

6^a Lezione

Di seguito riporteremo ancora qualche esempio (e saranno gli ultimi) di implementazione di costrutti funzionali. Cominceremo con l'implementare resto e quofo di una divisione, per poi passare ad un algoritmo che simuli il comportamento di un ciclo WHILE – DO

6.1 Definizione del resto e del quofo di una divisione intera

Sappiamo che un qualsiasi numero intero puo' essere rappresentato nella forma $y = qx + r$. Quello che adesso cercheremo di fare sara' trovare il resto r della divisione x/y , che indicheremo con $rm(x,y)$. Procederemo per ricorsione primitiva, ricavando prima un modo ricorsivo per vedere $rm(x,y)$, quindi cercheremo di ricavare le relative funzioni $g(x)$ ed $h(x,y,z)$:

$$\begin{array}{l} \left\{ \begin{array}{ll} rm(x,0)=0 & \text{caso base} \\ y+1 = qx + r + 1 & \text{se } 0 < r+1 < x \\ y+1 = (q+1)x + 0 & \text{se } r+1 = x \end{array} \right. \longrightarrow \left\{ \begin{array}{ll} rm(x,0)=0 \\ rm(x,y+1) = \left\{ \begin{array}{ll} rm(x,y)+1 & \text{se } rm(x,y)+1 < x \\ 0 & \text{se } rm(x,y)+1 = x \end{array} \right. \end{array} \right. \\ \boxed{\left\{ \begin{array}{ll} g(x)=0 \\ h(x,y,z) = \left\{ \begin{array}{ll} z+1 & \text{se } z+1 < x \\ 0 & \text{se } z+1 = x \end{array} \right. \end{array} \right.} \end{array}$$

Figura 6.1

A questo punto, si capisce come l'implementazione di $g(x)$ sia molto semplice, mentre quella di $h(x,y,z)$ sia piu' complessa, e, in generale, legata alla nostra capacita' di decidere se $z+1=x$. Per questo motivo, cominciamo con l'implementare un algoritmo che, dati due naturali, riesca a decidere se sono uguali. Ad esempio, per vedere se $x=y$, basta vedere se $(x-y)+(y-x)=0$. Una implementazione di questo algoritmo sara':

$$=(x,y) = (- \wedge (P^2_{2,2} \wedge P^2_{1,1}) ; -) ; + ; !Sg$$

Come si vede, questo programma fornisce come risultato 1 se l'uguaglianza e' verificata, zero altrimenti. Detto questo, passiamo con l'implementare la funzione $h(x,y,z)$, il che' sara' molto piu' semplice. Infatti, bastera' il seguente algoritmo:

$$h(x,y,z) = (z+1) * (!Sg(z+1=x))$$

Che sara' implementato da:

$$h(x,y,z) = ((P^3_{3,3} ; S) \wedge P^3_{1,1}) ; (P^2_{1,1} \wedge =) ; (P^1_{1,1} \parallel !Sg) ; *$$

In modo del tutto analogo a quanto fatto per il resto, e' possibile implementare anche il quoto (qt) della divisione, ovvero il q della formula $y = qx + r$. Anche in questo caso si procede per ricorsione primitiva:

$$\begin{cases} qt(x, 0) = 0 \\ qt(x, y+1) = \begin{cases} qt(x, y) & \text{se } rm(x, y) + 1 < x \\ qt(x, y) + 1 & \text{se } rm(x, y) + 1 = x \end{cases} \end{cases} \longrightarrow \begin{cases} g(x) = 0 \\ h(x, y, z) = \begin{cases} z & \text{se } rm(x, y) + 1 < x \\ z + 1 & \text{se } rm(x, y) + 1 = x \end{cases} \end{cases}$$

Figura 6.2

a questo punto, procedendo in modo del tutto analogo al resto della divisione, possiamo implementare la funzione $h(x, y, z)$. Per non dilungarci eccessivamente omettiamo I passaggi di questa operazione.

6.2 Implementazione del costrutto WHILE – DO

Supponiamo di avere le seguenti funzioni:

- $f : N^n \rightarrow N^n$, che e' la nostra funzione da iterare.
- $g : N^n \rightarrow N$, la quale, applicata agli argomenti di f , serve per vedere se ho finito.

Definiamo ora la funzione $f^{=0}(x_1, \dots, x_n) : N^n \rightarrow N^n$, la quale controlla, ad ogni iterazione, se $g(x_1, \dots, x_n) = 0$. Se cio' e' falso viene applicata f a (x_1, \dots, x_n) , altrimenti si esce dal ciclo. Quindi, il ciclo di funzionamento sara' del tipo:

$x_1, \dots, x_n \rightarrow f(x_1, \dots, x_n) \rightarrow f(f(x_1, \dots, x_n)) \rightarrow \dots \dots \dots$ fino a quando non succede che $g(f \dots (f(x_1, \dots, x_n))) = 0$

Vediamo ora di esaminare piu' attentamente il comportamento di questa funzione:

Figura 6.3

$$f^{=0}(x_1, \dots, x_n) = \begin{cases} f^k(x_1, \dots, x_n) & \text{se } k \text{ e' il piu' piccolo} \\ \uparrow & \text{naturale tale che:} \\ \text{altrimenti} & \begin{cases} g(f^i(x_1, \dots, x_n)) = 0 & \text{(e' definito) con } i \leq k \\ g(f^i(x_1, \dots, x_n)) > 0 & \text{per } i < k \\ g(f^k(x_1, \dots, x_n)) = 0 & \text{per } i = k \end{cases} \end{cases}$$

Qui i e' il numero di iterazioni cui si e' arrivati

notiamo che questa funzione, per certi valori di x_1, \dots, x_n , ha soluzione indefinita. Cio' puo' avvenire sia nel caso il processo entri in un loop infinito (questo avviene se non esiste un k con le caratteristiche di cui sopra), sia nel caso in cui una tra le funzioni f o g sia indefinita. In questo caso, non sapendo come andare avanti, il nostro algoritmo si impianterebbe.

Probabilmente e' inutile sottolineare che l'implementazione di questo costrutto e' piuttosto complessa, e quindi, non essendo per noi di importanza fondamentale, la tralasciamo.

6.3 Simulazione di una macchina a registri tramite paradigma funzionale

Vediamo a grandi linee quali sono le idee che stanno sotto la simulazione di una macchina a registri attraverso il nostro paradigma funzionale.

Consideriamo una macchina a n registri e prendiamo il caso in cui la funzione in entrata crei la configurazione iniziale rappresentata dalla sequenza $(1, x, 0, 0, \dots)$ dove 1 e' il contatore di programma all'inizio, mentre x e' il valore del primo registro, che e' anche l'unico ad essere al momento utilizzato.

Da cio' possiamo dedurre un programma che implementi questa particolare funzione di entrata, che sara':

Funz Entrata (x) = $(0; S) \parallel P_{1,1}^1 \parallel 0 \parallel \dots n-1$ volte ... $\parallel 0$

Parallelamente alla funzione d'entrata, potremmo implementare anche la funzione d'uscita, la quale sara' una semplice proiezione dei registri che contengono I dati di output.

Per quanto riguarda il ciclo di funzionamento vero e proprio della macchina, questo consistera' nell'iterare una funzione che applichi le istruzioni elementari solo nel caso in cui il contatore di programma corrisponda col numero dell'istruzione elementare. Se corrisponde bisogna eseguire l'istruzione. L'iterazione si fermera' quando $P_{n+1,1}^1 > S$, dove S e' la lunghezza del programma, cioe' quando il contatore di programma finisce di puntare istruzioni.

Con questo abbiamo finito il nostro studio sul paradigma funzionale, che abbiamo visto essere equivalente alle macchine a registri e di Touring. Quindi questi sono 3 mezzi con cui implementare qualsiasi algoritmo.

6.4 Il paradosso di Russell

Prendiamo l'insieme R definito come $R = \{x: x \notin X\}$. Questo insieme, nel modo in cui noi lo concepiamo, e' in realtà un insieme che non contiene elementi.

Ci chiediamo ora se $R \in R$, il che e' vero per definizione di appartenenza ad un insieme, ma e' falso se vediamo come e' fatto R . Quindi, arriviamo ad una scrittura del tipo $R \in R \Leftrightarrow R \notin R$, detta **paradosso di Russell**.

Vedremo che paradossi di questo tipo saranno piuttosto frequenti e ci saranno molto utili per fondare la nostra teoria della calcolabilità.

6.5 Programmi che prendono in input programmi

Iniziamo ora un discorso che renderemo nei prossimi capitoli più chiaro e completo. Abbiamo visto che i programmi sono in corrispondenza biunivoca coi numeri naturali (questo perché le stringhe sono numerabili, e un programma può essere visto come una lunga stringa). Ora, dal momento che noi consideriamo macchine i cui input siano sequenze finite di numeri, allora anche tutti i possibili input, come i programmi, sono numerabili, in quanto sequenze finite di numeri sono anch'esse stringhe. Quindi, dal momento che posso rappresentare un programma con un naturale, potrò dare in pasto ad un programma un altro programma, anziché il solito input numerico.

Ad esempio, al programma numero 3 potrò fornire, anziché l'input numero 4, il programma numero 4, in quanto c'è corrispondenza biunivoca tra le due cose. In particolare, un caso interessante si verifica se al programma 3 fornisco come input il programma 3, cioè lui stesso.

In proposito, consideriamo ora un certo $x \in N$, e prendiamo in considerazione P_x , il programma in corrispondenza biunivoca col naturale x . Prendiamo ora $\Phi_x(y): N \rightarrow N$, la funzione che applica P_x ad y , in quanto. In questo caso, input ed output sono numeri naturali, e non sequenze di numeri, perché abbiamo visto sopra che le sequenze di numeri, essendo stringhe, sono in corrispondenza biunivoca con N . Costruisco ora:

$$f(x) = \begin{cases} \Phi_{x(x)+1} & \text{se } P_x \downarrow x \\ 0 & \text{altrimenti} \end{cases}$$

Dimostreremo che questa funzione non è calcolabile, e che quindi, non esiste nessun programma che possa essere implementato per calcolarla.

7^a Lezione

Vediamo ora di capire meglio le nozioni che sono state accennate nel paragrafo 6.5. In particolare, cominceremo col vedere come si puo' associare un numero naturale ad un singolo programma, per poi passare a vedere una rapida carrellata di funzioni che non sono calcolabili, cioe' che non sono implementabili da alcun programma.

7.1 Associazione di numeri a programmi

Ormai sappiamo che ad ogni programma puo' essere associato un numero naturale. Adesso cercheremo di trovare qualche metodo concreto per effettuare tale associazione.

Dato un certo linguaggio di programmazione, si avranno, in generale, un certo numero di strutture dati concrete attraverso le quali codificare I dati del problema (che sono astratti). Chiameremo **codifica** la rappresentazione di dati astratti con dati concreti. Supponiamo, ad esempio, di avere un programma P scritto in uno dei tre linguaggi di programmazione visti, e di rappresentarlo attraverso un certo numero. In questo modo possiamo realizzare una codifica del tipo di dato programma.

Chiamiamo P l'insieme di tutti I programmi scrivibili con uno dei tre linguaggi visti, e consideriamo la funzione $f: P \rightarrow N$ tale che:

$$f(P) = \begin{cases} P & \text{se } P \downarrow 0 \\ x_1 := 0 & \text{se } P \uparrow 0 \end{cases}$$

Dimostreremo che questa funzione non e' calcolabile, ma per far questo e' indispensabile saper associare un numero naturale ad un programma. Iniziamo con l'associare (codificare) coppie di numeri naturali con un naturale. Cio' puo' essere fatto dalla seguente funzione:

$$\Pi: N^2 \rightarrow N$$

$$\Pi(m,n) = 2^m(2n+1) - 1$$

ad esempio, $\Pi(2,4) = 2^2(2*4 + 1) - 1 = 35$, quindi la coppia di naturali (2,4) sara' codificata dal naturale 35. Questo metodo di codifica delle coppie e' detto **effettivo** poiche' ad ogni numero naturale corrisponde sempre una certa coppia e viceversa.

Il prossimo passo che ci accingiamo a compiere e' quello di codificare le singole istruzioni di un programma tramite numeri naturali. Cio' puo' essere fatto, per quanto concerne una macchina a registri, che ricordiamo avere istruzioni di 4 tipi, nel seguente modo:

- **Assegnamento** $\lceil x_i := 0 \rceil \leftarrow \text{Codifica} \rightarrow 4(i-1)$
- **Successore** $\lceil x_i := x_{i+1} \rceil \leftarrow \text{Codifica} \rightarrow 4(i-1) + 1$
- **Trasferimento** $\lceil x_i := x_j \rceil \leftarrow \text{Codifica} \rightarrow 4(\Pi(i-1, j-1)) + 2$
- **Salto** $\lceil \text{if } (x_i=0) \text{ then goto } q \rceil \leftarrow \text{Codifica} \rightarrow 4(\Pi(i-1, q-1)) + 3$

dove $\Gamma_{oggetto}$ corrisponde al numero che codifica oggetto (in questo caso parliamo di istruzioni). Questa codifica delle istruzioni consiste nel rappresentare le istruzioni di tipo Assegnamento con multipli di 4, quelle di tipo Successore con multipli di 4 cui sommiamo 1, quelle di tipo Trasferimento con multipli di 4 cui sommiamo 2 e cosi' via. In questo modo, associamo ad ogni istruzione un numero naturale e viceversa.

Adesso vediamo due modi con cui e' possibile codificare i programmi.

7.1.1 Metodo del prodotto di numeri primi

Il primo metodo per codificare i programmi consiste, dato un programma P con istruzioni (I_1, \dots, I_n) , nel codificarlo con il prodotto:

$$\Gamma_{P} = \Gamma_{I_1} * \Gamma_{I_2} * \dots * \Gamma_{I_n}, \quad \text{con } 2, 3, \dots, n \text{ numeri primi}$$

Ad esempio, dato il programma con istruzioni:

- $I_1 = x_1 := 0$, da cui $\Gamma_{I_1} = 0$
- $I_2 = x_1 := x_2$, da cui $\Gamma_{I_2} = 4 (\Pi(0, 1)) + 2 = 10$

otteniamo $\Gamma_P = \Gamma_{I_1, I_2} = 2^1 * 3^{11} = 354294$, quindi questo programma di 2 istruzioni, secondo la nostra codifica, corrisponde al numero naturale 354294. Questa funzione, purtroppo, non e' suriettiva, in quanto vi sono dei numeri che non rappresentano alcun programma. Per questo motivo cerchiamo un algoritmo di codifica migliore.

7.1.2 Metodo delle potenze del 2

Se indichiamo Γ_{I_1} con a_1 , Γ_{I_2} con a_2 , ..., Γ_{I_n} con a_n , possiamo codificare il nostro programma P con il prodotto:

$$\Gamma_P = \Gamma_{I_1, I_2} = (2^{a_1} + 2^{a_2+1} + 2^{a_1+a_2+a_3+2} + \dots + 2^{a_1 + \dots + a_n + (n-1)}) - 1$$

Ad esempio, preso il programma di prova visto per il metodo precedente, otteniamo:

$$\Gamma_P = \Gamma_{I_1, I_2} = (2^0 + 2^{11}) - 1 = 2048$$

quindi il nostro programma di due istruzioni, con questo sistema di codifica, risulta essere corrispondente al numero 2048. Contrariamente al primo metodo di codifica, questo si puo' dimostrare essere una funzione biettiva da P (insieme dei programmi) ad N .

7.2 La macchina universale

Definiamo **macchina universale** la funzione $\Phi(n, x) : N^2 \rightarrow N$ che calcola:

$$\begin{cases} y & \text{se } P_n \uparrow x \\ \uparrow & \text{se } P_n \downarrow x \end{cases}$$

Diamo ora una breve spiegazione di come si comporta una macchina universale. Essa non e' altro che una funzione che prende in ingresso un programma P_n (che vediamo sotto forma della sua codifica n) e gli applica il secondo input, ovvero il numero naturale x . Se x e' un input definito per il programma P_n , verrà computato l'output corrispondente, ovvero y , altrimenti l'output sarà indefinito.

Spesso, quando il programma in ingresso alla macchina universale e' ben conosciuto, si usa indicare questa funzione sotto la forma $\Phi_n(x) = \Phi(n, x) : N \rightarrow N$, che e' esattamente analoga alla precedente, solo che in questo caso si intende che la macchina universale venga applicata a P_n .

7.3 Primo esempio di funzione non calcolabile

Daremo adesso la definizione di una funzione, che dimostreremo essere non calcolabile. In generale, per dimostrare che una funzione non e' calcolabile, bisogna dimostrare che non vi e' nessun algoritmo che la calcola. Cio'

in generale, si vede eseguendo una dimostrazione per assurdo, nella quale si assume che la funzione sia calcolabile sperando di arrivare ad una qualche contraddizione. In questo caso la dimostrazione sarebbe terminata.

Affermazione:

la seguente funzione non e' calcolabile:

$$f(x) = \begin{cases} \Phi_x(x) + 1 & \text{se } P_x \downarrow x \\ 0 & \text{altrimenti} \end{cases}$$

Dim:

Supponiamo per assurdo che $f(x)$ sia calcolabile. Allora esisterà un certo programma P che la calcola. Se la codifica di tale programma è il naturale m , allora P_m sarà il programma che calcola $f(x)$. Da questo deriva che $\Phi_m(x) = P_m(x) = f(x)$. Vediamo cosa succede valutando $\Phi_m(m)$, il che, in pratica, è un autoriferimento:

$$\Phi_m(m) = f(m) = \begin{cases} \Phi_m(m) + 1 & \text{se } P_m \downarrow m \\ 0 & \text{altrimenti} \end{cases}$$

allora, se $P_m \downarrow m$, si ha che $\Phi_m(m) = \Phi_m(m) + 1$, il che è assurdo perché sarebbe come dire che $\Phi_m(m)$ fornisce due output diversi. Dunque è impossibile che $P_m \downarrow m$. Dunque deve essere che $P_m \uparrow m$, ma allora $\Phi_m(m) = 0$, ma ciò vorrebbe dire che il programma in realtà è definito su m , il che abbiamo appena visto essere impossibile. *Cio' dimostra quindi che $f(x)$ non e' calcolabile.*

7.4 Deduzione di funzioni non calcolabili per ragionamento diagonale

Consideriamo la seguente figura:

Prog	0	1	2	...
Input	$\phi_0(0)$	$\phi_1(0)$	$\phi_2(0)$	
0	$\phi_0(1)$	$\phi_1(1)$	$\phi_2(1)$	
1	$\phi_0(2)$	$\phi_1(2)$	$\phi_2(2)$	
2				

Sequenza di numeri naturali più indefinito. Es.: 3, 5, 7, 9, ...

Figura 7.1

Questa è una tabella infinita che, applicando ad ogni programma esistente ogni possibile input, mostra tutti gli output generabili in natura. Di conseguenza questa tabella mostra tutto ciò che è umanamente possibile calcolare, per cui se riusciremo a dimostrare che una certa sequenza di output non appartiene a nessuna colonna della tabella, avremo trovato di conseguenza una funzione non calcolabile.

Consideriamo ora la sequenza infinita di numeri naturali (o di simboli di indefinito) generati dalla diagonale principale della tabella. Ora, modifichiamo tale sequenza in tutti i suoi elementi, e facciamo finta che essa corrisponda ad una colonna della tabella, cioè ad una colonna di output di un dato programma. Ma allora questa colonna dovrebbe fare già parte della tabella, perché essa è composta dagli output di tutte le funzioni calcolabili. Purtroppo, ci accorgiamo che questa colonna non può far parte della tabella, in quanto, ogni suo i -esimo elemento sarà diverso dell' i -esimo elemento della sequenza diagonale. Concludiamo, da questo, che la nostra sequenza di output non è generabile da alcun programma ed è dunque frutto di una funzione non calcolabile.

Più in generale, qualsiasi sequenza di naturali nata dalla modifica di tutti gli elementi della diagonale, determina una funzione non calcolabile. Da questo possiamo anche dedurre che le funzioni calcolabili sono molte meno delle funzioni non calcolabili.

7.5 Secondo esempio di funzione non calcolabile

In questo paragrafo dimostreremo che una funzione che prende in input un programma ed un numero determinando se detto programma e' definito su tal numero, non e' calcolabile.

Affermazione: (HALTING PROBLEM - PROBLEMA DELLA FERMATA)
la seguente funzione non e' calcolabile:

$$h(x,y) = \begin{cases} 1 & \text{se } P_x \downarrow y \\ 0 & \text{se } P_x \uparrow y \end{cases}$$

Dim:

Supponiamo per assurdo che h sia calcolabile da un certo programma P . Consideriamo ora la seguente funzione:

$$g(x,y) = \begin{cases} \uparrow & \text{se } P_x \downarrow y \\ 0 & \text{se } P_x \uparrow y \end{cases}$$

Ora, se $h(x,y)$ e' calcolabile da un certo programma P_m , allora anche $g(x,y)$ sara' calcolabile da un programma molto simile a P_m . Prima di vedere come costruire questo programma simile a P_m , modifichiamo g . Porremo:

$$g(x) = \begin{cases} \uparrow & \text{se } P_x \downarrow x \\ 0 & \text{se } P_x \uparrow x \end{cases}$$

Operiamo adesso a P_m quelle modifiche di cui parlavamo in precedenza, in modo da costruire un programma per $g(x)$. Per questo, basta aggiungere a P_m delle istruzioni che controllano il primo registro per vedere se il risultato della computazione e' 0 o 1. Qualora esso fosse 1 lo si sostituirebbe con \uparrow , cioe' si farebbe ciclare l'algoritmo. Abbiamo quindi ottenuto un certo P_k , leggermente diverso da P_m , che calcola $g(x)$. Questo si puo' anche affermare scrivendo $\Phi_k(x) = P_k(x) = g(x)$.

Ma da questo ricaviamo, seguendo la definizione di $g(x)$ e operando con il solito autoriferimento, che:

- $\Phi_k(k) = \uparrow$ se $P_k \downarrow k$, che e' impossibile perch e' $P_k \downarrow k$, per cui $\Phi_k(k)$ dovrebbe essere calcolabile.
- $\Phi_k(k) = 0$ se $P_k \uparrow k$, che e' impossibile perch e' in questo caso $\Phi_k(k)$ non dovrebbe avere output.

Queste 2 contraddizioni sono piu' che sufficienti per farci affermare che $h(x,y)$ non e' calcolabile.

7.6 Esercizi

Vediamo un esercizio sull'applicazione della macchina di Turing

Testo:

Realizzare un programma per una macchina di Turing monodimensionale che determini se il numero di a in memoria e' pari al numero di b .

Soluzione:

Vediamo ora una soluzione per questo problema. Innanzitutto, vediamo che alfabeto useremo:



Figura 7.2

La nostra filosofia sara' quella di cercare una a verso destra, sostituirla con $*$, tornare all'inizio, cercare una b verso destra, sostituirla con $*$, tornare all'inizio e cosi' via. I limiti della memoria che ci fanno capire di essere ad uno dei due estremi della sequenza sono i caratteri blank. Se il numero di a risulterà diverso dal numero di b , il programma ciclerà all'infinito sui caratteri blank, altrimenti si arrestera'.

Vediamo adesso gli stati che vogliamo usare ed il loro scopo:

- $q_{r,a}$ = Cerca una a andando verso destra
- $q_{r,b}$ = Cerca una b andando verso destra
- $q_{r,a,s}$ = Torna all'inizio (va a sinistra finche' non trova blank), poi cerchera' una a
- $q_{r,b,s}$ = Torna all'inizio (va a sinistra finche' non trova blank), poi cerchera' una b
- $q_{r,b \text{ senza } a}$ = Torna all'inizio (va a sinistra finche' non trova blank), poi cerchera' una a

Detto come funziona l'algoritmo, vediamo l'implementazione vera e propria:

Configurazione da cui parto		Azioni che commetto e stato cui vado		
Stato	Cosa leggo	Cosa scrivo	Nuovo stato	Direzione testina
$q_{r,a}$	b	b	$q_{r,a}$	D
$q_{r,a}$	a	*	$q_{r,b,s}$	S
$q_{r,b,s}$	a	a	$q_{r,b,s}$	S
$q_{r,b,s}$	b	b	$q_{r,b,s}$	S
$q_{r,b,s}$	*	*	$q_{r,b,s}$	S
$q_{r,b,s}$	blank	blank	$q_{r,b}$	D
$q_{r,b}$	a	a	$q_{r,b}$	D
$q_{r,b}$	*	*	$q_{r,b}$	D
$q_{r,b}$	b	*	$q_{r,b,s}$	S
$q_{r,a,s}$	b	b	$q_{r,a,s}$	S
$q_{r,a,s}$	a	a	$q_{r,a,s}$	S
$q_{r,a,s}$	*	*	$q_{r,a,s}$	S
$q_{r,a,s}$	blank	blank	$q_{r,a}$	D
$q_{r,a}$	blank	blank	$q_{r,b \text{ senza } a}$	S
$q_{r,b}$	blank	blank	$q_{r,b}$	D (a > b)
$q_{r,b \text{ senza } a}$	*	*	$q_{r,b \text{ senza } a}$	S
$q_{r,b \text{ senza } a}$	b	b	$q_{r,b \text{ senza } a}$	S

8^a Lezione

Questo capitolo sara' interamente dedicato all'approfondimento di uno dei teoremi piu' importanti della teoria della calcolabilita', ovvero il **teorema del parametro**.

8.1 Significato del teorema del parametro

Questo teorema ci permette di costruire funzioni che forniscono come output programmi. Il classico esempio che si mostra per capire il significato di questo teorema e' il seguente:

si prenda una funzione $f(x,y) : N^2 \rightarrow N$ tale che $f(x,y) = y^x$. Allora, se teniamo fisso x (diremo che x e' un **parametro**), $f(x,y) = f_x(y)$ diventa una funzione nella sola variabile y . Ad esempio, per $x=2$ ottengo $f_2(y) = y^2$, per $x=3$ $f_3(y) = y^3$, e cosi' via. A questo punto, ci chiediamo se esiste una certa funzione calcolabile S , che dato in input un certo naturale n , mi fornisca in output **un programma** che calcoli $f_n(y) = y^n$. Tale funzione esiste per il teorema del parametro che enunceremo e dimostreremo nel corso del paragrafo 8.2.

Quindi, nel caso della nostra funzione $f(x,y) = y^x$, esistera' una certa funzione $S : N \rightarrow N$ calcolabile tale che:

$S(2) = \text{programma che calcola } f_2(y) = y^2$

$S(3) = \text{programma che calcola } f_3(y) = y^3$

.....

$S(n) = \text{programma che calcola } f_n(y) = y^n$

dove ricordiamo che $S(x)$, essendo un programma, viene codificato come numero naturale.

Da questo possiamo dedurre che $\Phi_{S(x)}(y) = f(x,y)$, cioe' applicando la macchina universale al programma prodotto da $S(x)$ ed al relativo input y , otteniamo l'output della funzione $f(x,y)$.

Da cio' che si e' detto finora risulta chiaro, inoltre, che partendo ad esempio da un programma con 2 variabili, possiamo generare, tramite parametrizzazione, infiniti programmi con una sola variabile.

8.2 Enunciato e dimostrazione del teorema del parametro

Teorema:

Data una funzione $f(x,y)$ calcolabile, esiste una funzione totale calcolabile $S : N \rightarrow N$ che parametrizza f , cioe' tale che $\Phi_{S(x)}(y) = f(x,y)$. Questo teorema e' valido anche se le variabili sono piu' di due.

Dim:

Per dimostrare questo teorema non faremo altro che trovare un programma che costruisca la funzione S .

Prendiamo un programma di una macchina a registri $P = R_1, \dots, R_m$ che implementi la nostra funzione, calcolabile, $f(x,y)$. Prendiamo una funzione S tale che $S(x)$ sia la codifica del programma che prende la seguente configurazione iniziale dei registri

y	0	0	0	0	0	0	0	...
---	---	---	---	---	---	---	---	-----

e pone in R_1 , dopo la computazione, $f(x,y)$ come risultato finale.

A questo scopo, innanzitutto utilizzeremo le due istruzioni:

- $x2 := x1$
- $x1 := 0$

ottenendo così la configurazione:

0	y	0	0	0	0	0	0	0	...
---	---	---	---	---	---	---	---	---	-----

Ora, dal momento che voglio computare $S(x)$, dovrò scrivere x nel primo registro. Per questo eseguirò x volte l'istruzione:

- $x1 := x1 + 1$

ottenendo così la seguente configurazione:

x	y	0	0	0	0	0	0	0	...
---	---	---	---	---	---	---	---	---	-----

Ora uso l'ipotesi del teorema, cioè che la nostra $f(x,y)$ sia calcolabile (e quindi vi sia un programma che la calcoli, che abbiamo detto essere P). Utilizzo tale programma per computare $f(x,y)$, con x e y che sono già pronte nei primi due registri della macchina. Dal momento che le istruzioni I_1, \dots, I_n di P devono venire eseguite dopo che sono già state eseguite le istruzioni di cui sopra, bisognerà opportunamente cambiare il valore degli indici di queste istruzioni ed il valore dei GOTO di P che adesso dovranno puntare a nuovi indici.

Quindi, ad esempio avremo che:

$$S(2) = \begin{cases} \begin{array}{l} x2 := x1 \\ x1 := 0 \\ x1 := x1 + 1 \\ x1 := x1 + 1 \\ I_1' \\ \dots \\ I_n' \end{array} & \text{indici di} \\ & \text{registri e goto} \\ & \text{cambiati} \end{cases}$$

quindi è chiaro che $S(x)$ fornisce in output la codifica di un programma per calcolare $f(x,y)$ dato il parametro x .

Vediamo ora un esempio concreto sull'argomento. Consideriamo la funzione $f(x,y) = 0$ per ogni x,y . Questa funzione è calcolabile, e un programma che la calcola potrebbe essere $P = \{x1 := 0\}$, per cui avremo, ad esempio:

$$S(1) = \begin{cases} \begin{array}{l} x2 := x1 \\ x1 := 0 \\ x1 := x1 + 1 \\ x1 := 0 \end{array} & \text{indici di} \\ & \text{registri e goto} \\ & \text{cambiati} \end{cases}$$

Solitamente il teorema del parametro si utilizza per dimostrare la calcolabilità di un certo programma P che prende in input altri programmi (x,y, \dots) e fornisce in output un nuovo programma P' . Cioè si fa ponendo i programmi (x,y, \dots) in input come parametri, di modo che l'output di $S(x,y, \dots)$ sia il nostro P' . Facciamo un esempio concreto:

8.3 Esempio 1. Prodotto di due programmi.

Consideriamo il seguente problema. Vogliamo costruire un programma $P : N^2 \rightarrow N$ che, dati in ingresso due programmi $(x$ che computa $f(z)$ ed y che computa $g(z)$), mi restituisca in output un programma che computi il prodotto $f(z)*g(z)$. Ovviamente, costruire da zero tale programma per determinate funzioni f e g sarebbe complicato, invece usando il teorema del parametro tale operazione può essere semplificata.

Sappiamo che la premessa necessaria anche il teorema del parametro sia applicabile, è che esista una funzione di partenza $f(x,y)$ calcolabile (di cui si conosce il programma). Nel nostro caso, la funzione di partenza conterrà 3 variabili, e quindi sarà del tipo $h(x,y,z)$, in cui x,y , corrispondenti ai programmi per $f(z)$ e $g(z)$, saranno i nostri parametri, mentre z sarà l'unica variabile "vera" da considerare.

Dobbiamo quindi dimostrare che $h(x,y,z)$ sia calcolabile. Per questo lo scriviamo tale funzione come:

$$h(x,y,z) = \Phi_x(z)*\Phi_y(z) = \Phi(x,z)*\Phi(y,z)$$

A questo punto, immaginando di conoscere il programma Φ per la macchina universale, la nostra $h(x,y,z)$ sarà computata da:

$$h(x,y,z) = (P^3_{1,1} \wedge P^3_{3,3} \wedge P^3_{2,2} \wedge P^3_{3,3}) ; (\Phi \parallel \Phi) ; *$$

quindi il teorema del parametro e' applicabile. Cio' vuoldire che esistera' una certa funzione $S(x,y) : N^2 \rightarrow N$ tale che $\Phi_{S(x,y)}(z) = h(x,y,z)$.

Ora, conoscendo $h(x,y,z)$, applicando la dimostrazione del teorema potremo costruire tutti i programmi calcolati da $S(x,y)$. Cioe' fissando le nostre 2 funzioni f,g come parametro, avremo subito di risposta un programma che ci calcolera' $f(z)*g(z)$. Ad esempio, se $f(z) = z+1$ e $g(z) = z+2$, da $S(x,y)$ ricaveremo subito il programma che calcolera' $(z+1)*(z+2)$, come fatto nell'esempio finale del paragrafo 8.2.

Quindi, in definitiva, il teorema del parametro ci serve per individuare classi di programmi "figli" di un programma piu' generale.

8.4 Esempio 2. Composizione di programmi

Iniziamo una piccola serie di esempi nei quali ci chiediamo come si possono calcolare alcuni fondamentali schemi di programmazione funzionale. Iniziamo con lo schema di composizione di funzioni $(;)$ che, come sappiamo, prende in input due programmi (x,y) e li trasforma in un unico programma che opera la composizione di funzioni. Quindi la nostra funzione di partenza avra', come quella vista nel paragrafo 8.3, 3 variabili di cui 2(x e y) saranno parametri, mentre una (z) sara' la variabile vera e propria.

Per risolvere questo problema con l'ausilio del teorema del parametro, cominciamo, innanzitutto, con lo scrivere la nostra funzione $h(x,y,z)$ di partenza:

$$h(x,y,z) = \Phi_y(\Phi_x(z)) = \Phi(y, \Phi(x,z))$$

come sappiamo, questa funzione deve essere calcolabile perche' il teorema sia applicabile. Cio' si puo' dimostrare, semplicemente, fornendo un programma che la computi. Questo programma puo' essere:

$$h(x,y,z) = (P^3_{2,2} \wedge ((P^3_{1,1} \wedge P^3_{3,3}) ; \Phi)) ; \Phi$$

A questo punto sappiamo che il teorema del parametro e' applicabile, per cui esistera' una certa funzione $S(x,y)$ che prende in ingresso I programmi che computano le 2 funzioni da comporre, e restituisce in uscita il programma che computa $(f \circ g)(z)$ per quelle due particolari funzioni f e g . Quindi, in questo modo, ho costruito tutta la classe dei programmi che computa tutte le composizioni di funzioni possibili in natura.

9^a Lezione

Vediamo ora un ultimo esempio di applicazione del teorema del parametro ai costrutti funzionali. Nell'occasione, cercheremo di implementare una classe di programmi per computare tutte le possibili funzioni $\exp(s)$.

9.1 Esponenziazione di programmi

Data una funzione $f : N \rightarrow N$ calcolabile, di cui possediamo il programma x che la computa, cerchiamo di fornire un programma per $\exp(f)(z,y) = f^y(z)$. Quindi, in definitiva, dato in input un programma che calcola f , voglio in output un programma che calcola $\exp(f)$.

Si vede come questo problema sia molto simile ai due già visti nel capitolo 8, per cui è intuitivo usare il teorema del parametro. In questo caso, la nostra funzione da cui partiamo avrà però un solo parametro (x) e due variabili vere e proprie (z,y). Quindi, essa potrà essere scritta come:

$$h'(x, z, y) = \Phi_{S(x)}(z, y) = ((\Phi_x(z))^y \quad \text{dove abbiamo chiamato la funzione } h' \text{ e non } h \text{ per non confonderla con} \\ \text{quella della ricorsione primitiva che vedremo tra poco.}$$

Come al solito, dobbiamo dimostrare che $h'(x, z, y)$ è computabile. Per far questo, come è ormai consuetudine, cerchiamo di trovare un programma che computi tale funzione. In questo caso, operiamo per ricorsione primitiva:

$$(\Phi_x(x))^y = h'(x, z, y) = \begin{cases} h'(x, z, 0) = g(x, z) = z \\ h'(x, z, y+1) = \Phi_x((\Phi_x(x))^y) \end{cases} \xrightarrow{\hspace{1cm}} \begin{cases} g(x, z) = z \\ h(x, z, y, u) = \Phi(x, u) \end{cases}$$

Figura 9.1

sia $g(x,z)$ che $h(x,z,y,u)$ sono facilmente computabili in programmi funzionali, dal momento che supponiamo conoscere il programma che computa la macchina universale.

Quindi il teorema del parametro è applicabile. Da questo possiamo dedurre che esiste $S : N \rightarrow N$ che, dato un programma x che computa una certa funzione f in ingresso, mi genera un programma che computi $\exp(f)$ per quella particolare funzione f .

9.2 Funzioni che tendono ad infinito molto rapidamente

Vogliamo ora costruire una funzione totale che tenda ad infinito più velocemente di qualsiasi funzione esponenziale. Dal momento che una funzione esponenziale può essere vista come un unico ciclo FOR, concludiamo

che un ciclo FOR non sarà sufficiente a computare tale funzione, ma vedremo che sarà indispensabile adoperare un ciclo WHILE-DO.

Costruisco ora la seguente successione di funzioni calcolabili, tutte del tipo $f_i : N \rightarrow N$

Figura 9.2

$$\begin{aligned}
 f_0 &= S \text{ (successore)} && \text{Duplica la memoria} \\
 f_1 &= D2 ; \exp(f_0) && \text{da cui abbiamo } x \xrightarrow{D2} x, x \xrightarrow{\exp(s)} 2x \\
 f_2 &= D2 ; \exp(f_1) && \text{da cui abbiamo } x \xrightarrow{D2} x, x \xrightarrow{f_1} 2x \xrightarrow{D2} 4x \xrightarrow{f_1} \dots \xrightarrow{f_1} 2^{x_x} x \\
 f_3 &= D2 ; \exp(f_2) && \text{da cui abbiamo } x \xrightarrow{D2} x, x \xrightarrow{f_2} 2x \xrightarrow{f_2} 2^{2^x} x \xrightarrow{f_2} \dots \xrightarrow{f_2}
 \end{aligned}$$

$$f_n = D2 ; \exp(f_{n-1}) = D2 ; \exp(D2 ; \exp(\dots(D2 ; \exp(S)) \dots))$$

f_n , nonostante sia composta da tantissimi cicli annidati, non è la funzione che tende ad infinito più rapidamente. Infatti, se prendiamo $g : N \rightarrow N$ tale che $g(x) = f_n(x)$, con f_n funzione di figura 9.2, questa tenderà ad infinito ancora più rapidamente. In pratica $g(100)$ sarà composta da 100 cicli FOR annidati, $g(200)$ da 200 cicli FOR annidati e così via.

Si può dimostrare che $g(x)$ non è calcolabile soltanto il costrutto FOR, ma abbiamo bisogno di un WHILE, in quanto il FOR è un costrutto finito, mentre noi abbiamo bisogno di un costrutto che possa ciclare anche indefinitamente, visto che le f_i sono teoricamente di numero infinito.

9.3 Un esercizio sui costrutti funzionali

Utilizzando i costrutti funzionali, troviamo un programma per:

$$\text{div}(x,y) = \begin{cases} 1 & \text{se } x \text{ divide } y \\ & \text{con } 0 \text{ che divide } 0 \\ 0 & \text{altrimenti} \\ & \text{ma } 0 \text{ non divide } y \text{ con } y > 0 \end{cases}$$

sappiamo che x divide y se $\text{rm}(x,y) = 0$. Quindi avremo, facilmente, $\text{Div}(x,y) = (\text{rm} ; !\text{sg})$

9.4 Insiemi decidibili

Un insieme $A \subseteq N$ si dice **decidibile** (o ricorsivo) se esiste un programma, un automa, ecc che può calcolare la **funzione caratteristica** di A , definita come:

$$\chi_A(x) = \begin{cases} 1 & \text{se } x \in A \\ 0 & \text{se } x \notin A \end{cases}$$

In pratica, questo programma, automa, ecc deve essere in grado di decidere se un elemento appartiene ad un insieme o meno. Esempi di insiemi di questo tipo sono gli intervalli, i numeri pari, dispari: quelli, insomma, calcolabili da un programma "riconoscitore" che decide se un elemento appartiene ad un insieme o no (come un automa di una grammatica tipo 3).

10^a Lezione

Alla fine dello scorso capitolo, avevamo visto la definizione di insieme decidibile. Osserviamo che se prendiamo due insiemi B e C, con B contenuto in C, in generale non sappiamo nulla sulla loro decidibilità. Potremmo avere B decidibile e C no, C decidibile e B no, entrambi decidibili o nessuno decidibile. Quindi, non possiamo dedurre, per un insieme, se esso è decidibile partendo da un suo sottoinsieme o viceversa. Questo perché, togliendo o aggiungendo elementi ad un insieme, ne viene cambiata la composizione, e quindi l'algoritmo che calcola la sua funzione caratteristica (sempre che questo esista).

10.1 Decidibilità di codomini di funzioni

In generale, gli output di un programma non è detto che costituiscano un insieme decidibile. Vi sono comunque delle funzioni di cui si può affermare subito la decidibilità del codominio, come si vede dal seguente teorema:

Teorema:

Se una funzione $f: N \rightarrow N$ è crescente e totale allora il suo codominio è decidibile.

Dim:

Per sapere se un certo $y \in N$ fa parte del codominio della funzione f , è sufficiente calcolare $f(0), f(1), \dots, f(n), \dots$ sino a che $f(n) = y$ (in questo caso y appartiene al codominio) o $f(n) > y$ (in questo caso y non appartiene al codominio).

10.2 Decidibilità dell'insieme dei programmi che terminano su tutti gli input

Di seguito, cercheremo di dimostrare che l'insieme $T = \{n : \Phi_n \text{ è totale}\}$ dei programmi che terminano su tutti gli input, non è decidibile. Per questo, prendiamo la funzione $h(x,y)$ del problema della fermata, che abbiamo visto essere non calcolabile:

$$h(x,y) = \begin{cases} 1 & \text{se } Px \downarrow y \\ 0 & \text{se } Px \uparrow y \end{cases} \quad g(x) = h(x,x) = \begin{cases} 1 & \text{se } Px \downarrow x \\ 0 & \text{se } Px \uparrow x \end{cases}$$

Figura 10.1

Ora, dal momento che $h(x,y)$ non è calcolabile, allora non lo sarà nemmeno la funzione $g(x) = h(x,x)$ di cui sopra. Ma $g(x)$ è la funzione caratteristica dell'insieme $k = \{x : Px \downarrow x\}$ dei programmi definiti su se stessi. Dal momento che $g(x)$ non è calcolabile, allora non esiste un programma in grado di dire se un programma sta o no in k , quindi k è non decidibile.

Dimostriamo ora che $T \subseteq k$, il che ci potrebbe essere d'aiuto in seguito. Per far ciò facciamo vedere che esiste un elemento che sta in k ma non sta in T (mentre è chiaro che ogni elemento di T sta in k , in quanto se una funzione è totale allora è anche definita su se stessa). Consideriamo la seguente funzione $f(x)$:

$$f(x) = \begin{cases} \uparrow & \text{se } x = 0 \\ 1 & \text{se } x > 0 \end{cases}$$

Questa funzione e' chiaramente calcolabile (ad esempio da una macchina a registri), per cui essa sara' calcolata da un certo programma P la cui codifica e' $\lceil P \rceil = a$. Ora, dal momento che Pa calcola una funzione non totale, in quanto Pa non e' definito sullo zero, abbiamo che $a \notin T$. Quindi, se facciamo vedere che $a \in k$, cioe' che a e' definito su se stesso, la nostra dimostrazione termina. Ma cio' avviene sicuramente, a meno che $a=0$. Ma in quest'ultimo caso basterebbe aggiungere a Pa una istruzione "inutile" (tipo $x1:=x1$) per cambiare la codifica di a, che diventerebbe cosi' diverso da zero. Quindi $a \in k$, e cio' implica che $T \subseteq k$, che e' cio' che volevamo

Torniamo adesso alla nostra dimostrazione della non decidibilita' di T. Consideriamo la seguente funzione:

$$f(x) = \begin{cases} \Phi_x(x) + 1 & \text{se } \Phi_x \text{ e' totale} \\ 0 & \text{altrimenti} \end{cases}$$

Dimostriamo anzitutto che questa funzione non e' calcolabile. Ipotizziamo, per assurdo, che $f(x)$ sia calcolabile. Cio' implicherebbe l'esistenza di un certo programma $P_m = \Phi_m$ che la calcoli. Ora, dal momento che $f(x)$ e' una funzione totale, allora anche Φ_m e' totale, per cui, in particolare, potremo calcolare $\Phi_m(m)$. Ma, per definizione di $f(x)$, avremmo $\Phi_m(m) = \Phi_m(m) + 1$ il che e' assurdo. Da cio' concludiamo che $f(x)$ non e' calcolabile.

Supponiamo ora che T sia decidibile. Se dimostriamo che $f(x)$ e' calcolabile partendo da questa ipotesi, otteniamo una contraddizione e quindi dimostriamo che T e' non decidibile.

Poniamo dunque che T sia decidibile. Allora possiamo sempre stabilire se un programma termina o meno su tutti gli input. Quindi, preso un qualunque programma x, lo passiamo ad un programma P che "decide" T (che mi dice, cioe', se Φ_x e' totale): se il risultato sara' 1, allora Φ_x sara' totale, viceversa nel caso il risultato sia zero. Nel primo caso, avremo che il programma Px sara' definito su tutti gli input, quindi in particolare su x stesso. Da cio' ricaviamo che P e' una quasi perfetta implementazione della nostra $f(x)$, in quanto, dato un programma x in ingresso, mi puo' fornire con facilita' in uscita $\Phi_x(x)$ se Φ_x e' totale. A questo punto, aggiungendo un 1 a questo risultato e fornendo in uscita zero qualora x non fosse totale, si ottiene un programma per $f(x)$, che pero' abbiamo dimostrato essere non calcolabile. Questa e' una contraddizione che ci fa concludere dicendo che **T e' non decidibile**.

10.3 Funzioni calcolabili con codomini non decidibili

Vediamo un esempio di funzione calcolabile il cui codominio non e' decidibile:

$$f(x) = \begin{cases} x & \text{se } x \in k \quad (\text{cioe' } Px \downarrow x) \\ \uparrow & \text{altrimenti} \end{cases}$$

Poiche' k e' non decidibile, questa funzione sembrerebbe non calcolabile, in quanto non riusciamo a sapere se un certo x appartiene o no a k. Invece, questa funzione si puo' calcolare senza sfruttare l'appartenenza a k. Per farlo e' sufficiente prendere un certo $x \in N$, decifrarlo col corrispondente programma Px e passare lo stesso input x a Px. A questo punto la computazione potra' terminare o meno. Nel primo caso deduciamo che $Px \downarrow x$, per cui $x \in k$, altrimenti si entra in ciclo. Ma fortuna vuole che per questa funzione si dovesse proprio entrare in ciclo nel caso $x \notin k$, quindi abbiamo dimostrato che $f(x)$ e' calcolabile nonostante il suo codominio non sia decidibile.

10.4 Insiemi semidecidibili

Parallelamente a quanto fatto per gli insiemi decidibili, definiamo adesso gli insiemi semidecidibili. Successivamente analizzeremo le proprieta' di tali insiemi e vedremo quale nesso vi sia tra insiemi decidibili ed insiemi semidecidibili.

Un insieme $A \subseteq N$ si dice **semidecidibile** se la seguente funzione, detta **funzione semicaratteristica** di A, e' calcolabile:

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

da questa definizione osserviamo subito come la funzione $f(x)$ vista nel paragrafo 10.3 non sia altro che la funzione caratteristica di k , con la piccola differenza dell'output che invece di essere 1 e' x nel caso $x \in A$. Da cio' deduciamo che, dal momento che $f(x)$ era calcolabile, allora deve esserlo anche $X_k(x)$. Quindi k e' **semidecidibile**. Si puo' dimostrare, invece, che T non e' nemmeno semidecidibile.

10.5 Relazione tra insiemi decidibili e semidecidibili

Teorema:

Un insieme $A \subseteq N$ e' decidibile se e solo se A e \bar{A} sono semidecidibili.

Dim (\rightarrow):

Sia A decidibile, vogliamo dimostrare che A e \bar{A} sono semidecidibili. Un algoritmo che semidecide A e' quello che decide A modificato con qualche istruzione in modo che entri in ciclo quando dovrebbe restituire 0 (ad esempio mettendo delle istruzioni tipo "1 Goto 1"). Un programma che semidecide \bar{A} sara' un programma che decide A modificato in modo che entri in ciclo quando $x \in A$ e restituisca 1 se $x \notin A$.

Dim (\leftarrow):

Siano A e \bar{A} semidecidibili. Facendo lavorare in parallelo I due programmi per semidecidere A e \bar{A} , uno dei due sicuramente termina e ci dice se $x \in A$ o $x \notin A$.

Facciamo adesso una rapida carrellata sulle proprieta' che derivano da quelle appena viste:

- Dal momento che k non e' decidibile, ma e' semidecidibile, allora \bar{k} non e' semidecidibile.
- Si puo' dimostrare che T e \bar{T} non sono semidecidibili.
- Se A e' decidibile allora lo e' anche \bar{A} .

Per dimostrare che \bar{T} non e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile. Per dimostrare che T non e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se A e' decidibile allora \bar{A} e' decidibile basta provare a semidecidere \bar{A} e se si riesce a farlo allora A e' decidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

Per dimostrare che se \bar{T} e' semidecidibile allora T e' semidecidibile basta provare a semidecidere T e se si riesce a farlo allora \bar{T} e' semidecidibile.

Per dimostrare che se T e' semidecidibile allora \bar{T} e' semidecidibile basta provare a semidecidere \bar{T} e se si riesce a farlo allora T e' semidecidibile.

11^a Lezione

Vediamo adesso di dimostrare la decidibilità (o la non decidibilità) di un altro insieme, e cioè dell'insieme dei programmi che terminano su almeno un input.

11.1 Decidibilità dell'insieme dei programmi che terminano su almeno un input

Consideriamo l'insieme $A = \{x : (\exists y) \text{ } Px \downarrow y\}$ dei programmi che terminano su almeno un input. Parallelamente, scriviamo anche l'insieme $\text{!}A = \{x : (\forall y) \text{ } Px \uparrow y\}$ dei programmi che non terminano per nessun input, ovvero **l'insieme dei programmi che calcolano la funzione vuota**.

11.1.1 Semidecidibilità di A

Iniziamo cercando di dimostrare che A è **semidecidibile**. In tal caso, deve accadere che la funzione semicaratteristica di A sia calcolabile. Il primo problema che ci troviamo ad affrontare, nel trovare un algoritmo che decide A (che guarda cioè se un programma termina per qualche input) consiste nel fatto che non sappiamo a priori per quali valori di x un programma termina o va in ciclo. Questo è un problema perché, qualora il programma andasse in ciclo per un certo x , non potremmo mai sapere se, ad esempio per $x+1$, esso avrebbe potuto terminare, perché saremmo rimasti bloccati. Per risolvere questo problema, l'idea è quella di prendere il nostro processore e di fargli eseguire il programma per un certo quanto di tempo (ad esempio 10 minuti) su un certo input. Se il programma termina entro il quanto di tempo allora c'è un input su cui il programma termina, e quindi il programma sta in A , altrimenti si cambia

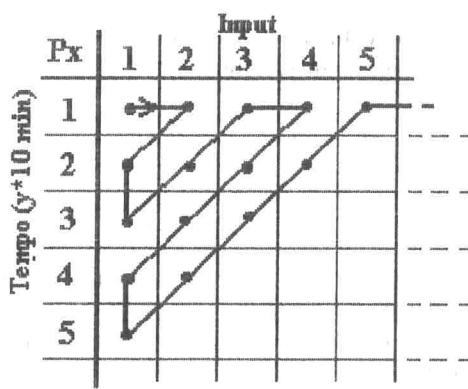


Figura 11.1

input e/o si aumenta la dimensione del quanto di tempo come illustrato nella seguente tabella di dimensioni infinite:

Quindi, si parte col testare il programma sull'input 1 con quanto di tempo di 10 minuti. Se il programma, in questa configurazione, termina, allora si conclude che il programma sta in A , altrimenti si prosegue testando il programma con l'input 2 e con quanto di tempo pari a 10 minuti. Se anche in questa occasione il programma non termina, si passa all'input 1 con quanto di tempo di 20 minuti, e così via, seguendo il percorso di figura 11.1. La prima occasione in cui il programma terminerà sarà quella che ci farà concludere che detto programma sta in A .

In questo modo, riusciamo a testare il programma su tutti gli input possibili senza il pericolo di bloccarci su valori indefiniti.

11.1.1 Semidecidibilità di $\neg A$

Cercheremo ora di dimostrare che $\neg A$ non è semidecidibile. Per far questo, partiremo dal presupposto che A sia decidibile, per arrivare ad una contraddizione. Così dimostreremo anche che A non è decidibile.

Iniziamo da qualche considerazione che potrà aiutare nella comprensione della dimostrazione. Innanzitutto, notiamo che tutti i programmi che terminano su se stessi terminano su almeno un input, mentre non è vero il viceversa. Quindi abbiamo che $k \subseteq A$, e di conseguenza $\neg A \subseteq \bar{k}$. Ciò può essere riassunto dalla seguente figura:

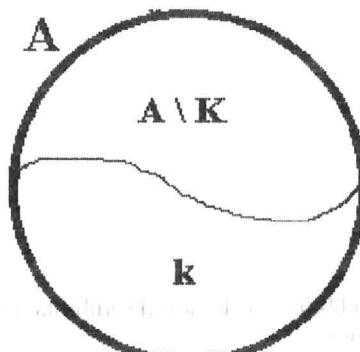


Figura 11.2

Ora, supponiamo che esista una **trasformazione** (funzione) calcolabile, che prendendo un certo programma x in ingresso, lo trasformi in un altro programma x' in uscita, nel modo descritto dalla figura 11.3:

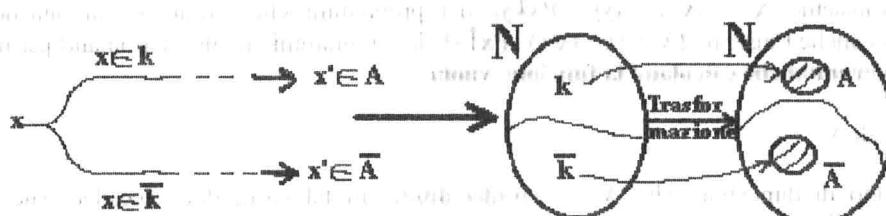


Figura 11.3

Dal momento che abbiamo supposto A decidibile, allora, preso un programma x , e applicatogli questa trasformazione, otengo un certo programma x' , il quale può appartenere o meno ad A . Nel caso $x' \in A$ (e dal momento che A è supposto decidibile posso saperlo) allora, tornando indietro, ricavo che $x \in k$. Viceversa, se $x' \notin A$, allora $x \notin k$. Quindi abbiamo trovato un programma che decide k , che però abbiamo dimostrato essere non decidibile. Questa contraddizione ci porta a dire che A non è decidibile (e quindi $\neg A$ non è semidecidibile).

A questo punto, per completare la dimostrazione, ci manca la trasformazione di cui abbiamo tanto parlato. L'idea è quella di partire dalla seguente funzione a due variabili:

$$f(x,y) = \begin{cases} x & \text{se } x \in k \\ \uparrow & \text{altrimenti} \end{cases}$$

questa funzione è esattamente analoga a quella vista nel paragrafo 10.3, che avevamo visto essere calcolabile. Quindi, essendo questa funzione calcolabile, posso applicare ad essa il teorema del parametro. Esisterà quindi una funzione $S(x)$ calcolabile totale tale che $\Phi_{S(x)}(y) = f(x,y)$. Ora, con grande sorpresa scopriamo che $S(x)$ è proprio la trasformazione che stavamo cercando, infatti:

- $x \in k \rightarrow \Phi_{S(x)}(y) = x \quad (\forall y) \rightarrow S(x) \in A$ Perché $S(x)$ termina sempre la computazione, quindi tanto più su un solo input
- $x \notin k \rightarrow \Phi_{S(x)}(y) = \uparrow \quad (\forall y) \rightarrow S(x) \notin A$ Perché $S(x)$ non termina mai la computazione, quindi questo è un programma per la funzione vuota.

Quindi $S(x)$ si comporta proprio nel modo che volevamo dalla nostra trasformazione, e quindi la dimostrazione è completa.

Concludiamo questo paragrafo con una osservazione. Per compiere questo tipo di dimostrazioni, ci comporteremo spesso in questo modo, cioè aggiungendo una variabile a funzioni già conosciute in modo da poter applicare il teorema del parametro ed ottenere così funzioni calcolabili totali che soddisfino le nostre esigenze. Vedremo in seguito altri esempi a proposito.

11.2 Insiemi riducibili

Dati $A, B \subseteq N$, A è **riducibile a B** ($A \leq_T B$) se esiste una funzione calcolabile totale $s : N \rightarrow N$ tale che:

- $x \in A \Rightarrow s(x) \in B$
- $x \notin A \Rightarrow s(x) \notin B$

questa proprietà può essere riassunta dalla figura sottostante:

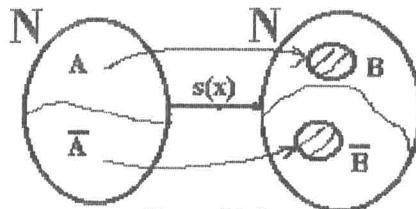


Figura 11.4

Teorema:

Consideriamo ($A \leq_T B$). Allora:

1. Se B è decidibile allora anche A è decidibile
2. Se B è semidecidibile allora anche A è semidecidibile
3. Se A non è decidibile allora anche B non è decidibile
4. Se A non è semidecidibile allora anche B non è semidecidibile

Dim (1):

In pratica viene riproposto il procedimento della dimostrazione che abbiamo visto nel paragrafo 11.1.2. Se B è decidibile, allora, un programma per decidere A prenderà un qualunque $x \in N$ e lo darà in pasto alla nostra funzione $s(x)$. Il risultato potrà stare o non stare in B (e questo verrà scoperto dal programma che decide B). Qualora $s(x) \in B$, allora, tornando indietro col ragionamento, si potrà affermare che $x \in A$. Altrimenti, se $s(x) \notin B$, si dedurrà che $x \notin A$. Quindi abbiamo trovato un programma per decidere A .

Dim (2):

Se B è semidecidibile, allora, un programma per semidecidere A compierà le seguenti operazioni: preso un qualunque $x \in N$, lo darà in pasto alla nostra funzione $s(x)$. Il risultato potrà stare o non stare in B (e questo verrà scoperto dal programma che semidecide B). Qualora $s(x) \in B$, allora, tornando indietro col ragionamento, si potrà affermare che $x \in A$. Altrimenti, il programma che semidecide B entrerà in un ciclo infinito, di conseguenza anche il "programma padre" che semidecide A entrerà in un ciclo infinito, e ciò mostra la correttezza del nostro programma che semidecide A .

Dim (3):

Se A non è decidibile, allora nemmeno B lo è in quanto, se lo fosse, A sarebbe decidibile per (1), il che porta ad una contraddizione.

Dim (4):

Se A non è semidecidibile, allora nemmeno B lo è in quanto, se lo fosse, A sarebbe semidecidibile per (2), il che porta ad una contraddizione.

11.3 Esercizi sulla riducibilità di insiemi

Vediamo ora qualche esercizio sulla riducibilità degli insiemi.

1. Testo:

Sia P l'insieme dei numeri pari, $k \leq_T P$?

Svolgimento:

No. Infatti, essendo k non decidibile e P decidibile, non può esistere una trasformazione $s(x)$ adeguata alla definizione di riducibilità, perché non rispetterebbe i punti del teorema del paragrafo 11.2.

2. Testo:

Sia P l'insieme dei numeri pari, e D l'insieme dei numeri dispari. Allora $P \leq_T D$?

Svolgimento:

Si, in quanto, se considero la funzione $s(x) = x+1$, questa è una funzione calcolabile totale tale che :

- $x \in P \Rightarrow s(x) \in D$
- $x \in !P$ (cioè $x \in D \Rightarrow s(x) \in !D$ (cioè $x \in P$)

Cio' puo' essere riassunto dalla seguente figura:

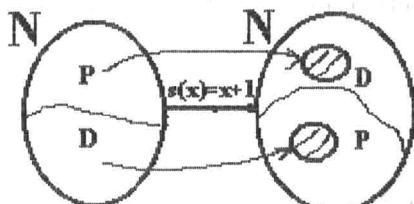


Figura 11.5

3. Testo:

Siano A e B insiemi decidibili. Allora $A \leq_T B$?

Svolgimento:

Questo problema si riduce essenzialmente nel trovare una adeguata trasformazione $s(x)$. Per questo, preso un qualsiasi $x \in N$, lo si passera' al programma che decide A . A questo punto, se $x \in A$, si fara' in modo che $s(x) = c_0$, con $c_0 \in B$ preso a caso tra gli elementi di B . Parallelamente, se $x \in !A$, si fara' in modo che $s(x) = c_1$, con $c_1 \in !B$ preso a caso tra gli elementi di $!B$. Cio' puo' essere riassunto dalla figura 11.6:

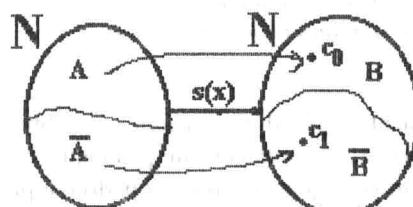


Figura 11.6

In pratica $s(x)$ si puo' riassumere nella composizione di due funzioni costanti. Ovviamente questo ragionamento si puo' fare solo se $B \neq N \neq \emptyset$, altrimenti mi troverei a dover cercare c_0 o c_1 nell'insieme vuoto, il che e' impossibile.

In questo caso, non ci siamo preoccupati di scoprire la vera identita' di c_0 o c_1 in quanto a noi interessa solo vedere che c'e' un programma non scriverlo.