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: $A \vee B = \overline{\overline{A \vee B}} = \overline{\overline{A} \wedge \overline{B}}$. Un altro modo è usano la funzione normalizzazione, che normalizza i numeri a 0 e 1:

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

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

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

```
x = 0
while $\lnot P(x)$:
    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 $\lnot P(x)$:
    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).