo nuovamente per assurdo. $i \in W_i \implies i \in K \implies \bot \implies i \notin W_i$.

8.4 Relazione tra creatività e completezza

Nota: La seguente è l'ultima dimostrazione richiesta per l'orale.

Da qui in poi verrà usato il simbolo $\leq$ sottintendendo $\leq_m$. Vediamo un risultato importante per gli insiemi produttivi

Teorema 8.1. *Un insieme $A \subseteq \mathbb{N}$ è produttivo $\overline{K} \leq A$.*

Questo implica come corollario che A è creativo sse A è r.e. e $K \leq A$. Di conseguenza insiemi creativi ed insiemi completi coincidono.

Dimostrazione. Dimostriamo il primo teorema. Abbiamo i due versi del $\iff$ da dimostrare. Il più semplice dei due è quello inverso: $\overline{K} \leq A \implies A$ è produttivo.

Sia quindi $\overline{K} \leq A$. Sia f la funzione di riducibilità di $\overline{K}$ in A. Sappiamo anche che $\overline{K}$ è produttivo, con funzione di produzione uguale ad *Id*. L'idea è di ritornare a $\overline{K}$ da A attraverso f. Dopodiché, utilizzando la funzione di produzione di K, troviamo un nuovo elemento che, attraverso alla funzione f, sta in A e non in W_i.

Ricordiamo che la controimmagine di una insieme A attraverso una funzione f è definita come $f^{-1} : \mathbb{N} \to \mathbb{N}$ tale che $f^{-1}(A) = x | f(x) \in A)$.

Consideriamo $f^{-1}(W_i)$, la controimmagine di W_i via f. Abbiamo che sicuramente $f^{-1}(W_i) \subseteq \overline{K}$ e che è r.e. È importante che f sia totale. Se calcolo la controimmagine di un insieme attraverso una funzione parziale potrei ottenere un insieme non ricorsivamente enumerabile. L'indice di $f^{-1}(W_i)$ può essere calcolato in maniera effettiva attraverso una funzione totale calcolabile h. Abbiamo quindi che corrisponde a $W_{h(i)}$. Pensiamo alla funzione $\varphi_i(f(x)) = \varphi_{h(i)}(x)$ per s-m-n. Dove sta $h(i)$? Sta in $\overline{K}$ ma non in $f^{-1}(W_i)$, essendo $\overline{K}$ produttivo

con funzione di produzione uguale a *Id*. Passando quindi, attraverso f, al corrispondente in A ($f(h(i))$) abbiamo che sta in A ma non in W_i. Quindi A è produttivo.

La dimostrazione poteva essere fatta con un altro insieme produttivo. In particolare, se B è produttivo e $B \leq A$ allora A è produttivo.

Passiamo ora al verso opposto: A produttivo $\implies \overline{K} \leq A$.

Sia A produttivo e sia f la funzione di produzione per A. Vado alla ricerca di $W_{s(i)}$ tale che:

$$W_{s(i)} = \begin{cases} \{f(s(i))\} & \text{se } i \in K \\ \emptyset & \text{se } i \in \overline{K} \end{cases}$$

Non sappiamo, per ora, se esiste s; supponiamo però di poterla definire. Per dimostrare la sua esistenza ricorreremo al teorema del punto fisso.

Vogliamo dimostrare che $i \in \overline{K}$ sse $f(s(i)) \in A$. Supponiamo che $i \in \overline{K}$. Allora $W_{s(i)} = \emptyset \subseteq A$. Posso applicare la produttività e quindi concludere che $f(s(i)) \in A$.

Supponiamo che $i \in K$. Allora $W_{s(i)} = \{f(s(i))\}$. Supponiamo per assurdo che $f(s(i)) \in A$. Allora $W_{s(i)} \subseteq A$. Sfruttando la produttività avremmo che $f(s(i)) \in A$ e che $f(s(i)) \notin W_{s(i)}$. Ma questo è assurdo. Perciò $f(s(i)) \notin A$.

La dimostrazione è incompleta, manca la dimostrazione dell'esistenza di s. Ci serve il secondo teorema di ricorsione. Partiamo da $g(i, z, x)$ tale che:

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

Per s-m-n esiste $h(i, z)$ tale che $\varphi_{h(i,z)}(x)$ ha lo stesso comportamento. Esiste ora s tale che $\forall i \varphi_{s(i)} = \varphi_{h(i,s(i))}$. Abbiamo quindi che

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

□

Dimostrazione. Dimostriamo il corollario. Sia A creativo. Allora A è r.e. e $\overline{A}$ è produttivo. Allora $\overline{K} \leq \overline{A}$. Ma questo implica che $K \leq A$.

Sia A r.e. e completo. Allora $K \leq A$. Allora $\overline{K} \leq \overline{A}$, quindi $\overline{A}$ è produttivo, di conseguenza A è creativo. □

8.5 Insiemi r.e. non completi

Supponiamo di avere un insieme A r.e. Vogliamo dimostrare che A non è ricorsivo. Posso condurre questa dimostazione sempre dimostrando che $K \leq A$? Limitatamente agli insiemi creativi/completi allora sì. Il punto è, esiste un insieme r.e. non creativo? Questo va sotto il nome di problema di Post. È rimasto aperto per un certo numero di anni e veniva considerato un problema difficile.

È un analogo del problema $\mathbb{NP}$ vs $\mathbb{P}$. Nella teoria della complessità però si sà molto di meno. È più facile per un insieme r.e. essere completo che il contrario.

Per rispondere a questa domanda abbiamo bisogno di due risultati intermedi.

Sia A un insieme r.e. infinito. Esiste B ricorsivo infinito sottoinsieme di A? Sì. Una caratterizzazione degli insiemi ricorsivi è che sono enumerabili in maniera crescente. Se prendo l'enumerazione f di A e la "taglio" in pezzetti crescenti e li uso per definire la mia enumerazione di B ho che questa è crescente, e quindi B è ricorsivo infinito. Più precisamente, la mia enumeraziona g di B sarà fatta nel modo seguente, se f è la mia enumerazione di A: $g(0) = f(0)$ e $g(x+1) = f(\mu y.f(y) \geq g(x))$.

Se vediamo l'estrazione di sottoinsiemi come un'operazione di focalizzazione possiamo focalizzarci su insiemi di complessità desiderata

//TODO RIPORTARE FIGURA DAL QUADERNO (vedi figura sul quaderno azzurro).

Sia A un insieme ricorsivo infinito. Allora $\exists B$ r.e. non ricorsivo tale che $B \subseteq A$.

Consideriamo una funzione di enumerazione f per A iniettiva. Consideriamo l'insieme B così definito:

$$B = \{f(\varphi_i) \mid i \in \mathbb{N}\}$$

B è l'immagine di K via f.

Abbiamo che B è sottoinsieme di A e che B è r.e.; è facile da vedere. Dobbiamo dimostrare che B non è decidibile. Questo è vero perché altrimenti K sarebbe decidibile. Per vederlo è sufficiente dimostrare che vale il sse $i \in K \iff f(\varphi_i(i)) \in B$. È facile.

Noi abbiamo intrapreso questo discorso perché cercavamo un insieme r.e. non completo. Una proprietà interessante degli insiemi produttivi è che ogni insieme produttivo contiene un sottoinsieme r.e. infinito.

Teorema 8.2. *Sia $A \subseteq \mathbb{N}$ produttivo. Allora $\exists B$ r.e. infinito tale che $B \subseteq A$.*

È una proprietà cruciale che ci serve perché costruiremo un insieme infinito che non contiene alcun sottoinsieme r.e. In pratica abbiamo un insieme tale che qualsiasi suo sosttoinsieme infinito è talmente caotico da non essere nemmeno r.e.

Dimostrazione. Per la dimostrazione useremo la produttività. Proviamo ad approssimare A. Inizieremo con l'insieme vuoto. Per la produttività esiste $a \in A$. Ora nella mia approssimazione includo pure a. Per produttività esiste $a' \in A$ fuori dalla mia approssimazione, e continuo così all'infinito. Questo procedimento è costruttivo e mi crea la mia approssimazione r.e. di A.

Questo procedimento ha un analogo con il tentativo di approssimare l'insieme delle conseguenze di un insieme di assiomi. Per l'incompletezza esiste una formula che non è dimostrabile dal mio sistema formale. A questo punto potrei aggiungere la validità di questa formula al mio sistema formale e crearne uno nuovo. Ma a questo punto ho un nuovo sistema formale, e il ragionamento precedente si applicherebbe identicamente. Quindi non potrò mai avere un insieme che contenga tutte le conseguenze dei miei assiomi.

Vogliamo dimostrare che $W_{h(i,a)} = W_i \cup a$. Più precisamente vogliamo dimostrare che esiste un modo effettivo per calcolare un insieme r.e. ottenibile mediante estensione di W_i con a. $W_{h(i,a)}$ è il dominio di una certa funzione $\varphi_{h(i,a)}$. Vogliamo quindi che $\varphi_{h(i,a)}(x) = g(i,a,x) = \varphi_i(x)|(x = a)$. In pratica voglio che converga se $x = a$ oppure se x fa parte del dominio di φ_i. g è effettivamente calcolabile e ho quindi un metodo effettivo per calcolare $W_{h(i,a)}$ ogni volta.

Possiamo ora costruire la mia sequenza di approssimazione. $W_m = \emptyset$. Sia s la funzione di produzione per A. Ho che $s(m) \in A \setminus W_m$. Passo quindi a $W_{h(m,s(m))=m_1} = \{s(m)\}$. La mia seconda approssimazione sarà $W_{h(m_1,s(m_1))} = s(m), s(m_1)$.

Se definiamo $next(x) = h(x, s(x))$ possiamo definire $g(x) = next^x(m_0)$ e abbiamo che B è uguale al codominio di g. □

8.5.1 Complessità di Kolmogorov

Se noi vogliamo trasmettere un'informazione, diciamo il numero n, abbiamo due modi per farlo: uno è trasmettere direttamente n, ad esempio con la sequenza di bit che rappresnta n, che ha un costo che dipende da n; un'altra possibilità è trasmettere un modo per costruire n. Ad esempio un algoritmo.

In particolare, immaginiamo un programma i tale che $\varphi_i(0) = n$. Posso trasmettere direttamente i. Qual è più conveniente? Intuitivamente quello più piccolo. Confronto i con n. Se i è più piccolo trasmetto i, altrimenti trasmetto n.

Definizione 8.5. La complessità di Kolmogorov di n, $K(n)$, è il minimo i tale che $\varphi_i(0) = n$ $(= \min\{i \mid \varphi_i(0) = n\})$.

È una astrazione di una problematica reale legata alla comprimibilità delle informazioni.

Il mio confronto è tra n e $K(n)$.

8.5.2 Numeri random

Definizione 8.6. $n \in \mathbb{N}$ è un numero random se $n \leq K(n)$.

Dobbiamo pensare a n come alla sua espansione decimale: una stringa di bit.

Perché usiamo la nozione di *Random* per descrivere questa caratteristica? Perché è una idea del concetto. Se ci fosse una grande regolarità nella stringa per n è chiaro che posso creare un programma piccolino che lo generi. Si parla di dati lawful o lawless. I dati che seguono una regola non saranno random. Se sono random sono talmente disordinati che non riesco a dare nessun metodo generativo più breve dei dati stessi. È un approccio formale alla nozione di random legata alla comprimibilità del dato.

Come l'abbiamo vista la nostra nozione di "casualità" dipende dalla nostra enumerazione dei programmi. Per numeri grandi però non dovrebbe cambiare niente se cambia la numerazione.

Un'altra questione è che la nostra nozione di random va bene per sequenze finite. Noi in generale vorremmo idealmente una nozione che valga per sequenze infinite. In questo contesto però ci è sufficiente.

La randomicità di un numero è decidibile? È semidecidibile? Potrei pensare di fare una ricerca limitata ad n per un programma che genera n. Il problema è che le mie funzioni sono parziali calcolabili, potrebbero divergere. Quindi sembrerebbe che la risposta sia no.

Cosa possiamo però semidecidere? Se un numero non è random. Se $\mathcal{R}$ è l'insieme dei numeri random allora abbiamo che $\overline{\mathcal{R}}$ è semidecidibile. La nostra congettura è che $\mathcal{R}$ non sia nemmemno semidecidibile.

Abbiamo un altro modo per dimostrare quanto detto su $\overline{\mathcal{R}}$, oltre al lanciare in parallelo tutti i φ_i e aspettare che uno di essi termini. Lanciamo $\varphi_i(0)$ e vediamo se è uguale a n. Se $\varphi_i(0) = n$ e $i < n$ allora restituisco n, altrimenti divergo. Questa funzione $g(< i, n >)$ è una funzione di numerazione parziale che dà fuori numeri che non sono random. Prima o poi tutti i numeri non random verranno enumerati.

Il nostro claim è che l'insieme dei numeri random non è produttivo.

Vogliamo dimostrare che $\mathcal{R}$ non è produttivo. Lo dimostriamo facendo vedere che $\mathcal{R}$ non contiene nessun sottoinsieme r.e. infinito.

Noi dimostriamo ciò dimostrando che in $\mathcal{R}$ non ci sia nessun sottoinsieme ricorsivo infinito. Grazie ai risultati dimostrati in precedenza è equivalente.

Supponiamo, per assurdo, che A sia ricorsivo e che $A \subseteq \mathcal{R}$. Definiamo $g(i, x) = \mu n, n \in A$ e $n > i$. L'idea è che A è ricorsivo, e quindi possiamo decidere $n \in A$, e possiamo andare alla ricerca del più piccolo n maggiore di i. Questa funzione è calcolabile e per s-m-n abbiamo che possiamo calcolarla con $\varphi_{h(i)}(x)$.

Esitste p tale che $\varphi_p(x) = \varphi_{h(p)}(x) = \mu n, n \in A$ e $n > p$, per il teorema del punto fisso. Ora mi chiedo $\varphi_p(0)$ è random? Dovrebbe essere random, dato che l'output sta in $A \subseteq \mathcal{R}$. Ma al tempo stesso $\varphi_p(0)$ è generato da p e, per costruzione, $p < \varphi_p(0)$. E quindi sarebbe non random. Assurdo.

Questa parte è legata al paradosso di Berry.

Un insieme che non contiene insiemi r.e. infiniti è detto immune e il suo complementare è detto semplice.

Abbiamo quindi che l'insieme dei numeri random non è produttivo. Di conseguenza l'insieme $\overline{\mathcal{R}}$ è r.e. non ricorsivo e non creativo, ovvero non completo.